Nucleus Networking

Version v1

Generated 2026-10-18 11:53:23 UTC

Documentation for Nucleus, a .NET networking engine with a Unity integration.

Contents

Client state 11
Controller retention 16
Reading who controls a system 19
Control, Client State and Prediction 22
Predicted members and prediction history 23
Reconciliation and replay 27
What control means 36
Assigning and releasing control 38
When prediction is worth its cost 39
Authenticating clients 41
Connections 44
Creating a CoreManager 48
Driving the loop yourself 51
Host mode 55
Core, Managers and the Tick Loop 57
Kicking and disconnecting a client 58
The manager map 61
The network loop steps 63
Keeping the world across a disconnect 66
Server, client and host roles 68
Kick and refusal notices 70
Session events 72
Shutting a session down 75
Threading and lifetime 77
Ticks, tick rate and time 79
Common errors and what causes them 82
Diagnosing desync 85
Frequently asked questions 88
Filing a bug report 90
Diagnostics and Troubleshooting 91
Reading serialization history (and the raw-packet test hook) 92
Round-trip time, jitter and packet loss 94
Logging 96
Network statistics 99
Technical limitations 101
Violation reference 102
Violations 106
What the server trusts 109
Core API 111

Contents

What area of interest is 112
Distance interest conditions 114
The host interest report 116
Interest Management and Scaling 118
Interest level of detail 119
How distances are measured: pairwise and the spatial grid 121
An object is missing, or will not go away 123
Registering interest conditions 126
Spawn pacing 128
Stopping versus culling 130
Supplying positions to the interest system 133
What scales and what does not 135
Writing an interest condition 138
Capturing a shot 141
How lag compensation works 143
Lag Compensation 146
Registering hitboxes 147
Judging a shot with rewound queries 150
The rewind window and the view clamp 154
Channels and What Is Guaranteed 156
Messaging and Remote Calls 158
When a Message or Call Reaches Its Handler 159
Messages and Calls That Do Not Arrive 161
Handler Registration and Scope 163
Judging and Refusing a Relayed Call 165
Seeing Your Own Call: RpcSelfDelivery 167
Who May Send a Call, and How Many 170
Choosing Who Receives a Call 172
Sending a Message to Many Peers 174
State, a Message, or a Call? 176
System RPCs 178
Typed Messages 180
Channels and What Is Guaranteed 182
Adapting your engine 184
How physics replication works 186
Physics Synchronization 188
Locally predicted bodies 189
NetworkPhysicsComponent 191
PhysicsConvergence 194

Contents

Replicate a physics body 197
Swept projection targets 201
Trajectory projection 203
The content bundle protocol 205
Networked Scenes, Content Bundles and Persistence 208
Moving and carrying objects between scenes 209
Opening and closing scene instances 211
Placing clients in scenes 213
Registering a scene loader 215
Scene events and observers 216
Refusing a scene load 219
Registering a scene loader 221
How networked scenes work 222
Scene visibility: what a client in a scene can see 224
Scenes and bundles: things that look broken 226
Saving and loading a world 229
Choosing what a save keeps 231
Writing a world store 233
Reacting to changes: OnMembersChanged 237
Collection member reference 239
Sending your own data beside the members 244
Replicated State and Change Callbacks 247
What a member can hold, and how to keep one off the wire 248
NetworkMember 250
Networked collections 255
Retention, redundancy and recovery 259
How Often a Member Is Sent 263
How replicated state works 265
Letting clients write state 267
TransformComponent 269
Judging and arbitrating client writes 271
Your first networked component 273
Systems, Objects and Spawning 275
NetworkSystem reference 276
References between systems 280
Predicted spawn and despawn requests 282
Renting and finding systems in code 286
Who May Send a Call, and How Many 290
Spawn compensation 292

Contents

Binding spawns to your own engine 295
Starting and stopping systems 298
System groups 302
Parents, children and hierarchy 304
What a system is 306
Configuring a transport from launch arguments 308
Connecting across networks 310
Link states and connection events 312
Running a dedicated .NET server 315
Choosing a hosting topology 319
How transports work 322
Transports, Connectivity and Hosting 324
Packet transforms 325
The relay transport 327
Simulating latency, jitter and loss 331
The Synapse transport 334
The Transport contract 337
TransportManager 340
The Yak offline transport 342
Compression, Accuracy and Per-Type Registrations 344
Writing a Custom Serializer 346
Reaching Types You Cannot Annotate 350
Generator Errors and Warnings 352
How a Value Reaches the Wire 354
Serialization, Codegen and the Wire 356
Networked Type Attributes 357
The Shapes on the Wire 359
Wiring the Source Generator into a .NET Project 363
Type Identity on the Wire: Bundles and the Registry 365
What the Generator Emits 368
Writer and Reader 370
Nucleus Networking Documentation 376
API type reference 377
CodeBoost reference 380
Compatibility 382
Component Index 384
Configuration reference 387
Editor menus and authoring tools 392
FishNet to Nucleus name map 396

Contents

Reference & Project Meta 399
Installing and updating Nucleus in Unity 400
Performance and benchmarks 402
Project assets and loaders 404
Reporting a bug 408
Versioning, upgrading and release notes 410
Violation reference 411
Adding Nucleus to a .NET project 415
Choose your track: Unity or the core API 417
How replication works 419
Start Here 422
Installing the Unity integration 423
Nothing Is Replicating: A First-Run Checklist 425
Replicating your first object 427
Replicating your first system 430
Running two peers in one process 433
Terminology 437
Testing with two editors 440
What Nucleus is 442
Where to go next 444
Your first .NET host 446
Your first networked scene 451
Component Index 454
Control in a Unity script 457
Keeping a player's objects across a disconnect 459
Control, Client State and Prediction 461
Prediction settings on the Unity manager 462
Objects this peer does not control 464
Replaying a correction in Unity 466
What control means 472
Giving a client control of an object 474
When prediction is worth its cost 476
Sending client state from Unity 478
Host mode 483
Core, Managers and the Tick Loop 485
The manager map 486
The network loop steps 488
The Nucleus Core Manager component 491
Manager components reference 493

Contents

Play mode, domain reload and session lifetime 495
Reacting to session events in Unity 498
Starting and stopping a session in Unity 501
Tick rate and the loop in Unity 503
Common errors and what causes them 505
Developer tools in the editor 508
Diagnosing desync 512
Frequently asked questions 515
Filing a bug report 517
Handling violations in Unity 518
Diagnostics and Troubleshooting 521
Engine logging in the Unity console 522
Reading network statistics in Unity 524
Reading network statistics in Unity 528
Technical limitations 532
Violation reference 533
What the server trusts 537
Editor menus and authoring tools 539
Unity 543
What area of interest is 544
Making a host see what its players see 546
Interest Management and Scaling 548
An object is missing, or will not go away 549
Level of detail in Unity 552
Level of detail in Unity 554
Limit what a client receives by distance 556
Limit what a client receives by distance 558
Network Interest Object 560
Network Interest Object 562
Frozen or gone: what a stop does to the object 564
Frozen or gone: what a stop does to the object 566
Unity Interest Manager 568
What scales and what does not 570
Clipping shots at walls 573
Firing and judging a shot 575
Worked example: the HitscanRewind demo 580
How lag compensation works 582
Lag Compensation 584
Making an object hittable 585

Contents

UnityLagCompensationManager 588
Registering Call Handlers Across a Component's Lifetime 591
Channels and What Is Guaranteed 594
Messaging and Remote Calls 596
When a Message or Call Reaches Its Handler 597
Typed Messages in Unity 599
Messages and Calls That Do Not Arrive 601
Sending a Remote Call From a Component 603
State, a Message, or a Call? 605
NetworkNavMeshAgent 607
The network scene manifest and scene references 610
Driving a predicted body 613
How physics replication works 615
Physics Synchronization 617
Local physics scenes 618
NetworkNavMeshAgent 620
Physics execution modes 623
PhysicsSimulationDriver 625
Physics troubleshooting 627
ProjectedRigidbody 630
Replicate a rigidbody 633
NetworkTransform or ProjectedRigidbody 635
PhysicsSimulationDriver 637
Project assets and loaders 639
Addressables-delivered scenes 643
Content bundles in Unity 645
Networked Scenes, Content Bundles and Persistence 647
The network scene manifest and scene references 648
Networked scenes in Unity 651
How networked scenes work 653
Objects authored into a scene 655
Scenes and bundles: things that look broken 657
UnitySceneLoader component 660
UnitySceneManager component 662
Saving and loading a world from Unity 664
Replicated State and Change Callbacks 667
NetworkAnimator 668
NetworkTransform 672
Reacting to a replicated change in Unity 675

Contents

Replicating a value from a Unity script 677
How replicated state works 679
Unity Types and System.Numerics 681
Several systems on one object 683
Systems, Objects and Spawning 685
NetworkSystemObject component 686
NucleusBehaviour: the script base class 689
Player objects 693
Client-predicted spawning in Unity 696
Prefab identity and the prefab collection 699
Pooling spawned objects 701
Carrying and re-parenting objects 703
Spawning and despawning objects 705
What a system is 708
Blitz Relay Transport component 710
Connecting across networks 712
Building a headless Unity server 714
Choosing a hosting topology 716
How transports work 719
Transports, Connectivity and Hosting 721
Hosting and joining a relay session 722
Simulating a bad connection in the editor 724
Simulating a bad connection in the editor 727
Synapse Transport component 730
Synapse Transport component 732
Transport Manager component 734
Bundle Ids for Unity Assemblies 736
Code Generation in Unity 738
Generator Errors and Warnings 740
How a Value Reaches the Wire 742
Serialization, Codegen and the Wire 744
The Shapes on the Wire 745
Unity Types and System.Numerics 749

Client state

Using Unity? See Sending client state from Unity.

Client state is the per-tick values a simulation reads

Client state is what the controlling client sends the server every tick so the server can simulate the object: a move direction, a jump press, an aim angle. The server runs the same step the client does, from those values, and replicates the result. Sending client state rather than result state means the server only ever trusts what the player asked for, never where the player says it ended up.

Client state is ordinary `NetworkMember<T0>` fields constructed with `role: MemberRole.Predict`, declared in the system's own `NetworkComponent` beside the state the simulation computes from them. There is no separate client state class to write and nothing to register or attach: a member travels as client state because of its role.

Declaring client state

```
using System.Numerics;
using Nucleus.Components;
using Nucleus.Connections;
using Nucleus.Managers.NetworkLoop;
using Nucleus.Systems;

public partial class MoverComponent : NetworkComponent, IStateWriteValidator<Vector2>
{
    // Client state: written by the controller on its live tick, sent upstream every tick.
    public readonly NetworkMember<Vector2> MoveDirection = new(role: MemberRole.Predict);
    // A press that must not repeat when a missing tick is carried forward.
    public readonly NetworkMember<bool> JumpPressed = new(role: MemberRole.Predict, isCarr
iedForward: false);
    // State the server computes from the client state.
    public readonly NetworkMember<Vector3> Position = new();

    // Set by the game from the device on the controlling client.
    public Vector2 IntendedDirection;

    private const float Speed = 5f;

    // Server: once per Predict member per client tick taken. Write the member and decline
 to keep your own value.
    public StateWriteAction ValidateStateWrite(NetworkMember<Vector2> networkMember, Vecto
r2 proposedValue, Connection writingClientConnection)
    {
        if (proposedValue.LengthSquared() <= 1f)
            return StateWriteAction.Accept;

        MoveDirection.Value = Vector2.Normalize(proposedValue);

        return StateWriteAction.Decline;
    }

    protected override void OnEarlyFixedUpdate(StepDelta stepDelta, SimulationContext simu
lationContext)
    {
        // A client watching a system it does not control: nothing to simulate.
        if (simulationContext.Type is SimulationType.None)
            return;

        // The controlling client's live tick: this becomes the tick's client state.
        if (simulationContext.Type is SimulationType.Forward && NetworkSystem.IsController
(ControllerType.Client))
            MoveDirection.Value = IntendedDirection;

        // The server's step for a tick whose client state did not arrive: carry the last
one forward, leaving the press at its default.
        if (stepDelta.IsClientStateMissing)
            TryGetLastState(isEveryMemberCarried: false);

        if (!CoreManager.TransportManager.IsServerStarted)
            return;

        Position.Value += new Vector3(MoveDirection.Value.X, 0f, MoveDirection.Value.Y) *
(Speed / CoreManager.NetworkLoopManager.TickRate);
    }
```

```
}
```

`Position` here is ordinary state the server computes, so the controller sees its own movement a round trip late. To have the controller move at once and be corrected only when it is wrong, make the result `MemberRole.Validate` and let both peers run the step; see Predicted members and prediction history.

Who writes it, and when

Whoever controls the system writes its Predict members: the controlling client on its live tick (`SimulationType.Forward`), or the server for a system it controls. Write them inside the fixed update, from values the game gathered between ticks, so every pass of a tick reads the same numbers. A replay never writes them: it reads each past tick's value from the member's history.

A pure client sends its systems' Predict members to the server every tick, stamped with its tick. Each entry names, with one changed bit per member, only the members that differ from the newest of its ticks the server reported having decoded, so a value held still costs a bit. Both sides keep recent ticks as baselines, so a lost entry never breaks the chain; with no usable baseline every value is sent.

A host playing its own object writes the members in the same process, so nothing travels upstream. Each tick it still puts their values to the same validator a remote controller faces, and a declined value is reported to it as a correction.

Validating client state on the server

Implement `IStateWriteValidator<T0>` on the component, once per member type, and the server calls it once per Predict member for each client tick it takes:

```
StateWriteAction ValidateStateWrite(NetworkMember<T0> networkMember, T0 proposedValue, Con
nection writingClientConnection);
```

- `StateWriteAction.Accept` takes the client's value.
- `StateWriteAction.Decline` keeps what the member already holds.
- To clamp rather than refuse, write the member inside the call and decline: the server keeps your value, simulates with it and forwards it.
- A validator that throws declines.

The same interface judges a client's writes to ordinary state members; see Validating client writes. Several members of one type share one implementation, so compare the `networkMember` argument against your fields when they need different rules.

What the server does with each tick

The server takes one of the controller's ticks per server tick, released through the State Interpolation like any other state, so a burst that arrives together is spread over the ticks that follow. For each tick it takes it runs the validator, then steps the system with its Predict members holding that tick's values.

A tick it cannot take in time is dropped, and one that arrives after its tick was simulated is late. Both are counted on the controller's `Connection`:

```
public uint ClientStateDroppedCount { get; }
public uint ClientStateLateCount { get; }
public uint ClientStateReanchorCount { get; }
```

A client that stalls and resumes is re-anchored: when its client state arrives further behind the ticks the server already simulated for it than the State Interpolation, the server re-bases onto the client's ticks, sends a reconcile for each system the client controls, and counts it in `ClientStateReanchorCount`.

The server's step is stamped with the server's own tick in `StepDelta.Tick`. The controller's tick it simulates is in `SimulationContext.RemoteTick`, for pairing and logging only: a client chooses its own tick numbers, so never time a cooldown or anything else by it. See The simulation context.

Ticks whose client state did not arrive

A system a remote client drives is simulated on every server tick whether or not that tick's client state arrived: before the first one, and while the client's packets are lost. That is what lets the server act on the object in its own world, a collision for instance. On such a step `StepDelta.IsClientStateMissing` is true, `SimulationContext.IsRemoteTickEstimated` is true, and every Predict member reads its default. To carry the previous tick's client state forward instead, call the component's:

```
public bool TryGetLastState(bool isEveryMemberCarried)
```

The Predict members then read their values as they stood at the end of the previous tick, for the rest of the step: the client's own when that tick's arrived, and whatever the server's step left in the members otherwise, so a value the server writes into a member is what the next tick carries. Pass `false` to keep the members constructed with `isCarriedForward: false` at their defaults, so a jump press never fires twice; or send a press count, which repeats nothing when carried, and pass `true`. It returns `false`, leaving the defaults, before any client state has arrived. Nothing is carried unless you call it, so a game that does not carry is reconciled back to what the server simulated.

To know which tick you are carrying, use the overload that also reports the simulation context of that client state:

```
public bool TryGetLastState(bool isEveryMemberCarried, out SimulationContext lastStateSimu
lationContext)
```

On a missing step, `lastStateSimulationContext` is the context of the newest step this component's client state arrived for. On the server its `RemoteTick` is the controller's tick that state was written on and `IsRemoteTickEstimated` is false, so the missing step's own `RemoteTick` minus it is how many ticks you are carrying it over. That is useful for giving up on a carry that has gone on too long:

```
protected override void OnEarlyFixedUpdate(StepDelta stepDelta, SimulationContext simulati
onContext)
{
    if (stepDelta.IsClientStateMissing && TryGetLastState(isEveryMemberCarried: false, out
 SimulationContext lastStateSimulationContext))
    {
        // Stop walking once the client has been silent for more than five of its ticks.
        if (simulationContext.RemoteTick - lastStateSimulationContext.RemoteTick > 5)
            MoveDirection.Value = Vector2.Zero;
    }

    // Simulate from MoveDirection as usual.
}
```

On a step that is not missing it is the context of the step running now. Before any client state has arrived, or on a component with no Predict member, it is the default, whose `RemoteTick` is unset. As with the step's own context, compare these ticks but never time anything on the server by them.

A tick runs missing only once the State Interpolation says it should have arrived. Just after a stall drained the buffer, or with a client clock running slower than the server's, the next client tick runs when it arrives instead, so those are the server ticks that pass the object by.

Forwarded to every observer

The server forwards the Predict values it kept, after validation and its own writes, to every observer of the system, the controller included. Observers can drive presentation (footsteps, animation) from what the controlling player is doing, and an observer that predicts the system simulates from them (see Predicting another client's object).

On the controller a server value for a past tick lands in that tick's history when it differs from its own, so a later replay of that tick reads what the server simulated. A value the server writes from its own step reaches the controller the same way.

Reporting changes

`OnMembersChanged` reports Predict members as it reports every member, on change rather than every tick:

- `MemberChangeDirection.Read` on the server when a client's value that differs is taken (`TryGetWritingClient` names the sender), and on an observer receiving a changed value.
- `MemberChangeDirection.Correction` on the controller when the server's value for one of its past ticks differs from its own. The server's value is already in the member's history at that tick when the callback runs.

The simulation reads the members every tick in its fixed update anyway; use the callback for presentation, or to react to a correction.

Controller-only, regardless of state write access

Client state is controller-only no matter what a system's state write access allows. Client state from any connection other than the controller is refused: its bits are still consumed off the wire so stream alignment is not broken, but an `UncontrolledStateChangeViolation` is raised instead of the client state being applied.

Controller retention

> **Using Unity?** See Keeping a player's objects across a disconnect.

Connection ids are recycled. An object left naming a departed controller is one the next renter of that id inherits, along with its state write permission and its client state routing, because both are decided by the same comparison against `ControllerConnectionId`. Control is therefore always surrendered the moment a connection leaves - there is no configuration that keeps an object controlled by somebody who is gone. `ControllerRetentionPolicy` only decides whether a token is issued so the objects can be handed back to whoever comes next.

A policy either releases a departing controller or retains it

`ControllerRetentionPolicy` has two values:

- `Release` (0) - the object is simply uncontrolled once its controller leaves, and nothing is remembered about who held it. This is the default: it costs nothing and cannot leak, because an object nobody is coming back for should not occupy a retention record.
- `Retain` (1) - the object is uncontrolled, and it joins the retention record the departing connection is issued a token for, so a returning player can be handed it back.

`SystemManager.ResolveControllerRetentionPolicy(NetworkSystem)` resolves the policy for one system, most specific first:

`NetworkSystem.ControllerRetentionPolicyOverride`, if set.

The system's group's override, if set.

`SystemManager.DefaultControllerRetentionPolicy` (`Release` by default).

A system and its group both leave their override unset by default, so an unconfigured world resolves to `DefaultControllerRetentionPolicy` on one comparison each.

The token

`ControllerRetentionToken` is a readonly struct with `Id` and `Generation`, plus `IsValid` (true when `Id` is not the unset value). It is opaque: nothing in it addresses anything on the wire, it is meaningful only to the peer that issued it, and it is safe to store beside an account record. It is a token rather than the departed connection's id on purpose - ids are rented from a pool and handed out again, so a stored id would eventually name somebody else's retention. `Generation` is what makes a stale token fail rather than alias: a record's identifier reissued later is a different token, and only the current one resolves.

When a connection is first given control of something retained, the server reserves a token for it and delivers it once, reliably, over `ControllerRetentionTokenNotice`. On the client, `SystemManager.LocalControllerRetentionToken` holds the token reserved for this session, and `SystemManager.LocalControllerRetentionTokenReceived` fires when it arrives. That is the moment to persist it somewhere that outlives the session, because the connection carrying it is the one whose loss the token exists to survive.

A returning connection reclaims its objects with its token

`SystemManager.TryReclaimController(Connection connection, ControllerRetentionToken controllerRetentionToken)` hands a returning connection every object a retention record holds, and ends the record. It refuses only a token that does not resolve - a stale one, whose record expired, was evicted, or was already redeemed, returns `false` rather than a wrong

answer. It does not check whether the returning peer is the same player; that question is the game's to answer before calling it.

Three events report what happens to a record over its lifetime:

- `ControllerRetained` - a record opened for a departing connection, raised only when at least one object was actually retained.
- `ControllerRetentionEnded` - a record stopped being redeemable, carrying the token and a `ControllerRetentionEndReason`.
- `ControllerRetentionAdopted` - a record standing after this peer adopted a dead host's world. It only means anything under host migration, which is planned but not yet built.

Only a reclaim returns the objects; every other ending leaves them uncontrolled

`ControllerRetentionEndReason` explains why a record stopped being redeemable:

- `Reclaimed` (0) - a returning connection redeemed the token and was handed the objects.
- `Expired` (1) - the record outlived `SystemManager.ControllerRetentionSeconds` without being redeemed.
- `Evicted` (2) - the record was the oldest standing when a new one needed room under `SystemManager.MaximumRetainedControllerRecords`.
- `Emptied` (3) - every object the record held left the world or was handed to another controller, so there was nothing left to redeem.

Every reason but `Reclaimed` leaves the objects uncontrolled and nothing relating them to one another. A handler that wants to act on a record's objects as a set - despawning a player's belongings when they don't return, for example - has to do it from `ControllerRetentionEnded`, before that reason takes effect.

A record is bounded by both a timeout and a record count

`SystemManager.ControllerRetentionSeconds` (default `DefaultControllerRetentionSeconds`, 120) is how long a record stays redeemable before it expires. Setting it to `SystemManager.UnlimitedControllerRetentionSeconds` (0) disables expiry entirely, leaving `MaximumRetainedControllerRecords` as the only bound.

`SystemManager.MaximumRetainedControllerRecords` (default `DefaultMaximumRetainedControllerRecords`, 64) is how many records may stand at once. Past that, the oldest standing record is evicted to make room for a new one - the most recent departure, which is the one most likely to come back, is kept redeemable.

Proving that a returning peer is the same player is the game's job

The framework never decides that a returning peer is the same player who left. That is why a token is issued rather than an identity matched: `TryReclaimController` trusts whatever token it is handed and only refuses one that fails to resolve. Getting the token from a departed player back to the server is the game's business, not the engine's - present it through whatever channel the game's own authentication already trusts, and store it against a server-determined identity rather than in place of one. A server that accepts a token straight off the wire with no identity check behind it is trusting whatever a client hands it.

The Unity page covers the inspector fields for retention

Inspector fields for controller retention, and `NetworkPlayerDisconnectMode`, are covered on the Unity

page: Keeping a player's objects across a disconnect.

Host migration

Controller retention is what lets a peer promoted after a host dies rebuild retained records from the objects it adopted, rather than losing them. Retention itself does not migrate a host - see Surviving the loss of a host for that.

Reading who controls a system

Using Unity? See Control in a Unity script.

Checking control

`NetworkSystem.IsController(ControllerType)` answers whether the local peer controls a system:

```
if (networkSystem.IsController(ControllerType.AnyController))
{
    // The local peer is the server (with no controlling client) or the controlling client
.
}
```

`ControllerType` is a flags enum:

Value	Meaning
`None`	No controller.
`Server`	The local peer is the server and no client controls the system.
`Client`	The local peer is the client that controls the system.
`AnyController`	`Server \

Combine flags to test either case at once, as `AnyController` does.

The controller fields

`NetworkSystem` exposes the controlling identity directly:

- `ControllingClient` — the `Connection` of the controlling client, or `null` when the server controls the system.
- `ControllerConnectionId` — the connection Id of the controlling client, or `NetworkSystem.UnsetId` when the server controls it. This is the canonical identity that replicates to clients.

`ControllerConnectionId` is the value to compare against a known Id. It is not itself a peer: an Id names a connection, but a `Connection` instance is a live reference that can be missing, reassigned by the pool, or simply not yet resolved on a client (see below). Prefer `IsController` or `ControllingClient` when the question is "does the local peer control this", and reach for `ControllerConnectionId` only when comparing against an externally-known Id.

`ControllingClient` can briefly be `null` on a client even while `ControllerConnectionId` names someone: control replicates on the unreliable state stream while the peer roster arrives reliably, so the two can invert for a moment. The reference fills in once the peer is reported.

Reacting to control changes

```
networkSystem.ControllerChanged += OnControllerChanged;

void OnControllerChanged(Connection previousControllerConnection, Connection currentContro
llerConnection)
{
    // ...
}
```

`ControllerChangedHandler` is `void ControllerChangedHandler(Connection previousControllerConnection, Connection currentControllerConnection)`. Either Connection is `null` when control sits with the server, or with a peer client this peer holds no `Connection` for.

Finding what a peer controls

`InterestManager.TryGetControlledSystems` answers the reverse question — what does this connection drive:

```
if (coreManager.InterestManager.TryGetControlledSystems(connection, out IReadOnlyCollectio
n<NetworkSystem> controlledSystems))
{
    foreach (NetworkSystem controlledSystem in controlledSystems)
    {
        // ...
    }
}
```

An identifier overload, `TryGetControlledSystems(uint connectionId, out IReadOnlyCollection<NetworkSystem> controlledSystems)`, exists for a caller that only has an Id rather than the `Connection` itself.

This is not a player-object lookup. A connection controlling a character, a vehicle, and a turret is returned all three, in no particular order — the core has no concept of a player, and no opinion about which controlled system is "the" player. That distinction belongs to the game. The returned collection is a live view, not a copy, so copy it before doing anything that can reassign a controller while walking it.

Walking it in code that runs often

The view is an interface, so a `foreach` over it allocates an enumerator on every walk. That is fine for a one-off lookup, but not for code that runs every tick or for every pair, such as an interest condition. There, fill a list you keep instead:

```
private readonly List<NetworkSystem> _controlledSystems = new();

private void CheckControlledSystems(CoreManager coreManager, Connection connection)
{
    int count = coreManager.InterestManager.GetControlledSystems(connection, _controlledSy
stems);

    for (int i = 0; i < count; i++)
    {
        NetworkSystem controlledSystem = _controlledSystems[i];
        // ...
    }
}
```

`GetControlledSystems` clears the list, fills it with every system the connection controls, and returns the count; it allocates nothing once the list has grown to fit. Because it is a copy, control may be reassigned

while you walk it.

Control is exclusive

A system is controlled by the server or by exactly one client, never both. `ControllerConnectionId` is either `UnsetId` (server) or a single client's Id at any moment.

Comparing controllers: use the Id, not the reference

A `Connection` is pooled and reused once its link ends, so a reference kept from an earlier lookup can name a different peer later, while `ControllerConnectionId` is an Id to begin with.

Never compare `Connection` references to decide whether a peer controls a system. Compare `Connection.Id` values instead:

```
bool controls = networkSystem.ControllerConnectionId == someConnection.Id;
```

`IsController` and `TryGetControlledSystems` already do this correctly internally; write the same comparison by hand whenever a raw Id or Connection comparison is unavoidable.

Control, Client State and Prediction

Pages in this category:

- Client state
- Reading who controls a system
- Predicted members and prediction history
- Reconciliation and replay
- What control means
- Assigning and releasing control
- Controller retention
- When prediction is worth its cost

Predicted members and prediction history

Using Unity? See Prediction settings on the Unity manager.

A member's role says what it is for

`NetworkMember<T0>` takes a `role` constructor argument, `MemberRole.State` by default, readable afterwards as `NetworkMember<T0>.Role`:

```
public readonly NetworkMember<System.Numerics.Vector2> MoveDirection = new(role: MemberRol
e.Predict);
public readonly NetworkMember<System.Numerics.Vector3> Position = new(role: MemberRole.Val
idate, accuracy: 0.01f);
public readonly NetworkMember<ushort> Health = new();
```

Role	What it is	Who writes it	What the controller sends upstream
`State`	Ordinary state. Inbound values apply as they arrive.	The server, or a client its write access allows.	The value, when it changes and write access allows it
`Predict`	Client state: a per-tick value the simulation reads.	Whoever controls the system, on its live tick.	The value, every tick (see [Client state](./client-state-components.md))
`Validate`	A predicted result, computed from the client state.	Every peer that simulates the system.	Never the value: a hash of the tick's results, with its client state

Roles are per member. Members of every role can share a component, and the usual predicted object has all three: `Predict` members for what the player asked, `Validate` members for what the step computes from them, and `State` members for what only the server decides.

What a predicted role does depends on the peer:

- **The pure-client controller** predicts the system: it writes its `Predict` members on its live tick, runs the step at once to compute its `Validate` members, and never sends those upstream. The server checks the result and orders a correction when it differs.
- **The server** takes the controller's `Predict` members, simulates the same tick, and checks the controller's `Validate` result against its own.
- **Any other client** applies the server's values as they arrive, but a member with a role other than `State` takes part in a reconcile: it is held at the server's start-tick values while the controlled systems replay. A client that sets `NetworkSystem.IsObserverPredictionEnabled` predicts the system as its controller does (see Predicting another client's object).
- **A host** applies values as they arrive. It is the server, so nothing it predicts is ever corrected.

The server checks the result

A `Validate` member puts the check on the server, which holds the client state and everything else the step reads:

After its live fixed updates, the controller hashes its system's `Validate` members, each member's absolute wire encoding at its `Accuracy` in component and member order, and appends the hash, with

the sequence of the last reconcile it applied, to that tick's client state.

The server simulates the same tick of client state, hashes its own `Validate` members the same way and compares. On a difference it counts `Connection.ResultMismatchCount`, orders a reconcile (counted in `ReconcileSentCount`) and raises `NetworkSystem.ResultMismatched`; see Validate results the server found different.

Results the controller computed before the correction arrived carry the old sequence, so the server passes them over instead of ordering a reconcile for each. If the controller has not echoed the new sequence within a round trip plus `StateInterpolation`, the server sends the reconcile again (`ReconcileResentCount`), so a lost reconcile costs one round trip, not a stuck client. A resend that reaches a controller which already applied the first lands as an ordinary full rather than ordering a second replay.

The controller never judges its own prediction. Every server value for a predicted member that arrives, a full that primes or recovers it included, leaves the prediction standing; only a reconcile the server orders corrects it. That is why the check works for results that depend on state the client cannot see in time: a jump the server refuses because of health only the server drains is still caught, because the server holds both the client state and the health.

A hash compares exactly, so a `Validate` member is quantized to its `Accuracy` on every write, on both peers, and each tick starts from the numbers the wire carries. Choose an accuracy the simulation can tolerate rounding to. A `Validate` member may use any `SendInterval`, since the server compares against its own current values every tick and an ordered reconcile is a full.

A predicted system needs `Predict` members: the result check rides each tick of client state, and the reconcile pairs with the server's echo of that tick. A system with `Validate` members and no client state sends nothing for the server to check, and the server's values simply overwrite the prediction.

The result check runs after the client state is validated. A tick the server's `IStateWriteValidator<T0>` corrected therefore produces a different result, and the controller is reconciled onto the server's outcome.

An observer that predicts compares for itself

An observer predicting a system another client controls (`IsObserverPredictionEnabled`) has no result check of its own on the server, which checks only the controller. It compares the server's values for that system's `Validate` members itself, at the tick its own echo names, within each member's `Accuracy`:

- **A match** is discarded. The prediction stands and the server's value is not applied.
- **A mismatch** is recorded with the tick and a `MismatchCause`, and the system keeps predicting until the `Reconcile` step, where the engine restores and replays (see Reconciliation and replay).
- **A history too shallow** to reach the echoed tick is recorded as `MismatchCause.HistoryMissing`, and the system logs a warning once naming `SystemManager.PredictionHistoryTicks`.
- **A full the server did not order** (a prime or a loss recovery) is compared like any other value, and is recorded as `MismatchCause.Resync` only when a value differs.

The server sends a member only when it changes, so at the end of each packet that echoes a new tick, every `Validate` member on `SendInterval.Normal` without projection that the packet did not carry is compared against the server value the observer already holds. A paced or projected member's silence can also mean its send was not due, so it is checked only when a value arrives.

Prediction history depth

`SystemManager.PredictionHistoryTicks` sets how many ticks of history `Predict` and `Validate`

members keep, so a reconcile can restore any member to a past tick and a replay can read each past tick's client state. It must cover the round trip plus `SystemManager.StateInterpolation`, with headroom. At 60 ticks per second a 250 ms round trip is 15 ticks; with an interpolation of 2, use 24 or more.

Left at `SystemManager.UnsetPredictionHistoryTicks` (0), the default, every client gives the predicted members of every system it holds, controlled or not, `SystemManager.DefaultPredictionHistoryTicks` (64 ticks, about a second at 60 ticks per second). The server, which never rolls back, keeps the members' own two ticks. When a reconcile starts at a tick older than the history holds, the server's value still lands there and the system replays from it, and a warning naming the shortfall is logged once per system.

A member takes the history when it initializes, and a system grows its members to it when this client gains control, so set the value before systems spawn. A member served on an interval widens its history further on its own, independent of `PredictionHistoryTicks`, so a replay can still address the ticks between its serves.

Reading ring history

```
public bool TryGetValueByTick(uint tick, out T0 value)
public bool TryGetValueAtOrBeforeTick(uint tick, out T0 value)
public bool TryGetPreviousValueAndTickGap(out T0 previousValue, out uint tickGap)
public bool TryGetMismatchValues(out T0 predictedValue, out T0 serverValue)
public bool TryGetPreRewindValue(out T0 value)
```

- `TryGetValueByTick` returns the value written on exactly that tick, and false when the member was not written on it or its history no longer holds it. Use it to read a `Predict` member's value for a past tick.
- `TryGetValueAtOrBeforeTick` returns the value the member held as of that tick, the last write at or before it, and is false only when every tick it holds is newer. This is the lookup the restore uses, so a predicted member need not be written every tick.
- `TryGetPreviousValueAndTickGap` returns the value in the slot before the live one with the tick span between the two, for code that derives a rate of change (a velocity from position deltas, for example). It returns false when the live slot is not strictly newer than the previous one.
- `TryGetMismatchValues` returns this client's value and the server's at `ReplayReport.StartTick`, when the latest reconcile found this member wrong, from the restore until the next tick advance.
- `TryGetPreRewindValue` returns the value the member held before the latest reconcile restored it, the value that was on screen, until the next tick advance.

PredictedStateChangeViolation

A controller must not write a `Validate` member's state upstream: its client state carries what the server needs, and the server simulates the outcome. If a client sends state for a member it predicts, `NetworkSystem` raises a `PredictedStateChangeViolation`:

```
public struct PredictedStateChangeViolation : IViolation
{
    public uint SystemId;
}
```

It carries the `SystemId` of the `NetworkSystem` whose predicted member the sender attempted to change, and is raised with `ViolationAction.Ignore`. The incoming bits are consumed either way, by the caller's decode, so the wire stream stays aligned regardless of the verdict; only the ring is left untouched. Subscribe via `ViolationManager.PredictedStateChangeViolationDetected` to observe these without changing the outcome.

Opting in

Prediction is an explicit, per-member opt-in you add to your own `NetworkMember<T0>` field declarations; no base class turns it on for you. The Unity demo `Demos/Prediction/ValidatedJump` declares its walk direction and jump press count with `MemberRole.Predict`, every movement result of `ValidatedJumpMovementComponent` with `MemberRole.Validate`, and simulates them in its `OnEarlyFixedUpdate`.

See Reconciliation and replay for what happens once the server orders a correction: the restore, the replay the engine runs, and the report of what changed.

Reconciliation and replay

> **Using Unity?** See Replaying a correction in Unity.

A predicted member can diverge from the server: the server simulates a tick of the controller's client state, finds the controller's result different from its own, and orders a correction. The server also orders one itself when it re-anchors a stalled client, and when game code calls `SendReconcile`. Reconciliation is what the engine does next: it restores the client's world to the server's values as of one tick, replays every tick since then through the same fixed updates that produced them, and reports what was wrong and what changed. A game writes the simulation once; it does not write the replay.

One reconcile, in order

Detect, on `LateStateUpdate`. While the inbound packets apply, a reconcile the server ordered is recorded against the echoed client state tick as `MismatchCause.ServerOrdered`, instead of rewinding on the spot. An observer predicting a system records its own compare the same way. At the end of the packet that recorded the earliest tick, every predicted member of every system captures the value the server last sent as the restore base, so remote calls and `LateStateUpdate` callbacks that run before the `Reconcile` step cannot move it.

Restore, at the head of the `Reconcile` step, ahead of its callbacks. One start tick is chosen for the whole world: the earliest tick any member recorded. Every predicted member of every system this client predicts is set to its captured server value, its later history is cleared, and its value before the restore is kept as its pre-rewind value. A member the server has sent nothing for yet keeps its own history, rewound to the same tick. Systems this client does not control are held at their captured values. `ReplayReport` is filled with the members whose restored value differs from what this client held for the start tick.

`ReconcileRequired` is raised with the start tick on each restored system, in system Id order.

Replay. `ReplayStarting` is raised on every system the replay covers. Then, for each tick from the start tick plus one to the live tick minus one, every restored system runs its early fixed update and then its late one as `SimulationType.Replayed`, in system Id order.

Report the effect. Held systems return to their latest values. Each predicted member whose present the replay moved away from its pre-rewind value is added to `ReplayReport.Corrections` and raises `OnMembersChanged` with `MemberChangeDirection.Correction`. Then `ReplayCompleted` is raised on every system that raised `ReplayStarting`.

Continue, on `EarlyFixedUpdate`. The live tick runs the fixed updates as `SimulationType.Forward` from the corrected state.

A reconcile covers the whole world, once per inbound pass. Two members of one system, or two systems this client controls, that diverge in the same pass are restored to the same tick and replayed together, so one divergence costs one replay.

The fixed updates the engine replays

The engine replays the fixed updates of a system something simulates: a component that overrides one of them, or a callback registered with `NetworkSystem.RegisterSimulationCallback`:

```
// On NetworkComponent:
protected virtual void OnEarlyFixedUpdate(StepDelta stepDelta, SimulationContext simulatio
nContext) { }
protected virtual void OnLateFixedUpdate(StepDelta stepDelta, SimulationContext simulation
Context) { }

// Or registered per system with NetworkSystem.RegisterSimulationCallback:
public interface INetworkSimulationCallback
{
    void OnSimulationStep(NetworkLoopSteps networkLoopStep, StepDelta stepDelta, Simulatio
nContext simulationContext);
}
```

Within a system, components run first in component order, then the registered callbacks in registration order. `UnregisterSimulationCallback` removes a callback. A raw `INetworkLoopStepCallback` keeps running once per loop tick and is never replayed.

Peer	System	How the fixed updates run
Server or host	Driven by a remote client through client state	`Forward`, once per server tick on one of the client's ticks, after its client state is validated, or as a step missing its client state when that tick's did not arrive
Server or host	Controlled by the server, by the host's own client half, or by a client that sends no client state	`Forward`, once per tick
Controlling client	Its own system	`Forward` once per live tick, and `Replayed` once per tick a reconcile replays
Observing client with `IsObserverPredictionEnabled`	Another client's system	As the controlling client
Any other client	A system it does not control	`None`, once per loop tick; nothing is simulated

`NetworkSystem.IsSimulatedPerClientState` is true on the server for a system run on its client's ticks. A replayed step, and the server's step for a tick of a client's client state, is not a real frame, so its `Delta`, `FixedDelta` and `NormalizedDelta` are one tick interval and `TimeSinceLastFixedUpdate` is 0.

The simulation context

Each fixed update receives a `SimulationContext`, a read-only struct describing the step:

Field	Meaning
`Type`	The pass the step belongs to: `SimulationType.Forward` the first time a tick is simulated, `Replayed` for a past tick simulated again during a reconcile, `None` on a peer that does not simulate the system.

`RemoteTick`	The opposite end's tick the step pairs with. On the server, for a system a client drives, the controller's tick whose client state is being simulated. On a client, the server tick of the state it applied. `NetworkLoopManager.UnsetTick` when there is none: on the server for a system no client drives, and on a client before any state has arrived.
`IsRemoteTickEstimated`	True when `RemoteTick` was inferred rather than arriving with the data: on the server for a client tick whose client state did not arrive, where it is the tick the server assumed, and on a client for a tick no state was applied on, every replayed tick included, where it is the newest server tick applied carried on one tick per local tick.

Time everything by `StepDelta.Tick`, the local tick being simulated. On the server that is the server's own tick even for a step that runs one of the client's ticks. `RemoteTick` is the other end's numbering, and on the server it is a number the client chose: use it to pair what the server did with what the client logged, never for a cooldown, a timeout or any other timing.

```
protected override void OnEarlyFixedUpdate(StepDelta stepDelta, SimulationContext simulati
onContext)
{
    if (simulationContext.Type is not SimulationType.Forward || !CoreManager.TransportMana
ger.IsServerStarted)
        return;

    // Pair the server's step with the client's tick in a log line; time anything by stepD
elta.Tick.
    if (stepDelta.IsClientStateMissing)
        System.Console.WriteLine($"Server tick {stepDelta.Tick} ran client tick {simulatio
nContext.RemoteTick} without its client state.");
}
```

Writing the simulation

```
using System.Numerics;
using Nucleus.Components;
using Nucleus.Managers.NetworkLoop;
using Nucleus.Systems;

public partial class MoverComponent : NetworkComponent
{
    // Client state: what the player asked for this tick.
    public readonly NetworkMember<Vector2> Direction = new(role: MemberRole.Predict);
    // The result the controller predicts and the server checks.
    public readonly NetworkMember<Vector3> Position = new(role: MemberRole.Validate, accur
acy: 0.01f, correctionSmoothingTicks: 6);

    // Set by the game from the device on the controlling client.
    public Vector2 IntendedDirection;

    private const float Speed = 5f;

    protected override void OnEarlyFixedUpdate(StepDelta stepDelta, SimulationContext simu
lationContext)
    {
        // A client watching a system it does not control: nothing to simulate.
        if (simulationContext.Type is SimulationType.None)
            return;

        // The controlling peer's live tick: this becomes the tick's client state. A repla
y reads the recorded client state instead.
        if (simulationContext.Type is SimulationType.Forward && NetworkSystem.IsController
(ControllerType.AnyController))
            Direction.Value = IntendedDirection;

        // The server's step for a tick whose client state never arrived: carry the last t
ick's instead of the defaults.
        if (stepDelta.IsClientStateMissing)
            TryGetLastState(isEveryMemberCarried: true);

        // The live tick, the server's run of each client tick, and every replayed tick al
l compute here.
        Position.Value += new Vector3(Direction.Value.X, 0f, Direction.Value.Y) * (Speed /
 CoreManager.NetworkLoopManager.TickRate);
    }
}
```

The controller writes the client state for the live tick before the step reads it, on the live tick only; every other pass reads the `Predict` member, which a replay points at the tick it replays. Step by the fixed tick, never by frame time, so every pass of a tick computes the same value.

What a replayed tick sees

- `Validate` members write the replayed tick's slot, and each replayed tick becomes the member's newest, so `Value` reads the last tick replayed and the next live write lands after it.
- `Predict` members read the client state recorded for the replayed tick.
- `State` members keep their live value; writes to them land on the live value.
- Systems this client does not control read the server values that arrived with the start tick, unless the hold radius leaves them out.

- No physics step runs between the early and late fixed update, so movement that must replay cannot wait on one.

`NetworkLoopManager.SimulationType` is the pass running now (`None` outside a simulate step), mirrored as `NetworkSystem.SimulationType`. `NetworkLoopManager.ReplayTick` is the replayed tick during a replay and `NetworkLoopManager.UnsetTick` otherwise, and `SimulationTick` is the replayed tick during a replay and `Tick`, which stays the live tick throughout, otherwise. `NetworkSystem.IsPredicting` is true on the clients that replay a system: its controller, or an observer predicting it, while it has a predicted member.

Calls refused during a replay

A replay re-runs every tick since the start tick, so anything with an effect outside the simulation would happen again. While a replayed tick runs, the engine refuses these calls, logging once per kind of call:

- `TrySendRpc` returns false, and `Connection.SendMessage<T0>` sends nothing.
- `EnsureStartPredictedSystem` and `EnsureStopPredictedSystem` refuse with `PredictedSpawnRefusal.Replaying`.
- Writes to collection members and reference members change nothing.
- `NetworkAnimator.SetTrigger`, `ResetTrigger`, `CrossFade` and `Play` do nothing.

Play one-shot effects (sounds, particles, an RPC to the server) on `SimulationType.Forward` only.

What the reconcile reports

```
public event ReplayStartingHandler? ReplayStarting;
public event ReplayCompletedHandler? ReplayCompleted;
public delegate void ReplayStartingHandler(ReplayReport replayReport);
public delegate void ReplayCompletedHandler(ReplayReport replayReport);

public ReplayReport ReplayReport { get; }
```

`NetworkSystem.ReplayReport` describes the latest reconcile on the system. It is one instance per system, reused by every reconcile, so read it while the reconcile it describes is current:

Member	Meaning
`StartTick`	The tick the world was restored to. The replay re-runs the ticks after it.
`EndTick`	The last tick the replay re-runs: the live tick less one.
`IsReplaySkipped`	True when the replay would have run longer than `SystemManager.MaximumReplayTicks`, so the server's values stand at `StartTick` without one.
`Mismatches`	The cause: each `PredictionMismatch` (`Member`, `MemberName`, `Cause`) whose value at `StartTick` differed from this client's. Empty on a system restored only because another one diverged.

`Corrections`	The effect: each `MemberCorrection` (`Member`, `MemberName`) whose present the replay moved. Filled before `ReplayCompleted`.

`MismatchCause` is `ValueDiffered` (the server's value differed), `HistoryMissing` (the history no longer reached the tick), `Resync` (a prime or loss-recovery full whose values differed) or `ServerOrdered` (a reconcile the server sent). A held system's report names the start tick and lists no mismatches.

Inside a handler, `NetworkMember<T0>.TryGetMismatchValues(out predictedValue, out serverValue)` gives both values at the start tick and `TryGetPreRewindValue(out value)` gives the value that was on screen, both until the next tick advance. A correction reaches components through `OnMembersChanged`, so a replay that took back a jump can cancel its effects:

```
public override void OnMembersChanged(ulong memberFlags, MemberChangeDirection memberChang
eDirection)
{
    if (memberChangeDirection != MemberChangeDirection.Correction)
        return;

    if ((memberFlags & (ulong)MovementComponentFlags.JumpCount) != 0 && JumpCount.TryGetPr
eRewindValue(out ushort shownJumpCount) && JumpCount.Value < shownJumpCount)
        CancelJumpEffects();
}
```

On the controlling client each system also counts `ReconcileCount`, `ReplayedTickCount` and `LastReplayTickCount` since it was rented, and `GetMismatchCount(MismatchCause)` per cause. All of them stay at zero while predictions agree.

Smoothing a correction

Construct a member with `correctionSmoothingTicks` to have its `InterpolatedValue` glide from the pre-rewind value to the corrected one over that many presented ticks, while `Value` is correct at once:

```
public readonly NetworkMember<Vector3> Position = new(role: MemberRole.Validate, accuracy:
 0.01f, correctionSmoothingTicks: 6);
```

0, the default, shows the correction at once.

Limiting how long a replay runs

```
coreManager.SystemManager.MaximumReplayTicks = 30;
```

A reconcile whose replay would run longer lands the server's values at the start tick without replaying, and its report sets `IsReplaySkipped`. `SystemManager.UnsetMaximumReplayTicks` (0), the default, sets no limit.

Systems this client does not control

A system controlled by the server or by another client takes its values from the server, which are the truth, and is never compared or simulated on this client. It still takes part in a reconcile: the client only knows it up to the start tick, so from the restore its predicted members read the server values that arrived with the start tick, and the controlled systems replay against it there. It raises `ReplayStarting` and `ReplayCompleted` with the replayed systems, so whatever shows it can be placed where the replay sees it and put back. Before `ReplayCompleted` it returns to its latest values, so a replay leaves no net change on it.

Holding only the systems near a replay

Systems the client does not control are held at the start tick while the ones it controls replay. In a crowded scene, set a radius so only those near a restored system are held:

```
coreManager.SystemManager.ReplayHoldRadius = 20f;
```

Distances are read through the interest position reader (`InterestManager.SetPositionReader`). A system whose position cannot be read is always held, and the rest keep their latest values and raise no replay events. `SystemManager.UnsetReplayHoldRadius`, the default, holds every system, and so does a build without distance interest.

Predicting another client's object

A client can predict a system another client controls instead of holding it, opted in per system on that client:

```
observedSystem.IsObserverPredictionEnabled = true;
```

Its fixed updates then run as `SimulationType.Forward` on every live tick, from the client state the server forwards, which lands on the local tick its packet's echo names. On a live tick that client state has not arrived yet, so the step says `StepDelta.IsClientStateMissing` and the game carries the last client state forward with `TryGetLastState`, exactly as the server does. The observer compares the server's values for the `Validate` members with its prediction at its own echo, since the server checks only the controller's result, and a reconcile restores and replays the system with the ones the observer controls. Nothing new travels. The observer needs an echo of its own, so it must control a system that sends client state: one with `MemberRole.Predict` members. An observer controlling nothing of the kind gets no echo, and the server's values land on it directly.

Replaying from ReconcileRequired yourself

`ReconcileRequired` stays as the low-level hook, for a game that drives its own replay:

```
public delegate void ReconcileRequiredHandler(uint rewoundTick);

public event ReconcileRequiredHandler? ReconcileRequired;
```

It is raised at the head of the `Reconcile` step, after the restore, on each restored system in system Id order. `rewoundTick` is the start tick: the server's value there already includes that tick's client state, so replay from the tick after it. It is `NetworkLoopManager.UnsetTick` when the correction was a snap with no replay target, a reconcile the server sent that landed with no echo of the processed client state tick. A server correction of the controller's client state alone does not raise it: a `Predict` value the server changed lands in that tick's history and raises `OnMembersChanged` with `MemberChangeDirection.Correction` instead.

When something on the system overrides a fixed update or registers a simulation callback, the engine replays it after `ReconcileRequired`, and a subscriber should not replay it a second time. A system nothing simulates is left to the subscriber. Replay it on the `Reconcile` step (an `INetworkLoopStepCallback` returning `NetworkLoopSteps.Reconcile`), bracketing every past tick in tick order:

```
public void BeginReplayTick(uint tick);
public void EndReplayTick();
```

`BeginReplayTick(tick)` opens a replay of that tick: `Validate` members write its slot, `Predict` members read its client state, and `NetworkLoopManager.SimulationType` reads `Replayed`, so the refusals above apply. `EndReplayTick()` returns writes and reads to the live tick. A write to a predicted member outside the bracket lands on the live value and breaks the next result check. Re-simulate the ticks

from `rewoundTick + 1` up to the tick before the current one this way, then apply the current tick's values normally. Systems held for the reconcile stay held until the `Reconcile` step's callbacks have run.

SendReconcile

`SendReconcile(Connection)` is the server side of a forced, full-state correction: the server pushes a system's complete current state to one connection outright, rather than waiting for the result check to find a difference. The correction is queued and sent to that connection at the end of the tick.

```
public void SendReconcile(Connection connection);
```

Only the server can call it; a client attempting to reconcile a system is refused with a logged error, and nothing is queued. A reconcile is the one kind of correction that must never travel upstream, since accepting one from a client would let it overwrite any system on the server outright.

A server reaches for this when a gameplay event invalidates a controller's prediction in a way the result check would not catch: a teleport, a respawn, an out-of-band state change the client could not have predicted from its own client state. The engine also sends one itself for a `Validate` result that differs and when it re-anchors a client that stalled. The reconcile carries the `Predict` values the server applied for its start tick, after validation, and the controller writes them into its history at that tick, raising `OnMembersChanged` with `MemberChangeDirection.Correction` when they differ from its own. Asking again for the same system and connection before the flush queues nothing more. On arrival it is recorded as `MismatchCause.ServerOrdered` and restores and replays like any other reconcile.

Validate results the server found different

For `MemberRole.Validate` members the server compares a hash of the controller's result. When one differs it queues the correcting reconcile and raises `NetworkSystem.ResultMismatched` with a `ResultMismatchReport`: the controller's `Connection`, the server's `Tick` the difference was found on, the controller's `RemoteTick` it pairs with (for matching what the client logged, never for timing) and, when the controller sent its values beside the hash, the `Members` that differed. Each of those answers `TryGetResultMismatchValues(out controllerValue, out serverValue)` until the tick advances. The controller sends the values while its `SystemManager.IsResultValueSendEnabled` is set, the default; turn it off in a release build to keep the hash alone. No violation is raised, so what to do about the client is the game's call.

The report is one instance per system, reused by every result that differs, so read it inside the handler:

```
using System.Numerics;
using Nucleus.Components;
using Nucleus.Systems;

public sealed class ResultMismatchLogger
{
    // Server: subscribe once the system has started.
    public void Watch(NetworkSystem networkSystem) => networkSystem.ResultMismatched += On
ResultMismatched;

    private void OnResultMismatched(ResultMismatchReport resultMismatchReport)
    {
        foreach (NetworkMemberBase networkMember in resultMismatchReport.Members)
        {
            if (networkMember is NetworkMember<Vector3> vectorMember && vectorMember.TryGe
tResultMismatchValues(out Vector3 controllerValue, out Vector3 serverValue))
                System.Console.WriteLine($"Client {resultMismatchReport.Connection.Id}, it
s tick {resultMismatchReport.RemoteTick}: {networkMember.MemberName} was {controllerValue}
, server had {serverValue}.");
        }
    }
}
```

See also

Physics convergence followers can hide the visual jump a rewind produces; see the Physics documentation for how a follower interpolates toward the corrected pose instead of snapping to it.

What control means

Every `NetworkSystem` has three questions attached to it: who is the server, who controls it, and who observes it. They are answered separately, and mixing them up is the most common source of "why did that work on my machine but not over the network" bugs.

The three roles

The server is the server. `NetworkSystem` exposes `IsServerStarted` for this; the server's word on a system's state is final, whatever any client believes.

The controller is the one connection - server or a single client - that drives the system. Control is exclusive: at most one connection controls a given system at a time. `ControllerType` names who that can be:

```
[System.Flags]
public enum ControllerType : byte
{
    None = 0,
    Server = 1 << 0,
    Client = 1 << 1,
    AnyController = Server | Client,
}
```

Ask which one applies to the local peer with `IsController`:

```
if (networkSystem.IsController(ControllerType.Client))
{
    // This peer is the client controlling this system.
}
```

`ControllerType.Server`'s own meaning is exact: "the local peer is the server and no client controls the system." A system is never controller-less from the server's point of view - if no client controls it, the server does.

An observer is any peer the system is replicated to. Observation is separate from both of the above: the server and the controller are single, named connections, while a system can have any number of observers, including none.

Control is exclusive and server-assigned

Only the server assigns a controller, through `SetController`. A client that calls it is rejected outright - the server logs an error and nothing changes. Control replicates from there: every observer is told who the controller is, and `IsController` reads that replicated identity rather than anything a client could assert about itself.

Because control is exclusive, a system is never controlled by two clients at once, and a controller change is a single, well-defined event (`ControllerChanged`) rather than a negotiation.

What control decides

Three things ride on who the controller is, and all three are controller-only regardless of any write-access setting:

- **Who may send client state.** Only the controller's client state is accepted for the system.
- **Who predicts.** The controlling client predicts the system from its own client state, and the server checks

its result and corrects it when they differ. Any other client receives the server's state instead, and during a reconcile holds the system at the server's values while its own systems replay, unless it opts into predicting it as well with `NetworkSystem.IsObserverPredictionEnabled`.

- **Who may write state by default.** `StateWriteAccess.Controller` - "the server, or the single controlling client" - is the default access, and it is also the behavior every system had before write access existed as a setting at all. The controller can always write state; widening that to other observers is a separate, additive Pro setting (`StateWriteAccess.AnyClient`), covered on Letting clients write state.

The same shape holds for RPCs: `RpcSendAccess.Controller` is the default there too, and its own doc comment puts it plainly - "only whoever drives the object may ask it to do anything."

Observation is required under every check

Control and write access decide what a connection is *permitted* to do once it is already being told about the system. Being told about it - observation - is checked first, unconditionally, before either of them is consulted. The server asks the same three questions of a client's state write and of a client's remote call:

> Does the connection observe the system? If not, it is refused.
>
> Is it the system's controller (`ControllerConnectionId`)? If so, it is admitted.
>
> Otherwise it is admitted only when the access in force is the any-client one (`StateWriteAccess.AnyClient` or `RpcSendAccess.AnyClient`).

State writes and remote calls both go through this same check. A client that does not observe a system fails it immediately, regardless of whether it is the controller or the write access has been widened to any client. Widening control or write access never substitutes for being an observer - a connection has to be told about a system before anything it does about that system means anything. Observation itself is decided by the interest system, covered on What area of interest is.

The host case

A host runs the server and a client in the same process. That peer's `IsServerStarted` is true, so it is the server. If it also controls a given system - the ordinary case for a host's own player object - it satisfies `IsController(ControllerType.Client)` as well. And because a host holds every system in its own process, its own peer is exempt from being culled by interest: it observes everything by construction.

So on a host, one process can be server, controller and observer for the same system simultaneously. That collapses three checks that are independent everywhere else into one that always passes, which is exactly what hides the bugs that show up the moment a second machine joins: a missing observer registration, a write-access setting that was never actually exercised, a client state path that assumed the controller and the server were the same connection. None of those fail on a host, because on a host they can't - there is only one peer to be all three roles at once.

Assigning and releasing control

Using Unity? See Giving a client control of an object.

Assigning and releasing control

The server hands control of a `NetworkSystem` to a connection with `SetController`:

```
public void SetController(Connection controllingClientConnection)
```

Pass a `Connection` to make that client the controller, or `null` to return control to the server. Only a peer with a started server may call it; a client that calls it gets a logged error and nothing happens - the call does not act on a non-server caller, it rejects it. An assignment that does not actually change the controlling identity queues nothing, so calling `SetController` with the current controller (or `null` when the server already controls it) is a no-op.

One comparison decides three separate things

A single comparison - "does this connection hold `ControllerConnectionId`" - decides three separate things at once:

- **Client state routing**: which connection's client state the system applies.
- **Prediction**: whether this peer predicts the system locally at all.
- **State write permission**: the controller may always write state, regardless of `StateWriteAccess`.

They're decided by the same comparison because they describe the same relationship from three angles: the connection driving the object, the peer allowed to run it ahead of the network, and the peer allowed to tell the server what its state is. Splitting them would let a system take client state from one connection while writing state for another, which is not a state Nucleus represents.

A client write for a system it does not control - or a client delta that isn't fully absolute, since a client's write always has to be - is rejected by the server and raised as an `UncontrolledStateChangeViolation`:

```
public struct UncontrolledStateChangeViolation : IViolation
{
    public uint SystemId;
}
```

It's raised with `ViolationAction.Ignore`: the ordinary cause is a race (control or permission changing before the client's earlier write lands), not an attack, so the default response is to drop the write and move on rather than punish the connection.

When the controller leaves

The server drops the departing connection's reference automatically; a system does not have to notice its controller leaving on its own. Whether the controlling identity itself is cleared right away or held for a returning connection to reclaim is a retention decision, not part of this page - see Controller retention for holding a departed controller's objects.

Replication

Control is not something game code sends. `SerializeControllerToObservers` writes the current controller and its retention token into the state stream for every delta and full observer, so every peer learns a controller change the same way it learns any other replicated state - no separate message, no extra call.

When prediction is worth its cost

What prediction actually buys

Prediction removes one thing: the wait for a round trip before the controller sees the result of its own client state. Without it, a client presses a key, sends the resulting client state, and only moves once the server's outcome comes back. With it, the client applies that client state locally straight away and the server's outcome arrives later to confirm or correct that guess.

That is the entire benefit. Prediction does not make an object more accurate, more responsive to other peers, or cheaper to replicate. It only hides the round trip from the one peer driving the object.

Prediction costs bandwidth, memory and replay time

Turning prediction on for a `NetworkMember<T0>` is one constructor argument, its role:

```
public NetworkMember(CompressionLevel deltaCompressionLevel = CompressionLevel.Tight, bool
 isAutomaticPoolingEnabled = false, MemberRole role = MemberRole.State, float accuracy = f
loat.NaN, float interpolationSnapThreshold = float.NaN, SendInterval sendInterval = SendIn
terval.Normal, TransmissionMode transmissionMode = TransmissionMode.Interval, bool isExtra
polationEnabled = true, byte correctionSmoothingTicks = 0, bool isCarriedForward = true)
```

A result declared `MemberRole.Validate`, computed from client state declared `MemberRole.Predict`, is predicted by the controller and checked by the server. That does three things, and each one is a real cost:

- **A history, not just a live value.** A predicted member keeps a history deep enough to cover the round trip, so a reconcile can restore the member to a past tick and a replay can read each past tick's client state. Every client keeps `SystemManager.PredictionHistoryTicks` of it, or `SystemManager.DefaultPredictionHistoryTicks` (64) when that is unset, and a member served on a wider interval than the tick rate keeps more. An unpredicted member keeps two slots.
- **A replay whenever a prediction misses.** A matching result costs nothing beyond its hash on the upstream client state, plus the result values beside it while `SystemManager.IsResultValueSendEnabled` is on (the default, so the server can name what differed; turn it off to keep the hash alone). A result the server finds different orders a reconcile, which restores every system the client predicts to the server's values as of the wrong tick, and the engine re-runs the fixed updates (`OnEarlyFixedUpdate` and `OnLateFixedUpdate`, as `SimulationType.Replayed`) for every tick from there to the present. A replay covers about a round trip plus the state interpolation in ticks, all simulated in one frame.
- **Game code that tolerates being re-run.** Anything that happens during a replayed tick happens again, for every tick between the rewind point and the present. That is fine for a movement integration; it recomputes the same answer. It is not fine for a side effect that cannot be undone or repeated safely, so effects belong on `SimulationType.Forward` only. The engine refuses the worst offenders while a replayed tick runs: remote calls, messages, predicted spawns, collection and reference writes and `NetworkAnimator` triggers. A rigidbody cannot be re-stepped per object at all; use `ProjectedRigidbody` for bodies.

The cheaper alternatives

Most objects do not need any of this. In order of cost:

- **An uncontrolled object with interpolation.** No controller, no ring beyond the ordinary two slots, no result check, no replay. The receiver just animates between the values it receives, paced by

`SystemManager.StateInterpolation` (0 by default, capped at `MaximumStateInterpolation`). This is the default for anything nobody is driving.

- **A controller-written value with no prediction.** The controller still writes the member every tick, but its role stays `MemberRole.State`. The controller sees its own writes with the same round-trip delay as everyone else; no local guess, no rewind, no replay. Cheaper than prediction, more responsive than a purely server-driven value the controller doesn't write at all.
- **A purely local visual that is never replicated.** Not a `NetworkMember<T0>` at all. Particle effects, camera shake, UI feedback: anything that only needs to look right on the machine that triggered it costs nothing on the wire and needs no reconciliation, because there is nothing to reconcile.

The rule that disqualifies a candidate

If the value only ever needs prediction because something *reads* it (a position feeding a camera, a resource bar), the reconcile replay problem is contained: worst case, the display corrects itself. Prediction turns dangerous the moment the tick being replayed can trigger a side effect the replay can't take back: spawning an object, awarding score, firing an RPC. The engine refuses remote calls, messages and predicted spawns during a replay, but it cannot know about your own side effects: awarding score or writing to a static from a replayed tick happens again on every replay. Gate side effects on `SimulationType.Forward`, or keep the member driving them unpredicted.

Why most of a world has nothing to predict

Prediction only matters to the peer that controls the object, because it's the only peer waiting on its own round trip. `NetworkMember<T0>` checks this directly: a predicted result skips its own upstream serialization on a pure client that predicts its system, which is the controller (`NetworkSystem.IsController(ControllerType.Client)` and not server-started), or an observer with `NetworkSystem.IsObserverPredictionEnabled` set, because its value is computed locally from client state and checked by the server rather than sent. An object nobody controls has no local guess to make in the first place: there's no round trip to hide, because there's no client state driving it locally. Most objects in a networked world fall into that category: static geometry, background actors, anything the server alone moves. That's the uncontrolled-plus-interpolation case above, and it's the default, not the exception.

The choice is per member

`role` is a per-`NetworkMember<T0>` constructor argument, not a project setting or even a per-object one. A single component can predict its position while leaving its health, ammo count, or any other member as ordinary `MemberRole.State`; each one is its own opt-in. No shipped component turns prediction on; the Unity demo `Demos/Prediction/ValidatedJump` shows it, with the walk direction and jump press count of `ValidatedJumpMovementComponent` as `MemberRole.Predict` client state and every movement result as `MemberRole.Validate`. Prediction is available, per member, wherever the predict-and-replay cost is worth paying, but nothing turns it on by default.

Predicted spawning is Pro, predicted motion is not

Predicted members, the client state that drives them, and the reconcile replay described above are available in every build. Predicted *spawning*, predicted *despawn*, and spawn compensation are Pro-only: `NetworkSystem.PredictedSpawn.Pro.cs`, `SystemManager.PredictedSpawn.Pro.cs`, `NetworkSystem.SpawnCompensation.Pro.cs`, and `SystemManager.SpawnCompensation.Pro.cs` don't exist in a Free build. Predicting an object's motion or state costs nothing extra to license; predicting the moment an object comes into or out of existence does.

Authenticating clients

The contract

`IClientAuthenticator` has one member:

```
public interface IClientAuthenticator
{
    void OnClientConnecting(Connection connection);
}
```

`OnClientConnecting` is called once per connecting client, when its remote connection is established and before it is authenticated. The engine defines no credential and no wire format of its own — an implementation drives whatever handshake it wants, using ordinary `IMessage` types, and reports the outcome by calling `ServerManager.ApproveClient` or `ServerManager.DenyClient`.

The decision does not have to be immediate. `OnClientConnecting` can return right away and the outcome can arrive any number of ticks later, after a backend round trip. Until it does, the connection stays unauthenticated: the engine delivers it no message that requires authentication, and it observes nothing.

Call ApproveClient or DenyClient once you have a verdict

Call one of these from anywhere in your authenticator once you've reached a verdict:

```
void ApproveClient(Connection connection, string identity = null);
void DenyClient(Connection connection);
void DenyClient(Connection connection, ulong timeoutMilliseconds);
void DenyClient<T0>(Connection connection, T0 notice) where T0 : IMessage, new();
void DenyClient<T0>(Connection connection, T0 notice, ulong timeoutMilliseconds) where T0
: IMessage, new();
```

`ApproveClient` marks the connection authenticated and stamps `identity` onto `Connection.Identity`, the value that names this client across reconnects. Pass `null` to leave it unidentified. What the identity means, and how you correlate a returning peer with what it previously owned, is your code's business — the engine only carries the value.

`DenyClient(connection)` refuses the client and disconnects it, telling it nothing. That's the right default for a failed authentication: a refused peer is owed no explanation, and an explanation can help an attacker. `DenyClient(connection, notice)` sends a message of your own on the reliable channel before disconnecting, if you want the client to know why. The notice is any `IMessage` type you define, and its type is how the client tells a refusal apart from any other disconnect. It needs no authentication exemption: leave `IsAuthenticationRequired` at its default, and the refused client still receives it. A refusal is a kick, so the notice and the optional timeout behave exactly as they do for kicking a client.

A connection that's already resolved, or gone, is ignored by any of these calls.

The shipped default

Until you install your own, `ServerManager` uses `AddressClientAuthenticator`:

```
public sealed class AddressClientAuthenticator : IClientAuthenticator
{
    public void OnClientConnecting(Connection connection) =>
        connection.CoreManager.ServerManager.ApproveClient(connection, connection.RemoteAd
dress);
}
```

It approves every client immediately and identifies it by `Connection.RemoteAddress` — the address the server observed it connecting from, which can't be forged, since a peer spoofing its source address never receives the return traffic and never completes a connection.

An address is a location, not a player identity. Two clients behind one NAT — housemates, an office, a carrier-grade NAT — connect from the same address and resolve to the same identity, so code that restores a returning player's objects by identity hands the second client the first one's. The port is deliberately left out of the identity, so a client that drops and reconnects from a new port still matches; including it would break every reconnect on the same network.

This is fine for development, LAN, and single-address deployments. Anything else needs an authenticator of your own.

Installing your own

```
void SetAuthenticator(IClientAuthenticator clientAuthenticator);
```

Set this before the server starts listening — a client that connects before it's set is resolved by whichever authenticator was in force at the time. Passing `null` restores `AddressClientAuthenticator`.

Timeout

```
public ushort AuthenticationTimeoutSeconds = 30;
public const ushort UnsetAuthenticationTimeout = 0;
```

`ServerManager.AuthenticationTimeoutSeconds` bounds how long a connecting client can go unresolved before the server disconnects it, measured from the moment its remote connection is established. The default is 30 seconds. A silent client, or an authenticator that never reports an outcome, would otherwise hold a connection slot open indefinitely.

The check runs once per tick, on the `TickAdvance` step. Set `AuthenticationTimeoutSeconds` to `UnsetAuthenticationTimeout` (0) to disable it.

Events this manager raises

On the server, `ServerManager` raises:

- `ClientAuthenticated(Connection connection)` — after the client is marked authenticated and its `Identity` is stamped, so a handler can read both.
- `ClientAuthenticationDenied(Connection connection)`: for a client that was denied or timed out, never for one that merely dropped its connection. It is raised before the client is disconnected.

On the client, `ClientManager` raises:

- `LocalClientAuthenticated(Connection localClientConnection)` — once this client's connection is authenticated and carries its assigned Id.

Approval sends the client an `AuthenticationResponse`, a reliable message carrying the `ClientId` the server assigned it — the Id the client learns itself by. It's sent before anything else, so the client knows its own identifier before any other state that might reference it.

A worked example: token authentication

A token exchange needs a message type for the client to present its token, and an authenticator that validates it and calls `ApproveClient` or `DenyClient`. Since the client isn't authenticated yet when it sends this, the message must opt out of the default requirement:

```
[NetworkType]
public struct TokenAuthentication : IMessage
{
    public bool IsAuthenticationRequired => false;

    public string Token;
}
```

`IMessage.IsAuthenticationRequired` defaults to `true`, so a message type says nothing unless it overrides it — a type that forgot the override would be refused from every unauthenticated sender, including the client trying to authenticate with it.

```
public sealed class TokenClientAuthenticator : IClientAuthenticator
{
    public TokenClientAuthenticator(MessageManager messageManager)
    {
        messageManager.RegisterMessageHandler<TokenAuthentication>(OnTokenAuthentication);
    }

    public void OnClientConnecting(Connection connection)
    {
        // Nothing to do yet; wait for the client to present its token.
    }

    private void OnTokenAuthentication(in MessageContext messageContext, TokenAuthenticati
on tokenAuthentication)
    {
        Connection connection = messageContext.SenderConnection;

        if (TryResolveIdentity(tokenAuthentication.Token, out string identity))
            connection.CoreManager.ServerManager.ApproveClient(connection, identity);
        else
            connection.CoreManager.ServerManager.DenyClient(connection, new TokenRefusedNo
tice { Reason = "Invalid token." });
    }
}
```

The refusal notice is an ordinary message of your own, and the client registers an ordinary message handler for it to show the reason:

```
[NetworkType]
public struct TokenRefusedNotice : IMessage
{
    public string Reason;
}
```

The client sends its `TokenAuthentication` message on its own connection as soon as it connects. A token presented this way is a bearer credential: until the transport is encrypted it can be read off the wire and replayed, so it must be server-generated, unguessable, opaque about the identity it maps to, validated against server-held state so an unrecognized value fails closed onto a fresh identity rather than an assumed one, and expired or rotated on use.

Unity has no authenticator component, so set one from a script

There is no authenticator component. Call `ServerManager.SetAuthenticator` from a script once the managers exist.

Connections

What a Connection is

A `Connection` represents one peer: a client from the server's point of view, or the server from a client's point of view. It is a pooled object (`Connection : IPoolResettable`), rented when a peer arrives and returned when it leaves.

Because instances are pooled, a `Connection` reference is only valid for the instant you have it. Do not store the instance itself, and do not store its `Id` past the moment you read it. The next peer to connect can be handed the very same object, and `Id` values are recycled the same way. If you need to remember a peer for longer than that, hold a `ConnectionHandle` instead (see below).

Identity and role

Member	Meaning
`Id`	The connection's unique Id across all transports. `UnsetId` (`0`) means no Id is assigned.
`IsIdUnset`	`true` when `Id == UnsetId`.
`IdBySocket`	The Id assigned by the socket-level Transport. Unique within that transport, not across transports. Server sockets always read `ServerIdBySocket`.
`ServerIdBySocket`	Constant (`uint.MaxValue`) used for `IdBySocket` when the connection is a server socket.
`IsServer`	`true` when `IdBySocket == ServerIdBySocket`.
`IsClient`	`!IsServer`.
`IsConnected`	`true` when `LocalState == LocalConnectionState.Connected`.
`IsAuthenticated`	`true` when this side is a client that has authenticated, or when `IsServer`, or when `IsEmulated`.
`IsEmulated`	`true` for a connection that uses no socket, such as a fake or test peer.
`IsPeerStandIn`	`true` when this Connection names a peer the server told a client about, rather than a link this peer itself holds. A stand-in carries identity only; there is no socket behind it.
`IsHostLoopback`	`true` when this Connection is one half of a host's own loopback pair (the same peer at both ends).

ConnectionHandle

`Connection.Id` alone is not safe to hold: identifiers are recycled just like the pooled instances are. `ConnectionHandle` fixes that by pairing the `Id` with a `Generation`, a counter bumped every time the

underlying instance is rented.

```
public readonly struct ConnectionHandle : IEquatable<ConnectionHandle>
{
    public readonly uint Id;
    public readonly uint Generation;

    public bool IsValid => Id != Connection.UnsetId;
}
```

Get a peer's handle from `Connection.Handle`, and resolve it later through `TransportManager.TryGetConnection`:

```
ConnectionHandle handle = connection.Handle;

// ... later, possibly after the peer disconnected and another peer connected ...

if (transportManager.TryGetConnection(handle, out Connection resolved))
{
    // resolved is still the same peer the handle was taken from.
}
```

If the original connection has been recycled, or its Id reissued to a different peer, the generation no longer matches and resolution fails rather than silently naming whoever arrived next. `ConnectionHandle` is a plain value type: it allocates nothing, needs no pooling, and is local to the peer that took it (it never goes on the wire).

RemoteAddress versus Identity

`RemoteAddress` is where the connection connected from, as the Transport reports it (the peer's address for a socket transport; `null` when the transport has no such notion, such as emulated or in-memory connections). The port is deliberately excluded, so a reconnecting peer's value stays stable across the fresh port it arrives on. It is observed by the server rather than asserted by the client, so it cannot be forged — but several clients behind one NAT share an address. `RemoteAddress` identifies a location, not a player.

`Identity` is the value naming a client across reconnects, stamped by the client authenticator when the client is approved. It is `null` until then, and stays `null` if the authenticator supplies none. Anything that needs to recognize the same player across a reconnect should key on `Identity`, not `RemoteAddress`.

Finding connections

`TransportManager` exposes the active set and several lookups:

- `ActiveConnections` — the connections currently known to this peer. On the server this is the remote clients; on a client it is the peer stand-ins the server has reported (never the local client itself).
- `TryGetConnection(uint id, out Connection connection)` — resolves this peer's own Connection for an Id.
- `TryGetConnection(ConnectionHandle handle, out Connection connection)` — resolves a handle, failing if the generation is stale.
- `TryGetLocalClientConnection(out Connection connection)` — this peer's own connection as a client.
- `TryGetServerConnection(out Connection connection)` — this peer's own server socket connection.

Reading connection state

Each `Connection` tracks two states:

- `LocalState` — the state of the connection this device created (connecting your own client, or the server socket when hosting). Typed as `LocalConnectionState`.
- `RemoteState` — the state of the connection as seen from the remote side. Typed as `RemoteConnectionState`, a smaller enum with only `Disconnected` and `Connected`.

`LocalConnectionState`:

```
public enum LocalConnectionState : byte
{
    Disconnected = 0,
    Connecting = 1,
    Connected = 2,
    Disconnecting = 3,
    Error = 4,
    TimedOut = 5,
}
```

`Disconnected` is the zero value, so an unwritten state reads as down. The values are never written to the wire.

`ConnectionStateExtensions` (in `LocalConnectionStateExtensions.cs`) adds two helpers, both on `LocalConnectionState`:

```
public static bool IsConnectedOrDisconnected(this LocalConnectionState localConnectionStat
e);
public static bool IsAnyDisconnected(this LocalConnectionState localConnectionState);
```

`IsConnectedOrDisconnected` is true only for `Connected` or `Disconnected`. `IsAnyDisconnected` is true for anything except `Connecting` or `Connected` — this covers `Disconnecting`, `Disconnected`, `Error`, and `TimedOut`.

`TransportManager` raises an event for each kind of state change:

```
public event ConnectionStateChangedHandler ConnectionLocalStateChanged;
public event ConnectionStateChangedHandler ConnectionRemoteStateChanged;

public delegate void ConnectionStateChangedHandler(ConnectionStateChange connectionStateCh
ange);
```

Each carries a `ConnectionStateChange`:

```
public readonly struct ConnectionStateChange
{
    public readonly Invoker Invoker;
    public readonly Connection Connection;
    public readonly bool IsRemoteStateChange;
}
```

`Invoker` says which side raised it, `Connection` is the connection that changed, and `IsRemoteStateChange` tells you whether `RemoteState` or `LocalState` changed.

Describing a connection

`ConnectionExtensions.AsString` gives a human-readable summary for logging, safe to call on a `null` connection:

```
public static string AsString(this Connection? connection);
```

It reports the connection's `LocalState`, `Id`, `IdBySocket`, transport Id, `IsServer`, and `IsAuthenticated`.

For per-connection round-trip time and packet loss counters, see the diagnostics reference.

Creating a CoreManager

> **Using Unity?** See The Nucleus Core Manager component.

`CoreManager` is the root of a Nucleus session. Construct one and the session is running; there is no separate `Initialize` or `Start` call.

```
private CoreManager _coreManager = new();
```

That single line, taken from `TransportOperation`, is a complete bootstrap: by the time the constructor returns, every manager exists, the managers know about each other, and the network loop is ticking.

The constructor

```
public CoreManager(uint tickRate = NetworkLoopManager.DefaultTickRate, INetworkLoopStepPro
vider? networkLoopStepProvider = null)
```

Both arguments are optional.

- `tickRate` is the number of network ticks the session runs per second, fixed for the session's lifetime. Values outside `NetworkLoopManager.MinimumTickRate` through `NetworkLoopManager.MaximumTickRate` are clamped into range and logged. Every rate-derived value the managers hold is baked from this during construction, so it cannot change afterward.
- `networkLoopStepProvider` drives the network loop's steps. Leave it `null` to get the default provider, which steps the loop from the thread pool on a timer.

Supply your own `INetworkLoopStepProvider` here, not after construction. Nothing steps the loop until the constructor's last line, so a provider passed to the constructor means the default provider never runs at all, and the managers built above it register into a loop that is standing still while they do. Installing a provider after construction instead leaves the default provider driving the loop from the thread pool for the whole of that bring-up — a second thread writing the loop's collections while your code is still writing them too. This is why a host that runs its own loop names its provider in the constructor call rather than swapping one in later.

What construction builds, and in what order

The constructor builds each manager in sequence, passing itself (`this`) to each one, then tells every manager the whole graph exists, and only then starts the loop:

```
_managers.Add(NetworkLoopManager = new(this, tickRate));
_managers.Add(TransportManager = new(this));
_managers.Add(ServerManager = new(this));
_managers.Add(ClientManager = new(this));
_managers.Add(SystemManager = new(this));
_managers.Add(InterestManager = new(this));
_managers.Add(PacketManager = new(this));
_managers.Add(MessageManager = new(this));
_managers.Add(RpcManager = new(this));
_managers.Add(ViolationManager = new(this));
AddBundleManager();          // Pro only
_managers.Add(SceneManager = new(this));
AddWorldPersistenceManager(); // Pro only

foreach (ManagerBase manager in _managers)
    manager.ManagersInstantiated();

NetworkLoopManager.StartNetworkLoop(networkLoopStepProvider);
```

`NetworkLoopManager` is built first because it owns the clamped tick rate, and every manager built after it bakes rate-derived values from that. Every manager is then given a second pass, `ManagersInstantiated`, to resolve dependencies on the others now that the whole graph exists — a manager built earlier can still find one built later. `StartNetworkLoop` runs last of all, after every manager has registered its loop callbacks, so the loop never steps against a graph that is only half-built.

Teardown unwinds this in reverse: each manager's `Deinitialize` runs from the last-constructed manager back to the first, because a manager built later typically registered handlers onto managers built earlier.

The manager fields

Every build exposes these eleven `readonly` fields:

Field	Type
`NetworkLoopManager`	`NetworkLoopManager`
`TransportManager`	`TransportManager`
`ServerManager`	`ServerManager`
`ClientManager`	`ClientManager`
`SystemManager`	`SystemManager`
`InterestManager`	`InterestManager`
`SceneManager`	`Scenes.SceneManager`
`PacketManager`	`PacketManager`
`MessageManager`	`MessageManager`
`RpcManager`	`RpcManager`
`ViolationManager`	`ViolationManager`

Pro-only fields

A Pro build adds two more, contributed by partial class files (`CoreManager.Bundles.Pro.cs` and `CoreManager.Persistence.Pro.cs`) that hook into the constructor through the `AddBundleManager`

and `AddWorldPersistenceManager` partial methods:

Field	Type	Added by
`BundleManager`	`BundleManager`	`CoreManager.Bundles.Pro.cs`
`WorldPersistenceManager`	`WorldPersistenceManager`	`CoreManager.Persistence.Pro.cs`

In a Free build, neither partial method has an implementation, the calls compile away, and the fields do not exist on `CoreManager` at all — there is no null placeholder to check against. `BundleManager` is for content delivered from outside the build (Addressables, a patcher, a CDN); without it a build can only spawn what shipped inside it. `WorldPersistenceManager` is for putting a world on disk and building it back; without it a world lives exactly as long as the process holding it. Unlike the other manager fields, these two are not `readonly`, because they are assigned from a hook the constructor calls rather than from the constructor body directly.

Instance

```
public static CoreManager Instance { get; private set; }
```

`Instance` is a first-write-wins convenience for the common single-manager case: the first `CoreManager` constructed takes the slot and keeps it until it tears down, at which point the slot passes to the next live `CoreManager`, or becomes `null` if none remain. It can only ever name a live manager — a torn-down one is never left sitting in the slot.

```
public static bool EnsureSetInstance(CoreManager coreManager)
```

`EnsureSetInstance` re-points the slot to a specific live `CoreManager`. It returns `false` and leaves the slot untouched if you pass `null` or a `CoreManager` that has already been deinitialized; otherwise it returns `true`.

A process that only ever runs one `CoreManager` can read `Instance` freely. A process running several at once — a host pair, a two-peer test — should pass the `CoreManager` it means explicitly instead, since `Instance` can only ever name one of them.

Several CoreManagers in one process

Constructing more than one `CoreManager` in the same process is normal and supported; nothing about construction assumes there is only one. Each `CoreManager` owns its own complete set of managers — its own `TransportManager`, `ServerManager`, `NetworkLoopManager`, and so on — and none of that state is shared between instances. What is shared across every `CoreManager` in the process is the static registry that resolves `Instance` and the live-manager list `EnsureSetInstance` and teardown walk; that bookkeeping is process-wide, the managers themselves are not.

Where configuration lives

There is no engine-wide settings object. Each knob belongs to the manager it configures — the tick rate lives on `NetworkLoopManager`, transport configuration lives on `TransportManager`, and so on for the rest of the fields above. Configuring a session means setting fields on the specific manager that owns the behavior you want to change, documented on that manager's own reference page.

Driving the loop yourself

Why this page exists

FishNet has no equivalent of this page, because every FishNet entry point starts inside a Unity scene and Unity already owns the frame loop. Nucleus is a plain .NET engine first, so a host that is not Unity, a dedicated server process, a simulation harness, a game with its own engine loop, has to supply the clock that steps the network loop itself. `Nucleus/Managers/NetworkLoop` is where that clock plugs in.

The default is enough for most hosts

If you construct a `CoreManager` and pass nothing for the loop step provider, it gets a `SystemNetworkLoopStepProvider`. This is a `System.Timers.Timer` running at a 1 ms interval paired with a `Stopwatch`, raising loop steps from a thread pool thread on every elapsed callback. It throttles its variable-update cadence to an 8 ms simulated frame interval, and lowers that interval to match the tick interval once the configured tick rate rises above roughly 125 Hz (`NetworkLoopManager.MaximumTickRate` is 128), so the achieved rate does not fall short of what was asked for.

Nothing needs writing to use it. Write your own provider only when you have a clock the default cannot see: a dedicated server's own fixed-step loop, or a test that must step deterministically rather than on a timer.

The provider contract

A provider implements `INetworkLoopStepProvider`, in `Nucleus/Managers/NetworkLoop/INetworkLoopStepProvider.cs`:

```
public interface INetworkLoopStepProvider
{
    public bool IsStarted { get; }

    public bool IsTickFrame { get; }

    public void Initialize(NetworkLoopManager networkLoopManager);

    public void Start();

    public void Stop();

    public void Return();
}
```

`Initialize` hands the provider the `NetworkLoopManager` it will drive; `Start` and `Stop` bracket the period the provider may raise steps; `Return` tells the provider it is being let go and may pool itself. `Stop` is always called before `Return`.

`IsTickFrame` answers whether the frame the provider is driving runs a tick. The engine reads it to decide whether the frame's serialize steps write state and raise their callbacks. A provider built on `NetworkLoopStepDriver` forwards `NetworkLoopStepDriver.IsTickFrame`. A provider that invokes the steps itself answers true before the steps of a pass that ticks and false otherwise; left false, the pass sends its messages but writes no state.

Building a provider on NetworkLoopStepDriver

Both shipped providers, `SystemNetworkLoopStepProvider` and the Unity integration's, drive a `NetworkLoopStepDriver` rather than calling `NetworkLoopManager.InvokeNetworkLoopStep` themselves. The driver is the shared cadence engine: it owns the tick accumulator, the two-tick catch-up clamp, and the one canonical order the eleven `NetworkLoopSteps` are emitted in, so a provider only has to feed it a frame delta and let it decide when a tick actually runs.

```
public sealed class HostClockStepProvider : INetworkLoopStepProvider
{
    private readonly NetworkLoopStepDriver _driver = new();

    public bool IsStarted { get; private set; }

    public bool IsTickFrame => _driver.IsTickFrame;

    public void Initialize(NetworkLoopManager networkLoopManager)
    {
        float tickIntervalMilliseconds = 1000f / networkLoopManager.TickRate;

        _driver.Initialize(networkLoopManager, tickIntervalMilliseconds);
    }

    public void Start() => IsStarted = true;

    public void Stop() => IsStarted = false;

    public void Return() { }

    // Called once per frame from your host's own loop.
    public void OnHostFrame(float frameDeltaMilliseconds)
    {
        if (!IsStarted)
            return;

        _driver.AdvanceEarly(frameDeltaMilliseconds, isVariableUpdateAllowed: true);
        _driver.AdvanceLate(frameDeltaMilliseconds, isVariableUpdateAllowed: true);
    }
}
```

`AdvanceEarly` accumulates the frame, decides whether a tick fires, and invokes everything from `EarlyVariableUpdate` through the mid-cycle `VariableUpdate`. `AdvanceLate` finishes the same frame: on a frame that ticks it runs `EarlySerialize`, `LateSerialize` and `TickAdvance`, on every frame it sends that frame's messages, and it ends with `LateVariableUpdate`. The split exists so a host that runs distinct early/late phases (Unity's `Update`/`LateUpdate`) can call them separately; a host with one frame callback, like the one above, calls both back to back, the same way `SystemNetworkLoopStepProvider` does in its timer handler.

Naming your provider in the constructor

Pass your provider to the `CoreManager` constructor:

```
CoreManager coreManager = new(tickRate: 30, networkLoopStepProvider: new HostClockStepProv
ider());
```

`NetworkLoopManager` also exposes `UseNetworkLoopStepProvider(INetworkLoopStepProvider)`, and it does correctly stop and return the provider it replaces. But by the time you could call it after construction, the default

`SystemNetworkLoopStepProvider` is already stepping the loop from a thread pool thread, racing every registration the rest of your bring-up code makes in the meantime. Name your provider in the constructor and the default never starts at all.

Hand-driving for tests and deterministic simulation

For deterministic simulation, or a test that must control exactly when a tick happens, drive the loop directly with `NetworkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps, StepDelta)` instead of going through a provider's timer. Install a provider that never steps on its own and lets you say whether a pass ticks:

```
public sealed class HandDrivenStepProvider : INetworkLoopStepProvider
{
    public bool IsStarted { get; private set; }

    // Set by the code driving the loop: true before the steps of a pass that ticks.
    public bool IsTickFrame { get; set; }

    public void Initialize(NetworkLoopManager networkLoopManager) { }

    public void Start() => IsStarted = true;

    public void Stop() => IsStarted = false;

    public void Return() { }
}
```

Then set its `IsTickFrame` to true and invoke all eleven steps yourself, in the framework's canonical order:

```
HandDrivenStepProvider stepProvider = new();
CoreManager coreManager = new(networkLoopStepProvider: stepProvider);

NetworkLoopManager networkLoopManager = coreManager.NetworkLoopManager;
StepDelta delta = new(0, 0, 0);

stepProvider.IsTickFrame = true;

networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.EarlyVariableUpdate, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.EarlyStateUpdate, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.LateStateUpdate, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.Reconcile, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.EarlyFixedUpdate, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.LateFixedUpdate, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.VariableUpdate, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.EarlySerialize, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.LateSerialize, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.TickAdvance, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.LateVariableUpdate, delta);
```

This is exactly the order Nucleus's own test harness drives a full tick in. Do not reorder or skip steps: framework work is woven into specific steps (packet receive and message dispatch on `EarlyVariableUpdate`, state apply and RPC dispatch on `LateStateUpdate`, state serialization and the send of messages, RPCs and kicks on `LateSerialize`, the tick advance at the end of `TickAdvance`, among others), and tests that assert on replicated state depend on that order holding. Only `TickAdvance` advances `Tick`. Left false, `IsTickFrame` makes `EarlySerialize` and `LateSerialize` write no state and raise no callbacks, though the pass still sends its messages.

One loop, one manager

Each `CoreManager` owns exactly one `NetworkLoopManager`, and that manager enforces that only one thread drives it at a time. `InvokeNetworkLoopStep` claims the calling thread for the duration of the step; a second thread that calls in while a step is already executing is turned away rather than run, and the rejection is logged at most once a minute so a stuck second driver does not flood the log. A re-entrant call from the thread that already holds the step is not a second driver, and runs normally.

In practice this means: do not call a hand-driven `InvokeNetworkLoopStep` from one thread while a real provider (the default timer, or your own) is also stepping the same manager. Pick one driver for a given `CoreManager`'s whole life, either a provider or hand-driving, never both at once.

Host mode

What a host is

A host is one process running both roles at once: it starts a server and connects its own client to that server. `TransportManager.IsServerStarted` and `TransportManager.IsClientStarted` are both true, and `IsHostStarted` is defined as `IsServerStarted && IsClientStarted`. Nothing else distinguishes host mode from a pure server or a pure client: the same `CoreManager`, the same transport, both halves live in one process.

A host holds one Connection for its own player

A host's own client is described by one `Connection`. The server side keeps a roster entry for every connected player, and for the host's own client that roster entry is the client side's local `Connection` itself, under either Synapse pairing. A call that names the host's own player by its local `Connection`, such as `RpcTarget.To` or `RpcTarget.ObserversExcept`, therefore matches the roster entry the server holds.

Compare players by `Connection.Id` all the same:

```
bool isSelf = someConnection.Id == localClientConnection.Id;
```

A `Connection` is pooled and reused once its link ends, so a reference kept past that point names whoever rents the instance next, while the Id is what names a player on the wire.

Or resolve each side explicitly through the `TransportManager` rather than holding a reference across the two halves:

```
transportManager.TryGetLocalClientConnection(out Connection localClient);
transportManager.TryGetServerConnection(out Connection localServer);
```

`TryGetLocalClientConnection` returns this peer's own client `Connection`; it is false on a pure server. `TryGetServerConnection` returns this peer's own server `Connection`; a controlling client uses it to find the upstream observer target. Both resolve to the owner-held instance for that side rather than scanning transports.

IsHostLoopback

Both halves of the pairing above are marked `Connection.IsHostLoopback`, and the client half is also the server's roster entry for its own client. The mark identifies a link whose two ends are the same process. It is neither `IsEmulated` (no socket at all) nor `IsPeerStandIn` (a peer this client cannot reach directly) — it names a real, live link where both ends happen to be this peer.

State addressed across a loopback link is discarded on arrival rather than applied: whatever was sent was written by the process it would be delivered to, so there is nothing to learn from serializing it and reading it back. Neither half owes the other an acknowledgment either; the sending half settles its own outstanding ticks as it serializes them.

A `Transport` answers the same question at the socket level, before a `Connection` can be marked at all, through the abstract `Transport.IsOwnClientConnection(Connection serverSideClientConnection)`. It exists because a `Connection` itself carries no endpoint or handshake to prove the two ends are the same process — only the transport underneath it knows what it's holding. A transport that cannot tell must answer false rather than guess: a false answer costs a host the in-process delivery of its own traffic, but a wrong true answer would hand a stranger connecting over loopback this peer's own server half.

HostLoopbackDelivery

`TransportManager.HostLoopbackDelivery` controls when a host's two halves see what one of them addressed to the other. It is settable only to `HostLoopbackDelivery.Immediate`; setting anything else is refused with a `LogError` rather than silently ignored:

```
// Refused: logs an error and leaves HostLoopbackDelivery at Immediate.
transportManager.HostLoopbackDelivery = HostLoopbackDelivery.Deferred;
```

`Immediate` (the default, and the only value a game may set) runs the receiving half's handlers at the send site, before the send call returns — the same shape every other callback in the engine already uses. `Deferred` would run those handlers a tick later, at the loop step a genuinely remote copy lands on; it exists only so the engine's own tests can measure that alternative, reached through an internal setter game code cannot call.

A message delivered this way is never serialized. Under `Immediate` its handler runs synchronously at the send site, with nothing queued. `Deferred` queues it instead, and the queue is run at `NetworkLoopSteps.EarlyVariableUpdate`, the same step a wire copy would be read on.

Callbacks a host raises for itself

`TransportManager.PeerConnectionDiscovered` and `PeerConnectionDropped` normally hand a client a stand-in Connection (`Connection.IsPeerStandIn` true) for a peer it was told about. On a host these events instead name the server's own live roster entry for a joining or leaving peer — `IsPeerStandIn` reads false, because the Connection carries a real link rather than a bare identity. A host is told through `RaiseHostPeerConnectionDiscovered` / `RaiseHostPeerConnectionDropped` rather than through the roster, because the roster message it writes to its own client is the one discarded on its own loopback.

Neither event is ever raised for the host's own client. No client is told about itself.

The self-delivery hazard

Never pair an addressed send (`RpcTarget.To(connection)`) with immediate self delivery (`RpcSelfDelivery.Immediate`) on a surface the host's own client also listens on. On a host, the process running the server is the same process running that client, so a handler registered for the surface runs once for the send and once more for the host's own copy — the host ends up running every player's copy of a call meant for one. See the RPC page for the detail on `RpcSelfDelivery` and `RpcTarget`.

Starting a host

In Unity, set `UnityTransportManager`'s automatic start mode to `NetworkStartMode.Host` in the inspector, or call `StartHostAsync()` directly:

```
await unityTransportManager.StartHostAsync();
```

`StartHostAsync` connects every added transport as both a server and a client, in that order.

From plain C#, connect both invokers on the transport:

```
await transport.ConnectAsync(Invoker.Server);
await transport.ConnectAsync(Invoker.Client);
```

Core, Managers and the Tick Loop

Pages in this category:

- Connections
- Server, client and host roles
- Session events
- Ticks, tick rate and time
- Host mode
- The network loop steps
- Threading and lifetime
- Creating a CoreManager
- Authenticating clients
- Kick and refusal notices
- The manager map
- Driving the loop yourself
- Kicking and disconnecting a client
- Shutting a session down
- Keeping the world across a disconnect

Kicking and disconnecting a client

Kicking a client

`ServerManager.KickClient(Connection)` marks a connection for removal:

```
coreManager.ServerManager.KickClient(connection);
```

An overload takes an explicit `KickOptions` policy:

```
coreManager.ServerManager.KickClient(connection, KickOptions.BlockOutgoing);
```

`Connection` also exposes a shorthand that calls back into `ServerManager` under the default policy:

```
connection.Kick();
```

`KickClient` refuses only three things: a null `connection`, a `connection` whose `Transport` is already null (it has already gone back to the pool, so there's nothing left to disconnect), and a `connection` that already has a kick pending. It does not check whether this peer is a server; see Relation to the violation pipeline below for where that check actually lives.

KickOptions

`KickOptions` is a `[Flags]` enum governing what the server stops doing for a client during the window between the kick being marked and executed:

Value	Effect
`None`	Keep serving the client normally until the kick executes.
`PurgeOutgoing`	Discard the state of every kind and the remote calls already queued to the client. Messages already queued are kept, which is how a reason sent before the kick still reaches the client.
`BlockOutgoing`	Queue nothing further to the client: no deltas, no fulls, no despawns, no recovery, no remote calls and no messages.
`BlockIncoming`	Discard everything received from the client instead of processing it. The datagrams still arrive; the server just stops believing them.
`Default`	`PurgeOutgoing

Telling the client why

A kick can carry a notice: any message type of your own that explains it. Pass it to the kick and the engine sends it on the reliable channel, then marks the kick:

```
[NetworkType]
public struct KickedNotice : IMessage
{
    public string Reason;
}

coreManager.ServerManager.KickClient(connection, new KickedNotice { Reason = "Idle for too
 long." });

// The same call through the Connection shorthand:
connection.Kick(new KickedNotice { Reason = "Idle for too long." });
```

The order is the reason these overloads exist. `BlockOutgoing` refuses every message queued after the kick is marked, and `PurgeOutgoing` keeps the messages already queued, so the notice has to be sent first. Writing the two calls by hand is easy to get backwards: a message sent after `Kick()` is silently dropped. Pass the notice to the kick and the order is always right.

The notice is sent only to a client this call actually kicks. A kick against a client that already has one pending is ignored, notice and all, so a client is never told two different reasons.

On the client, the notice arrives as an ordinary message, so you handle it with an ordinary message handler, and it is dispatched before the client sees its own disconnect (see Kick and refusal notices). The notice's type is how the client tells a kick from any other disconnect: define one type per reason you want to distinguish, or one type with a reason field, and register a handler for it. Give a notice type the default `IsAuthenticationRequired`; a client being kicked is already authenticated, and even a refused client reads a notice sent through `DenyClient` (see Authenticating clients).

Every `KickClient` overload also takes a `KickOptions` policy, and the notice overloads take one too: `KickClient<T0>(connection, notice, kickOptions)`.

When the kick actually runs

Marking a connection doesn't tear it down immediately. The policy applies at once, and the disconnect itself is queued and runs once per frame, at `LateSerialize`'s place in the network loop, after that frame's messages have been sent, so a notice has already gone out by the time the link is asked to close. A kick marked after that point (from a `TickAdvance` or `LateVariableUpdate` callback) runs on the next frame.

At that point the transport is asked to close the link once the client has received what was already sent to it, so a client that receives its notice is normally gone almost at once. A client that never acknowledges is bounded by a timeout: if the link is still open once the timeout has passed, measured from the kick, it is closed outright at the start of the next tick, before anything that tick receives is read.

```
public ulong KickTimeoutMilliseconds = 500;
```

`ServerManager.KickTimeoutMilliseconds` is the timeout every kick uses unless you pass one. Keep a timeout of your own above 250 ms on Synapse: it resends a lost reliable message every 250 ms, so a shorter timeout can close the link before a lost notice is resent. To give one kick a different timeout, use an overload that takes it:

```
coreManager.ServerManager.KickClient(connection, KickOptions.Default, timeoutMilliseconds:
 2000);
coreManager.ServerManager.KickClient(connection, new KickedNotice { Reason = "Server resta
rting." }, KickOptions.Default, timeoutMilliseconds: 2000);
connection.Kick(timeoutMilliseconds: 2000);
connection.Kick(new KickedNotice { Reason = "Server restarting." }, timeoutMilliseconds: 2
000);
```

Once marked, a kick cannot be stopped or loosened: a second kick against the same client is ignored, and the first policy and timeout stand. A client that departs on its own before the kick runs is simply dropped from it.

Reading the pending state

Once a kick is marked, `Connection` exposes:

- `IsKickPending`: true from the moment the kick is marked until the connection closes.
- `PendingKickOptions`: the policy that was applied.
- `IsUsable`: true only while there's still a point acting on this connection: no pending kick, and a connected remote or local state. This is the one check that covers a kicked or already-disconnected peer in a single condition.

Relation to the violation pipeline

The violation pipeline's own `Kick` action calls `ServerManager.KickClient` too, but gates it on `IsServerStarted` first — a violation that settles on `Kick` while running as a client (for example, an impossible acknowledgment from its own server) logs that the kick isn't enforced rather than kicking the server's connection out from under itself. That gate belongs to the violation pipeline, not to `KickClient`, which enforces no such check on its own. See the violations page for how actions are decided.

The manager map

The eleven managers

`CoreManager` owns eleven managers as public fields, constructed once and held for the life of the session. Each one is documented in full on its own pages elsewhere; this page only says which manager owns which concern and how to reach it.

Manager	Owns
`NetworkLoopManager`	The tick loop: tick rate and the steps that drive a tick forward.
`TransportManager`	The transports a session sends and receives over.
`ServerManager`	Server-side session state.
`ClientManager`	Client-side session state.
`SystemManager`	Networked systems: spawning, ownership, and their lifecycle.
`InterestManager`	What each peer is allowed to see.
`SceneManager`	Scene load, stacking, and per-scene state.
`PacketManager`	Packet framing and dispatch.
`MessageManager`	Messages sent between peers.
`RpcManager`	Remote procedure calls.
`ViolationManager`	Detecting and acting on protocol violations.

The two Pro-only managers

`BundleManager` and `WorldPersistenceManager` are not part of the base `CoreManager`. They are added by Pro partial classes - `CoreManager.Bundles.Pro.cs` and `CoreManager.Persistence.Pro.cs` - each of which declares the field and fills in a partial hook (`AddBundleManager`, `AddWorldPersistenceManager`) that the base constructor calls unconditionally. In a Free build neither partial exists, the hooks compile away to nothing, and `CoreManager` simply has no `BundleManager` or `WorldPersistenceManager` field at all - not null, absent.

`BundleManager` owns content delivered from outside the build (Addressables, a patcher, a CDN). `WorldPersistenceManager` owns writing a world to disk and building it back.

Reaching a manager

From C#, every manager is a public field directly on `CoreManager`: `coreManager.TransportManager`, `coreManager.SystemManager`, and so on, including the two Pro fields when they're present.

From Unity, nine of the eleven have a matching `Unity*Manager` component with a `Nucleus*Manager` property that returns the same instance:

Component	Property
`UnityNetworkLoopManager`	`NucleusNetworkLoopManager`

`UnityTransportManager`	`NucleusTransportManager`
`UnityServerManager`	`NucleusServerManager`
`UnityClientManager`	`NucleusClientManager`
`UnitySystemManager`	`NucleusSystemManager`
`UnityInterestManager`	`NucleusInterestManager`
`UnitySceneManager`	`NucleusSceneManager`
`UnityPacketManager`	`NucleusPacketManager`
`UnityMessageManager`	`NucleusMessageManager`

Two exceptions have no Unity-side component of their own: `RpcManager` and `ViolationManager`. Reach either one through `UnityCoreManager.CoreManager` (the component that owns the session's `CoreManager`) or through `NucleusUnity.BoundCoreManager`, then take the field as in C#.

The tenth Unity component `EnsureManagers` adds, `UnityPhysicsManager`, is the odd one out in the other direction: it fronts no core manager at all. It's an inspector-side seat for physics execution settings that a physics coordinator reads directly, not a wrapper around one of the eleven fields above.

Every setting lives on the manager that owns it

There is no single settings type for a Nucleus session. Every knob - tick rate, a transport's connection settings, interest rules, scene load behavior - is a field on the manager that owns that concern. To change something, find its manager first.

Managers are instance-scoped

None of the eleven (or thirteen, with Pro) managers are static. Each belongs to one `CoreManager`, and a `CoreManager` is just an object - a process can construct more than one, each with its own independent set of managers. This is what lets a single process host a server and a client in the same run, or run several test peers side by side, without their state colliding.

The network loop steps

Eleven steps run in a fixed order

Nucleus never lets game code touch networked state at an arbitrary point in a frame. Every frame walks the loop's named steps in one fixed order, and a frame that runs a tick walks all eleven:

`EarlyVariableUpdate`, `EarlyStateUpdate`, `LateStateUpdate`, `Reconcile`, `EarlyFixedUpdate`, `LateFixedUpdate`, `VariableUpdate`, `EarlySerialize`, `LateSerialize`, `TickAdvance`, `LateVariableUpdate`.

Each is one flag of the `NetworkLoopSteps` enum. A callback for a step only ever receives that one flag; combinations are for declaring interest across several steps, never for a single dispatch.

The steps group by how often they run:

- **Variable steps run every frame.** `EarlyVariableUpdate`, `VariableUpdate` and `LateVariableUpdate` run on every frame the platform produces, whatever the frame rate is doing.
- **Tick steps run once per tick, on the frame that ticks.** `EarlyStateUpdate`, `LateStateUpdate`, `Reconcile`, `EarlySerialize`, `LateSerialize` and `TickAdvance` run at the engine's fixed tick rate. They are where replicated state is applied, reconciled and written, and where the tick advances.
- **Fixed steps run once per elapsed tick interval.** `EarlyFixedUpdate` and `LateFixedUpdate` run on the frame that ticks, once for each tick interval that has passed, so after a long frame they can run twice on one frame. A frame still runs at most one tick.

A frame that does not reach the next tick runs only its variable steps. Do not assume a tick step or a fixed step fires on every frame.

Several steps run framework work around your callback

Several steps carry framework work woven in alongside your callbacks, and whether that work runs before or after your callback matters:

- **`EarlyVariableUpdate` receives packets before your callback runs.** After your callback, received messages are dispatched to their handlers. If you need to react to something a message just delivered, do it downstream of this step, not inside it.
- **`LateStateUpdate` applies received state before your callback runs.** First `PresentedTick` catches up to `Tick`, then received state is applied, then received RPCs are dispatched, so a callback on this step sees the state and calls that arrived this tick already applied.
- **`EarlyFixedUpdate` advances a catching-up object before your callback runs.** In Pro, a newly spawned object that is still catching up advances one step first, so its step for the tick has already happened before anything simulates against it.
- **`LateSerialize` serializes changed state before your callback runs.** After your callback, the frame's messages, RPCs and kicks are sent. A write made inside this step's own callback is too late for this tick's state, but a message or call sent from it still leaves this frame.
- **`TickAdvance` ends the tick after your callback runs.** Systems whose stop was pending this tick finish stopping, clients that did not finish authenticating in time are dropped, `Tick` increments, and then systems whose start was deferred go live. It is the last step of a tick, so `LateVariableUpdate`, the frames between ticks and every message handler already read the tick their writes will ride.

The remaining steps (`EarlyStateUpdate`, `Reconcile`, `LateFixedUpdate`, `VariableUpdate`,

`EarlySerialize` and `LateVariableUpdate`) carry no framework work of their own; your callback is the only thing that runs on them.

Messages, RPCs and kicks are sent on every frame, not only on frames that tick. On a frame that does not tick, the engine still sends them at `LateSerialize`'s place in the order; that send is internal and raises no callback. A message, call or kick queued from a `TickAdvance` or `LateVariableUpdate` callback leaves on the next frame.

Where a step sits decides what your code may safely do

- **Read replicated state after it has been applied**, not before. `LateStateUpdate` and later is where incoming values are current; a read during `EarlyStateUpdate` or earlier can still see last tick's value.
- **Write before it is serialized.** `EarlySerialize` is the step for writing values you want sent this tick. `LateSerialize` serializes whatever changed before its callbacks run, so a write made in `LateSerialize` or later rides the next tick.
- **Do not expect a tick step every frame.** The tick steps and the fixed steps run at the tick rate, not the frame rate. Code that must run every frame regardless belongs on `VariableUpdate` or one of the other variable steps, and anything it moves should move by `StepDelta.NormalizedDelta` rather than the raw frame delta, so each tick sees an even distance.

Message and RPC dispatch

Two dispatch points fall directly out of this ordering: typed messages resolve on `EarlyVariableUpdate`, and system RPCs resolve on `LateStateUpdate`. The difference, and why each lands where it does, is covered in When a Message or Call Reaches Its Handler rather than repeated here.

Hooking into a step

In plain C#, implement `INetworkLoopStepCallback` and register it with the `NetworkLoopManager`:

```
public class ScoreTracker : INetworkLoopStepCallback
{
    public NetworkLoopSteps GetNetworkLoopSteps() =>
        NetworkLoopSteps.LateStateUpdate | NetworkLoopSteps.EarlySerialize;

    public void OnNetworkLoopStep(NetworkLoopSteps networkLoopStep, StepDelta stepDelta)
    {
        if (networkLoopStep == NetworkLoopSteps.LateStateUpdate)
        {
            // Read applied state here.
        }
    }
}

coreManager.NetworkLoopManager.RegisterNetworkLoopStepCallbacks(scoreTracker);
```

`GetNetworkLoopSteps` can combine any number of flags; `OnNetworkLoopStep` is called once per step it declared, each time with just that one flag. Unregister with `UnregisterNetworkLoopStepCallbacks` when the object is torn down.

In Unity, a `NucleusBehaviour` exposes the same eleven steps as per-step virtuals instead: `OnEarlyVariableUpdate`, `OnEarlyStateUpdate`, `OnLateStateUpdate`, `OnReconcile`, `OnEarlyFixedUpdate`, `OnLateFixedUpdate`, `OnVariableUpdate`, `OnEarlySerialize`, `OnLateSerialize`, `OnTickAdvance`, `OnLateVariableUpdate`. Override the ones you need; the base class only registers for the steps your concrete type actually overrides. See Tick rate and the loop in Unity

for the component side of this.

Testing a step in a hot path

`NetworkLoopStepsExtensions.FastContains` checks one `NetworkLoopSteps` value for another without allocating, useful when a hot path needs to branch on the current step:

```
if (networkLoopStep.FastContains(NetworkLoopSteps.EarlyFixedUpdate))
{
    // ...
}
```

Keeping the world across a disconnect

Choosing what a dropped client does with its world

When a client's link to the server drops, `ClientManager.DisconnectResetMode` decides what happens to the world that client was holding:

```
coreManager.ClientManager.DisconnectResetMode = DisconnectResetMode.RetainReceivedWorld;
```

`DisconnectResetMode` has two values:

- **`ClearReceivedWorld`** (the default): despawns every `NetworkSystem` the server sent and releases every networked scene it placed the client in. The next session starts from nothing.
- **`RetainReceivedWorld`** : leaves every received `NetworkSystem` started and every networked scene loaded, exactly as they stood when the link dropped.

`RetainReceivedWorld` only makes sense for a peer that means to keep the world rather than discard it — one that disconnects from a host and intends to start its own server to carry on hosting what it was just shown. A retained world is still stamped with the departed server's identifiers, so it's only usable once this peer adopts it as its own (see below). Any other client that reconnects elsewhere drops everything it's holding and rebuilds from whatever the new server sends, regardless of this setting — retention only ever survives until the next link comes up.

The field itself is present and settable in every edition. In a Free build it simply does nothing: the branches that honor `RetainReceivedWorld` live in the Pro partials (`ClientManager.Adoption.Pro.cs`, `ServerManager.Adoption.Pro.cs`, `SceneManager.Adoption.Pro.cs`), and a Free client always clears its world on disconnect regardless of how the field is set.

Adopting a retained world

A peer that retained a world takes it over with `ServerManager.TryAdoptRetainedWorld()` instead of respawning everything from scratch:

```
coreManager.ServerManager.AuthorityAdopted += OnAuthorityAdopted;

if (coreManager.ServerManager.TryAdoptRetainedWorld())
    await transport.ConnectAsync(Invoker.Server);

void OnAuthorityAdopted(uint adoptedSystemCount, uint adoptedSceneCount)
{
    // Re-attach whatever this peer only did as a spectator while it was a client.
}
```

`TryAdoptRetainedWorld()` returns `true` when a world was adopted, and `false` when this peer holds nothing it was sent. It fires `AuthorityAdopted(uint adoptedSystemCount, uint adoptedSceneCount)` on success, so a game can re-attach whatever it only does for objects it owns once this peer becomes the server for them.

Call it after the link has dropped and before starting the server. `TryAdoptRetainedWorld()` refuses to run once `TransportManager.IsServerStarted` is true, because being the server changes the meaning of the interest pass, the recovery pass, and the scene and bundle request handlers from the instant the server socket connects.

Adoption keeps every identifier the world already has and raises the allocators past them, rather than renumbering. Renumbering would surface as the entire world despawning and respawning, since an

identifier is the only name an object has.

`TryAdoptRetainedWorld()` doesn't start anything itself. The caller starts the server, and — if this peer's own player is meant to keep playing — connects its own client half afterward. A pure-server successor holds the world but has no character in it.

Skipping the reconcile on reconnect

A retained world normally reconciles against the new server as an ordinary deserialize: identifiers this peer already holds apply in place, and the objects never blink. Setting `SystemManager.RespawnWorldOnReconnectEnabled` skips that reconcile entirely — the world is dropped and rebuilt from the server's own account of it instead, at the cost of a spawn per object, but without carrying over anything from a reign the server never heard of. When this is set, `RetainReceivedWorld` no longer changes how reconnecting behaves.

Telling survivors where to regroup

Retaining and adopting a world only helps if the other peers can still find the promoted host — and the moment the old host drops, there's no channel left to tell them. `SessionDirectoryMessage` is how a game tells clients that beforehand:

```
public struct SessionDirectoryMessage : IMessage
{
    public ulong SessionId;

    public const ulong NoSession = 0;
}
```

Send it over `Channel.Reliable` once a client authenticates, and to any client that joins later:

```
coreManager.ServerManager.ClientAuthenticated += connection =>
{
    connection.SendMessage(Channel.Reliable, new SessionDirectoryMessage(sessionId));
};
```

`SessionId` names the session with whatever directory the game rendezvouses through, or `SessionDirectoryMessage.NoSession` when the host isn't registered with one. Nucleus carries the identifier alone and defines no address, service, or protocol — what it means, and which directory answers to it, is entirely up to the game.

Treat `SessionId` as a secret shared with the players and nobody else. A directory has no way of telling a peer that was told this identifier from one that wasn't, so whoever holds it can generally take the session over. That's the point of it, and also the whole of its security.

This page covers the world's state, not reconnecting the peers

This page covers keeping and adopting the world's *state*. Actually getting the survivors reconnected to the promoted host — relay host migration and the services it needs — is covered under transports and hosting, not repeated here.

Server, client and host roles

Using Unity? See Starting and stopping a session in Unity.

Invoker

`Invoker` names which network role an operation acts for: `Client` or `Server`. It does not assert that the role is started - it is a parameter, not a status check. A single process can run both roles by starting server and client separately; that combination is host mode.

```
public enum Invoker : byte
{
    Client = 0,
    Server = 1,
}
```

Starting a role

A role starts on a `Transport`, after the transport has been constructed, configured, and added to the `TransportManager`. See the transports pages for construction and configuration.

```
await transportManager.TryAddTransportAsync(transport);

await transport.ConnectAsync(Invoker.Server);
await transport.ConnectAsync(Invoker.Client);
```

`ConnectAsync(Invoker)` begins a connection using the transport's current configuration: a `Client` invoker connects the local client to the server, a `Server` invoker begins listening for connections. Calling both starts a host. The call returns a `ConnectionStateChangeResult`.

Reading the roles

`TransportManager` exposes three booleans for the local peer's current state:

Member	True when
`IsServerStarted`	The local server connection is connected.
`IsClientStarted`	The local client connection is connected **and authenticated**.
`IsHostStarted`	`IsServerStarted` and `IsClientStarted` are both true.

`IsClientStarted` does not flip on socket connect - it waits for authentication, so gameplay code never sees a client as started before the server has accepted it.

`NetworkSystem` re-surfaces the same three members so gameplay code can branch without reaching through to the transport layer:

```
if (networkSystem.IsServerStarted)
{
    // Authority-only logic.
}
```

IsAuthority was removed because it conflated two questions

`IsAuthority` was removed. It conflated two different questions, and the roles above answer them separately:

- **"Is this peer the server?"** - `IsServerStarted`. The server is always the server.
- **"Does this peer control this system?"** - `NetworkSystem.IsController(ControllerType controllerType)`. This is a method, not a bool property, because control is checked against a specific role:

```
if (networkSystem.IsController(ControllerType.Client))
{
    // This peer is the client controlling this system.
}
```

`ControllerType` is a flags enum (`None`, `Server`, `Client`, `AnyController`), so a check can ask "am I the server-side controller", "am I the controlling client", or "am I the controller at all" with `AnyController`.

The loud guards `EnsureIsController(ControllerType)` and `EnsureIsStarted(Invoker)` live on the Unity track's `NucleusBehaviourBase`, not on the core API - see the Unity page for them.

Stopping a role

`DisconnectLocalConnectionAsync(Invoker)` disconnects the local connection for the given role and closes its open sockets. `ShutdownAsync` stops all of a transport's local connections at once, for teardown.

```
await transport.DisconnectLocalConnectionAsync(Invoker.Client);

await transport.ShutdownAsync();
```

Neither call removes the transport from the `TransportManager` - the transport itself stays added, ready to `ConnectAsync` again, until it is explicitly removed.

ConnectionStateChangeResult

A role change reports its outcome as a `ConnectionStateChangeResult`:

```
public enum ConnectionStateChangeResult : byte
{
    Success,
    AlreadyInState,
    InvalidConnection,
    InvalidSocketState,
    UnspecifiedError,
}
```

`ConnectAsync` and `DisconnectLocalConnectionAsync` both return it directly; `ShutdownAsync` does not, since it tears down every local connection on the transport rather than reporting on one. The same information also arrives as an event: `TransportManager.ConnectionLocalStateChanged` fires with a `ConnectionStateChange` (the `Invoker`, the `Connection`, and whether the change was for the local or remote state), so code that isn't the caller of `ConnectAsync` can still react to a role starting or stopping.

Kick and refusal notices

The reason is a message you define

When the server kicks or refuses a client, the engine attaches no reason of its own. If the client should know why, the server sends a notice: an ordinary message type of your own, passed to the kick or the refusal, which the engine sends on the reliable channel ahead of the disconnect.

The client tells a kick or a refusal apart from any other disconnect by the notice's message type. Give each kind of reason you want to tell apart a type of its own, or carry the detail in a field:

```
[NetworkType]
public struct ServerFullNotice : IMessage
{
    public string Text;
}

[NetworkType]
public struct KickedNotice : IMessage
{
    public string Text;
}
```

A notice type needs no authentication exemption. A message from the server reaches a client that is not yet authenticated whatever its `IsAuthenticationRequired` says, so a refusal's notice arrives even though the client never got as far as being approved. Leave `IsAuthenticationRequired` at its default.

Sending one from the server

Pass the notice to the call that ends the client's session:

```
// Refusing a connecting client from an authenticator.
coreManager.ServerManager.DenyClient(connection, new ServerFullNotice { Text = "Server is
full." });

// Kicking a client already in the session.
connection.Kick(new KickedNotice { Text = "Idle for too long." });
```

Both send the notice first and then mark the kick, which is the only order that works: a kick blocks every message queued after it. Kicking and disconnecting a client covers the kick policy and the timeout, and Authenticating clients covers when to refuse. The plain `DenyClient(connection)` and `Kick()` send nothing, which is the right default for a failed authentication: a refused peer is owed no explanation, and one can help an attacker.

Receiving one on the client

The client receives a notice like any other message, through a handler registered with `MessageManager.RegisterMessageHandler<T0>`. Register the handlers before connecting, so a refusal that arrives during the handshake is not missed:

```
public sealed class DisconnectReasons
{
    public string LastReason { get; private set; }

    public DisconnectReasons(MessageManager messageManager)
    {
        messageManager.RegisterMessageHandler<ServerFullNotice>(OnServerFull);
        messageManager.RegisterMessageHandler<KickedNotice>(OnKicked);
    }

    private void OnServerFull(in MessageContext messageContext, ServerFullNotice notice) =
> LastReason = notice.Text;

    private void OnKicked(in MessageContext messageContext, KickedNotice notice) => LastRe
ason = notice.Text;
}
```

The notice arrives before the disconnect

A close the server makes is held on the client until the messages that arrived with it have been dispatched, so a kick's or a refusal's notice reaches its handler before the client's own disconnect is raised. A disconnect handler can therefore read whatever the notice handler recorded, such as `LastReason` above, and show it. When no notice handler ran, the disconnect had some other cause: the server sent no reason, the link timed out, or the server shut down.

Session events

> **Using Unity?** See Reacting to session events in Unity.

A session turns on four moments, spread across three managers. Each answers a different question, and each is raised at a specific point in a specific code path, not on a generic "something changed" tick.

ServerManager.ClientAuthenticated

```
public event ClientAuthenticatedHandler? ClientAuthenticated;
public delegate void ClientAuthenticatedHandler(Connection connection);
```

Raised on the server the moment a client is admitted, from inside `ApproveClient`. By the time it fires the connection is already marked authenticated and its `Connection.Identity` is stamped, so a handler can read both.

It is raised **before** the `AuthenticationResponse` is sent to the client and **before** the roster add is queued. Two things follow from that ordering:

- The client itself does not know it has been accepted yet (see `LocalClientAuthenticated` below), so nothing sent from here reaches it ahead of its own identity.
- Other clients have not been told this connection exists yet, because the roster add is queued after this event returns.

It is also not necessarily true yet that interest has registered the new connection as an observer of anything. `InterestManager` subscribes its own handler to `ClientAuthenticated` in `ManagersInstantiated`, so it runs whenever the framework happens to invoke subscribers, and it only registers the connection immediately when `Connection.CanReceiveState` is already true. A fresh remote client whose round trip has not been measured yet is instead queued into `InterestManager`'s internal wait set and registered a round trip later, once the transport's round-trip-time discovery fires. A host's own client is typically resolved immediately.

Either way, per-player setup done inside this handler is still tick-aligned with the roster: the roster add this connection is owed, and the roster add announcing it to everyone else, both ride the same tick's state stream regardless of when interest finishes registering it as an observer.

ServerManager.ClientDisconnecting

```
public event ClientDisconnectingHandler? ClientDisconnecting;
public delegate void ClientDisconnectingHandler(Connection connection);
```

Raised on the server for a client that has dropped, while the objects it controlled are still readable through it. It is raised from the release path — specifically from the same handler that is about to release the connection's controlled systems — immediately before that release runs, which is why a lookup of what the connection controlled still answers correctly here and not a line later.

Do not subscribe to a lower-level transport disconnect event expecting the same guarantee. Handler order between independent subscribers of a raw transport event is not something to rely on; `ClientDisconnecting` exists specifically because release itself is one of those subscribers, and this event is the one call site that can promise "before release" as a contract.

ClientManager.LocalClientAuthenticated

```
public event LocalClientAuthenticatedHandler? LocalClientAuthenticated;
public delegate void LocalClientAuthenticatedHandler(Connection localClientConnection);
```

The only moment a peer learns who it is. `Connection.Id` on the local client connection is `Connection.UnsetId` from the moment the socket connects until this fires. It is raised from inside the handler for the incoming `AuthenticationResponse` message, after the local connection's `Id` has been set to the server-assigned value and after it has been marked authenticated — both are true by the time a handler runs.

Raised identically on a host: the host's own client half receives the same response and adopts the same server-assigned identity, so a handler does not need to special-case which role it is running under.

TransportManager.PeerConnectionDiscovered / PeerConnectionDropped

```
public event PeerConnectionChangedHandler? PeerConnectionDiscovered;
public event PeerConnectionChangedHandler? PeerConnectionDropped;
public delegate void PeerConnectionChangedHandler(Connection connection);
```

These are about *other* peers, not this connection's own session. They fire once a `Connection` for that other peer becomes available or is about to go away, and they mean something different depending on role:

- On a client, `PeerConnectionDiscovered` fires from `TryAddPeerConnection` when the server reports a new peer. The `Connection` handed out is a stand-in — an identity with no live link behind it (`Connection.IsPeerStandIn` is true).
- On a host, the same event fires from `RaiseHostPeerConnectionDiscovered`, called once the server side of `ApproveClient` has already admitted the peer. The `Connection` is the server's own entry for that peer, so `IsPeerStandIn` is false. It is never raised for the host's own client, since no client is ever told about itself this way.

`PeerConnectionDropped` mirrors this exactly, firing from `TryRemovePeerConnection` on a client or `RaiseHostPeerConnectionDropped` on a host, in both cases while the entry is still held. Anything that referenced a stand-in must drop that reference inside this handler: the instance returns to its pool immediately afterward, and the next peer to connect can be handed that same object.

Unlike `ClientAuthenticated` and `ClientDisconnecting`, these two never fire for this peer's own session — they only ever describe someone else.

Ordering on a host

A host runs the server and client halves of the session in one process, and that collapses steps that are two separate machines' worth of events on a dedicated server plus remote client.

For the host's own client connecting, `ApproveClient` raises `ClientAuthenticated` first, then sends the `AuthenticationResponse`. Because the host's connection to itself is marked as a loopback pair before that send, the response is delivered in-process immediately rather than crossing a socket — which means `LocalClientAuthenticated` fires synchronously inside that same send call, immediately after `ClientAuthenticated`, rather than a round trip later.

For a remote peer authenticating against a host, `ApproveClient` also ends by calling `RaiseHostPeerConnectionDiscovered` for that same connection. So `ClientAuthenticated` and `PeerConnectionDiscovered` both fire for that one peer, in that order, out of the same call — something a pure client never sees for itself, since a client only learns about a peer once a roster entry lands.

Disconnection follows the mirror shape: `ClientDisconnecting` is raised from the release path before a

connection's controlled systems are released, and `PeerConnectionDropped` is raised (via `RaiseHostPeerConnectionDropped`, host only, and never for the host's own client) out of the same connection-state-changed handling that release runs from.

Each event allows some work and asks you to defer the rest

Event	Safe here	Defer instead
`ClientAuthenticated`	Read `Connection.Identity`; do per-connection setup — it is still tick-aligned with the roster add queued right after.	Anything that assumes the connection is already a registered observer; that may not be true until a round trip later.
`ClientDisconnecting`	Look up what this connection controlled or owned; nothing has released yet.	Anything that depends on the connection's controlled systems already being gone — release runs immediately after this returns.
`LocalClientAuthenticated`	Read this peer's own `Connection.Id` for the first time; anything scoped to "who am I".	Assuming the roster of other peers has already arrived; it rides the state stream separately.
`PeerConnectionDiscovered`	Store a reference to the peer's `Connection` for later use.	—
`PeerConnectionDropped`	Drop every stored reference to this peer's `Connection`.	Holding onto it past this handler — the instance is pooled and reused right after.

Shutting a session down

> **Using Unity?** See Play mode, domain reload and session lifetime.

`CoreManager.Deinitialize()` tears a session down: it stops the network loop, shuts every transport down, and releases every manager the `CoreManager` owns. Call it once, from whatever code owns the session's lifetime, when that session is done.

```
coreManager.Deinitialize();
```

The unwind order

`Deinitialize()` does three things, in this order, and the order is the point:

Stop the loop provider. The network loop's step provider is released first, because the default provider steps the loop from a thread-pool thread, and every manager released below mutates collections a step walks. Nothing else can safely happen while the loop is still stepping.

Shut the transports down. Every transport is shut down next. Shutting a socket down drives the disconnect path through handlers spread across the managers, and every one of those managers is still whole at this point. Doing this later, partway through the manager unwind, would run those handlers against managers that no longer exist.

Release the managers in reverse of construction order. A manager built later resolved and registered onto managers built earlier (handlers on the message manager, a spawn gate on the interest manager, callbacks on the network loop manager). Releasing the newest first means nothing is released while something built after it still holds a reference to it.

Both the loop-provider release and the transport shutdown are idempotent on their own, so each manager's `Deinitialize` still reads as the one place its resource is released.

Idempotence

`IsDeinitialized` is set to `true` before any of the unwind runs, not after. A second call to `Deinitialize()` returns immediately. This absorbs the two paths that would otherwise race or double-run: a host that already tore the session down explicitly, and a destroy callback (a platform's `OnDestroy`, for example) that arrives afterward. Neither path needs to check `IsDeinitialized` itself first, though it can:

```
if (!coreManager.IsDeinitialized)
    coreManager.Deinitialize();
```

The Deinitialized event

```
public event DeinitializedHandler? Deinitialized;
public delegate void DeinitializedHandler(CoreManager coreManager);
```

`Deinitialized` is raised last, after every manager has been released and after the `Instance` slot has already moved on to the next live `CoreManager` (or to `null`). Despite that, the `CoreManager` and its manager fields are still readable during the raise. A consumer that keyed something per loop can use this instant to find its key through `NetworkLoopManager` and drop its own entry, without racing whatever the `Instance` slot now points at.

```
coreManager.Deinitialized += static torndDown =>
{
    // torndDown.NetworkLoopManager is still readable here.
};
```

A torn-down CoreManager is not reusable

Once `IsDeinitialized` is `true`, that instance is done. There is no re-initialize path. A process that wants to run another session constructs a new `CoreManager`. This applies whether the process is exiting or looping back to run the next session in place: the next session's `CoreManager` is a fresh object, not the old one reset.

ResetRegistry for a host running sessions back to back

`CoreManager` keeps a static registry of every live manager, which is what `Instance` resolves from and what the slot is handed on through when a manager leaves. The registry holds strong references: a live manager must stay resolvable as a successor for the `Instance` slot, so nothing in it is ever collected while it is still registered.

A process that runs one session after another in the same run — rather than one `CoreManager` per process — calls `CoreManager.ResetRegistry()` at the very start of the next run, before anything can construct a new `CoreManager`. It clears the live-manager registry and clears `Instance` to `null`, so the new run cannot inherit managers left over from the run before it.

```
CoreManager.ResetRegistry();
```

`ResetRegistry()` drops references rather than tearing anything down. It assumes whatever is still registered at that point belongs to a run that already ended and may be holding onto engine objects the host already destroyed. A session that wants an orderly unwind still calls `Deinitialize()` while it is live, before the next run resets the registry.

Shutting transports down without a full teardown

Tearing down transports does not require tearing down the whole `CoreManager`. `TransportManager` exposes the same shutdown path used internally, scoped to one transport:

```
await coreManager.TransportManager.TryRemoveTransport(transport);
```

`TryRemoveTransport` removes the transport from the manager's list and awaits `Transport.ShutdownAsync()` on it, then rescans the local server and client connections. It returns `false` if the transport was not in the list to begin with. This is the right call for stopping one transport — dropping a socket a session no longer needs — while the rest of the session keeps running.

Threading and lifetime

The default provider runs on the thread pool

`CoreManager` drives its network loop through an `INetworkLoopStepProvider`. Pass none and it falls back to `SystemNetworkLoopStepProvider`, which runs a `System.Timers.Timer`. A `System.Timers.Timer` raises its `Elapsed` event on a thread-pool thread, not the thread that constructed the `CoreManager`.

This is the first thing a plain .NET host gets wrong. Every loop step callback (`INetworkLoopStepCallback.OnNetworkLoopStep`), every message handler, every RPC handler, and every `NetworkMember<T0>` changed callback dispatches from inside a network loop step. With the default provider, that means all of it runs on a pool thread by default. If your own code assumes it is running on the thread that built the `CoreManager` - touching a UI collection, a non-thread-safe cache, anything not built for concurrent access - that assumption is wrong unless you supplied your own provider.

One loop drives one manager

`NetworkLoopManager.InvokeNetworkLoopStep` claims the step with an `Interlocked.CompareExchange` before doing any work. If a second thread calls in while a step is already executing on a different thread, that call is refused outright: the step's callbacks do not run, and the refusal is logged as an error (throttled to at most once every sixty seconds so a stuck second driver does not flood the log). The step is not queued or retried; it simply costs that tick's work for whatever registered on it.

A re-entrant call from *inside* a step, on the thread that already holds it, is not a second driver and runs normally.

The refusal exists because two threads walking or mutating the same loop-owned collection at once is exactly the kind of fault that does not surface where it happens: a modified-collection exception out of a receive drain, a null entry from a torn callback lookup, a pooled `Writer` handed to two owners at once. Refusing the second call keeps the corruption from happening instead of trying to detect it afterward. The usual cause of seeing this in your own log is a `CoreManager` that was never given a step provider, leaving the default timer stepping it from the thread pool while something else - a manual step, a second timer - also drives it.

Register your provider at construction, not after

`CoreManager`'s constructor takes an optional `INetworkLoopStepProvider`:

```
public CoreManager(uint tickRate = NetworkLoopManager.DefaultTickRate, INetworkLoopStepPro
vider? networkLoopStepProvider = null)
```

If you pass `null`, the constructor starts the default `SystemNetworkLoopStepProvider` as its last step, once every manager has finished instantiating. By the time the constructor returns, the loop is already running on a pool thread.

Any registration you make after that - a manual `RegisterNetworkLoopStepCallbacks` call, wiring up your own systems, anything that touches loop-owned state from your own construction code - is now racing a loop that is already stepping. There is no window after construction where the loop is guaranteed idle.

In a console host, name your own `INetworkLoopStepProvider` in the `CoreManager` constructor call instead of accepting the default. Driving the loop yourself (or from a provider whose timing you control) means nothing steps until you say so, so your own post-construction setup runs before the loop can touch anything.

The DEBUG-only guards

Two internal checks compiled only into a Debug build of `Nucleus.dll` log an error when loop-owned state is touched while a step is executing on a *different* thread: one covers outgoing message sends, the other the loop's own collections. They do nothing in Release; a Debug run is what surfaces a threading mistake that a Release run silently gets away with. If you are chasing a corrupted collection or a torn write that never reproduces in Debug, that alone is a clue: run the reproduction in Debug and check the log for these.

Isolation between CoreManagers

The step-callback registry belongs to each `NetworkLoopManager` instance; it is not static state. Two `CoreManager` instances in the same process - a host pair, a bridge test - each own their own loop and their own callback registrations. One manager's registered callbacks are never dispatched by the other's steps.

Async work and the loop

`ISceneLoader.LoadSceneAsync` and `UnloadSceneAsync` return `Task<bool>` and may await freely - a download, a disk read, an engine operation - without minding which thread the continuation resumes on. Nothing in the loop blocks waiting for them.

The engine brings an awaited outcome back onto the loop thread itself. Once your loader's task completes, the engine's follow-up is queued for the loop rather than run inline, so it always resumes on the loop regardless of which thread the awaited operation completed on. Queued follow-ups run during `NetworkLoopSteps.EarlyVariableUpdate`, after that tick's messages have dispatched. A scene operation that finishes on a background thread therefore has its loop-owned follow-up (registering objects, tearing down on release) land at that same controlled point in the tick every time, not whenever the background work happens to finish.

`IBundleLoader` (Pro-only) is the equivalent seam for bundle loads: `LoadBundleAsync` and `UnloadBundleAsync` are likewise `Task<bool>`-returning and the engine does not care which thread they complete on, for the same reason.

The Unity integration drives the loop from the main thread instead

Unity drives the loop from the main thread instead of the default pool-thread timer, so callbacks, handlers and scene/bundle continuations land on Unity's main thread the way the rest of your MonoBehaviour code expects. See the Unity integration's own network loop page for how its provider is wired up.

Ticks, tick rate and time

Using Unity? See Tick rate and the loop in Unity.

Tick rate

`NetworkLoopManager.TickRate` is the number of ticks the loop runs each second. It is set once, from the `tickRate` argument passed to the `CoreManager` constructor, and has no setter afterward:

```
public uint TickRate { get; }
```

If the caller names no rate, `CoreManager`'s constructor defaults to `NetworkLoopManager.DefaultTickRate` (30). A requested rate outside `NetworkLoopManager.MinimumTickRate` (5) through `NetworkLoopManager.MaximumTickRate` (128) is clamped to the nearer bound and logged as a warning rather than rejected.

Every rate-derived value in the engine, the loop provider's tick interval, the state retention window, each `NetworkMember`'s send interval in ticks, is baked once from this value while the managers are constructed. There is no supported way to change the tick rate mid-session; a new rate means a new `CoreManager`.

```
CoreManager coreManager = new(tickRate: 60);
uint rate = coreManager.NetworkLoopManager.TickRate; // 60
```

The tick counter

`NetworkLoopManager.Tick` is the current local tick, incremented once per tick at the very end of the `TickAdvance` step, after its callbacks have run:

```
public uint Tick { get; private set; } = FirstTick;
```

Because it advances as the last thing a tick does, `LateVariableUpdate`, the frames between ticks and every message handler already read the tick their writes will ride.

It starts at `FirstTick` (1), never at zero. `UnsetTick` is 0, and the framework relies on "a live tick is never the unset value" wherever a tick doubles as a presence flag, such as a `NetworkMember`'s self-clearing writer stamp.

`Tick` is this peer's own local counter, not a shared wall clock. Two connected peers do not share a tick number: a client's `Tick` and the server's `Tick` are two independently incrementing counters that happen to run at the same rate. To estimate what tick a remote peer is on, read `Connection.TryGetServerTickEstimate` instead:

```
public bool TryGetServerTickEstimate(uint localTick, out uint serverTickEstimate)
```

It derives the estimate from the newest state packet received from that connection, and returns `false` until at least one has arrived.

Subtick percentage

`SubtickPercentage` is the fraction, 0 through 1, of the current tick that has elapsed:

```
public float SubtickPercentage { get; private set; }
public void SetSubtickPercentage(float percentage)
```

It is set through `SetSubtickPercentage`, which clamps the value into the 0-1 range. `NetworkLoopStepDriver` publishes it on every frame that runs a variable update, just before `VariableUpdate`, so the default `SystemNetworkLoopStepProvider`, the Unity integration's provider,

and any provider you build on the driver all keep it current; do not set it yourself from such a provider. Only a provider that invokes the steps without the driver leaves it at zero, its default, which is a safe value: interpolation reading it resolves to the previous value rather than an undefined intermediate one.

StepDelta

Every loop step callback receives a `StepDelta`, carrying four millisecond fields:

```
public struct StepDelta
{
    public long Delta;
    public long FixedDelta;
    public long TimeSinceLastFixedUpdate;
    public double NormalizedDelta;
}
```

- `Delta` — milliseconds since the step last ran. Use it for variable-timing work.
- `FixedDelta` — the fixed delta for the step, when one applies; zero otherwise. Use it for work that needs consistent timing, such as physics or anything keyed to the tick rate.
- `TimeSinceLastFixedUpdate` — milliseconds since the last fixed update completed. Not modified until that fixed update finishes.
- `NormalizedDelta` is the frame's delta adjusted so the frames between two ticks add up to exactly one tick interval. Use it for anything you move every frame, as the next section explains. `NetworkLoopStepDriver` sets it, so it is zero when you invoke the steps yourself.

Move per-frame work by the normalized delta

Moving an object by the raw frame delta is the most common way to spend bandwidth without noticing. Frames never line up with ticks, so an object moved by each frame's own delta covers a different distance in every tick, and an object whose per-tick movement keeps changing costs more to replicate than one moving evenly.

`NormalizedDelta` removes that unevenness while keeping the object moving every frame. Each frame gets its delta plus anything held back from the frame before. On a frame that ticks, whatever the frames since the last tick add up to beyond one tick interval is held back and added onto the next frame, so no time is lost and the object's speed is unchanged. A single frame longer than a tick interval runs its whole delta plus anything held back.

Read it on `VariableUpdate`, and convert it to seconds yourself:

```
public void OnNetworkLoopStep(NetworkLoopSteps networkLoopStep, StepDelta stepDelta)
{
    float seconds = (float)(stepDelta.NormalizedDelta / 1000.0);
    position += velocity * seconds;
}
```

In the belted asteroid and roaming character benchmarks, moving every frame by the raw frame delta cost between three and six times as much as moving once per tick. Moving every frame by `NormalizedDelta` brought it back to within a tenth of the once-per-tick cost.

Running your own code on the loop

Implement `INetworkLoopStepCallback` to have code invoked on chosen loop steps:

```
public interface INetworkLoopStepCallback
{
    public NetworkLoopSteps GetNetworkLoopSteps() => NetworkLoopSteps.None;

    public void OnNetworkLoopStep(NetworkLoopSteps networkLoopStep, StepDelta stepDelta);
}
```

`GetNetworkLoopSteps` declares which steps the callback wants; `OnNetworkLoopStep` is invoked on each of them, receiving the step and its `StepDelta`.

Register and unregister the callback against the `NetworkLoopManager`:

```
public void RegisterNetworkLoopStepCallbacks(INetworkLoopStepCallback networkLoopStepCallb
ack)
public void UnregisterNetworkLoopStepCallbacks(INetworkLoopStepCallback networkLoopStepCal
lback)
```

A callback registered from inside another callback's step is queued and takes effect from the following step; the current step has already decided who it dispatches to. The same applies to unregistering: a callback that departs mid-step is still invoked for the remainder of that step.

To find an already-registered callback of a known type back through the loop, rather than keeping a separate reference to it:

```
public bool TryGetFirstNetworkLoopStepCallback<T0>(NetworkLoopSteps networkLoopStep, out T
0 networkLoopStepCallback)
    where T0 : class, INetworkLoopStepCallback
```

Pass a single step, not a combination; callbacks are held per flag, so a combined value finds nothing.

Error handling

A callback that throws does not stop the step. The exception is caught per callback, logged, and the step continues to its remaining callbacks:

```
The loop callback [<CallbackType>] threw during [<Step>]; the step went on to its remainin
g callbacks: [<exception>].
```

One misbehaving callback therefore cannot block the framework work, or any other registered callback, on the same step.

Common errors and what causes them

Paste the text of a log line into this page's index to find what produced it and what to change. Every line below is quoted verbatim from the engine.

Index

- ["ran on thread [X] while a network loop step is executing on thread [Y]..."](#off-loop-writer-work) — off-loop writer work
- ["touched the loop-owned collection [...] on thread [X] while a network loop step is executing on thread [Y]..."](#off-loop-collection-access) — off-loop collection access
- ["Step [X] was turned away on thread [...] because this manager is already executing a step on thread [...]..."](#second-driver) — a second driver stepping the loop
- ["Violation [X] from Connection [...]"](#violation-lines) — a violation raised against a peer
- ["A Recovery entry from Connection [...] references unknown NetworkSystem Id [...]; that entry is skipped."](#recovery-and-retention) — recovery named a system the local world no longer has
- ["...acknowledges [an impossible tick]..." (InvalidStateAckViolation)](#recovery-and-retention) — an ack above the local tick
- "...retention window was exceeded..." (RetentionExceededViolation) — acked tick fell out of the retention window
- ["A StatePacket subpacket from Connection [...] declares [X] body bits but only [Y] remain; discarding the rest of the tick."](#serializer-and-framing-faults) — declared-body-bits mismatch
- ["State subpacket [X] from Connection [...] left the reader at [...] bits but its framed boundary was [...]..."](#serializer-and-framing-faults) — framed-boundary mismatch
- ["Connection [...] sent unrecognised PacketType [X]; the packet is discarded."](#version-skew) — version skew, not a kick

Off-loop writer work

`"[Caller<MessageType>] ran on thread [X] while a network loop step is executing on thread [Y]. This work must not run concurrently with the loop; it races the outbound WriterPool and can corrupt sent packets."`

Cause. Something serialized or accumulated a message from a thread other than the one currently running a network loop step. The outbound `WriterPool` is rented from the loop during serialization, so a concurrent caller can be handed a `Writer` that is already in use.

Fix. Move the work onto the loop. If it's resuming after an `await`, route it through the loop rather than sending inline from wherever the continuation happened to land.

This guard is compiled only in `DEBUG`. A Release build has no such check and fails silently: the race still happens, it just never gets a log line naming it. Reproduce and diagnose the fault in Debug, then confirm the fix in Release.

Off-loop collection access

`"[Caller] touched the loop-owned collection [Name] on thread [X] while a network loop step is executing on thread [Y]. This work must run on the network loop thread; touching the collection off the loop races the loop and can silently corrupt it."`

Cause. An async continuation resumed off the loop thread and then touched a collection the loop owns without locking — chiefly the system routing table or the server's open-scene table. The loop mutates and enumerates these with no lock of its own, so a foreign thread touching them races the loop's reads and writes and can tear the collection or drop an entry.

Fix. Switch back onto the loop before touching loop-owned state, rather than acting from wherever the continuation resumed.

Also `DEBUG`-only. Same consequence in Release: the guard disappears, the race doesn't.

Two threads are driving one manager

`"Step [X] was turned away on thread [...] because this manager is already executing a step on thread [...]. One loop must drive one manager, and a second driver races every collection the loop owns. A manager whose owner never supplied a step provider, leaving the default one driving the loop from the thread pool, is the usual cause. At least [N] steps have been turned away since the previous report."`

Cause. Two threads are stepping the same manager's loop. The usual case is an owner that never supplied its own `INetworkLoopStepProvider`, so the default background-timer provider is driving the loop from the thread pool at the same time as whatever else is stepping it.

Fix. Make sure exactly one driver steps the manager — supply the owner's own step provider (a Unity integration's loop provider, a test harness driving ticks by hand) rather than letting two sources call in.

This line is throttled to at most once per minute per manager, because the incident it was built for produced hundreds of thousands of identical lines in minutes. Every turned-away step is still counted even when not logged, and the reported count (`At least [N] steps have been turned away`) is what tells you the real severity — a report you see once with a count of 1 is very different from one with a count in the thousands, even though the line text looks the same either way.

Violation lines

`"Violation [TypeName] from Connection [...]"`, sometimes followed by `"...settled on a kick, but this peer is not a server; the kick is not enforced."` or `"Disconnecting Connection [...] for violation [TypeName]."`

A violation writes one of these lines only when its settled action is `Log` or `Kick`. Most types default to `Ignore` and write nothing, so register an `IViolationObserver` to see those. Look up `TypeName` in the violation reference for the payload fields it carries and what triggered it:

- `EmptyCollectionDeltaViolation` — a collection member delta declaring no operations; a compliant sender can't produce one.
- `InvalidStateAckViolation` — a Connection acknowledged a state tick above the local tick.
- `MalformedDataViolation`: reading data a client sent threw on the server; kicks in a release build, logs the exception in a debug build.
- `PredictedStateChangeViolation` — a controller sent state for a member that should only ride upstream as client state.
- `RecoveryUnconfirmedViolation` — a served recovery went unconfirmed for the full patience window and had to be re-served.
- `RetentionExceededViolation` — an acked tick fell beyond the retention window, forcing a full-world recovery.
- `RpcFloodViolation` — more remote calls arrived in one drain than the receiver admits.

- `RpcSendPermissionViolation` — a remote call was sent without permission (often a well-behaved client racing a permission revocation).
- `UnauthorizedSpawnViolation` — a client sent a full serialize for a system Id the server doesn't hold.
- `UncontrolledStateChangeViolation` — a Connection sent state for a system it doesn't control.
- `UnexpectedBundleRequestViolation` / `UnexpectedSceneRequestViolation` — a client sent a request only the server ever sends.
- `UnsolicitedBundleReportViolation` / `UnsolicitedSceneReportViolation` — a client reported a load or unload it was never asked to make.

Each violation has a default `ViolationAction` (`Ignore` or `Kick`) and a registered handler can override it per Connection. The action only governs the Connection consequence; the offending operation itself is always rejected regardless of what the action does. Logging is separate: it rides on `ViolationContext<T0>.Log`, which the raise site or the handler can set.

Recovery and retention

`"A Recovery entry from Connection [...] references unknown NetworkSystem Id [X]; that entry is skipped."`

The sender believed the peer still held a system that the local world no longer has. The entry is skipped and the rest of the recovery continues.

`InvalidStateAckViolation`, `RetentionExceededViolation`, and `RecoveryUnconfirmedViolation` (see the violation index above) round out recovery diagnosis. All three default to `Ignore` rather than a kick: a single raise of any of them is ordinary behavior for a peer on a bad link — heavy loss, a retention window that's too short for the peer's latency, or a recovery that got lost along with everything else. What separates a lossy link from an actual bug is repetition: an occasional raise is the cost of an imperfect network, while raises that keep recurring for the same Connection mean its latency genuinely exceeds the retention window, or something is wrong with how it acknowledges.

Serializer and framing faults

`"A StatePacket subpacket from Connection [...] declares [X] body bits but only [Y] remain; discarding the rest of the tick."`

`"State subpacket [Type] from Connection [...] left the reader at [...] bits but its framed boundary was [...] (declared [X] body bits, off by [Y]). The stream is misaligned; discarding the rest of the tick."`

Both mean a declared bit count didn't match what was actually read or what remained in the buffer. This is a serializer question, not a gameplay one — it points at a reader/writer pair that disagree about a type's shape, not at anything the game did wrong. The rest of the tick is discarded to avoid reading garbage as if it were the next subpacket.

Version skew

`"Connection [...] sent unrecognised PacketType [X]; the packet is discarded."`

The header was read, but nothing on this side knows how to interpret that `PacketType`. This is what version skew between peers looks like on the wire: an older or newer build sent a packet type the receiver has never heard of. The packet is silently discarded rather than treated as a violation — an unrecognised type on its own isn't proof of a hostile peer, only of a version mismatch, so it's not a kick.

Diagnosing desync

Work the layers in order

"It looks wrong on the other peer" is four different bugs wearing one symptom. Check them in this order, because each rung rules out the ones below it:

Is the object there at all? If the receiver never spawned it, or the object vanished when it shouldn't have, this is interest: an `IInterestCondition` culled the pair (`InterestEffect.Spawn`, resolving to `InterestMembership.Unspawned`), or a despawn genuinely happened.

Is it there but frozen? The object exists on the receiver but stopped updating a while ago. This is `InterestEffect.Stop`: the pair resolved to `InterestMembership.Stopped`, which retains the receiver's object and drops it from the delta stream rather than despawning it. It looks identical to a dead connection from the outside; it isn't.

Is it there and moving, but wrong? Position, rotation, or some other member holds a value that doesn't match the sender. This is a state question: either the value never changed on the receiver, or it changed to something incorrect.

Is it there, correct, but late? The value is right eventually, just visibly behind. This is interpolation and the retention window, not a data bug.

Don't skip ahead. A value that "looks wrong" because the object was actually culled five seconds ago and you're staring at a stale render is not a serializer bug, and chasing it as one wastes time.

Never arrived vs. arrived and wrong

Once you're at rung 3, split it again before touching a serializer:

- **A value that never changes on the receiver** is an observation or access question. Is the Connection actually registered as an observer of that `NetworkSystem`? Did an interest condition stop it (rung 2, revisit it)? Does the sender even own or control the member it's trying to change? A member that never updates usually means the write never left the sender, or never reached this receiver, not that the receiver decoded it wrong.
- **A value that changes, but to the wrong thing**, is a serializer or prediction question. The receiver is getting deltas and applying them, so the pipeline is working; the value it lands on is what's wrong. That points at how the member is being written or projected (`TransmissionMode.Divine` vs `TransmissionMode.Interval`), or, for a controlled member, at prediction/reconciliation on the controlling side.

These two failure modes live in different subsystems. Don't debug a serializer because a value never showed up, and don't debug interest because a value arrived and was simply incorrect.

Read the violation stream first

Most desyncs that are actually protocol faults - not just late or stale data - raise a violation before you'd notice anything visually. `ViolationManager` is the framework-wide pipeline for this: register a `ViolationHandler<T0>` to decide the `ViolationAction` for one violation type, or an `IViolationObserver` to hear every type raised, or subscribe the matching per-type event (for example `UncontrolledStateChangeViolationDetected`) to just watch.

The types worth knowing when chasing desync:

- `UncontrolledStateChangeViolation` - a peer sent a state change for a `NetworkSystem` it

doesn't control.

- `PredictedStateChangeViolation` - a controller sent state for a predicted member, which should only ride upstream as client state.
- `EmptyCollectionDeltaViolation` - a collection delta declared no operations, which a compliant sender can't produce; usually a crafted packet or a diverged serializer pair.
- `InvalidStateAckViolation` - a peer acknowledged a tick above anything ever sent to it.
- `RetentionExceededViolation` - a peer's acknowledged tick fell beyond the retention window and had to be re-served the whole world. A single raise is ordinary heavy loss; chronic raises mean the peer's latency exceeds `SystemManager.StateRetentionMilliseconds`, or it's deliberately withholding acknowledgments.
- `RecoveryUnconfirmedViolation` - consecutive recoveries were served to a Connection without it ever confirming one.
- `UnauthorizedSpawnViolation` - a client's full serialize named a system identifier the server doesn't hold, which would have spawned an object on the server.

If a symptom in rungs 1-3 has a matching raise, that's your subsystem, directly. **If nothing is raised at all**, the fault sits in something the violation pipeline doesn't police: a legitimate interest decision, ordinary loss inside the retention window, an interpolation lag, or a prediction mismatch that's still a well-formed packet. Absence of a violation is evidence, not a dead end.

The `host` special case

A host still serializes its own client's packets normally on the send side - nothing is skipped there. What changes is receipt: the arriving loopback copy is discarded unread rather than deserialized, because whatever it carries either duplicates what the sending half already applied in-process, or is a stale re-apply of the same object the host itself authored.

This means a bug on the receive-and-apply path - deserialization, violation checks that only run on an incoming packet, interest resolution for an arriving state, anything downstream of "a packet was read" - never runs for a host's own client at all. If a repro only shows up between two separate processes (two real clients, or a client against a dedicated server) and never on a host testing against its own client, look at the receive path first. Host loopback simply never exercises it.

Timing faults

Some symptoms point at the tick loop, not at data. The loop runs its steps in a fixed order - network reads happen early, state is read and applied at the start of a step, serialized and sent at the end of one - and a callback only sees what's already been applied by the point it runs on. Signs you're looking at a step-ordering problem rather than a data problem:

- A value read in one callback is stale compared to the same value read in a different one on the same tick - check which steps each callback runs on, not the value itself.
- A change looks like it "doesn't take" until the following tick, or takes twice - it's being written on the wrong side of a state-apply boundary, or read before the write it depends on has happened.
- Symptoms vary with frame rate or with multiple fixed updates landing inside one variable update - that's the loop's own cadence, not the network.

Fixing these means moving the callback to the correct step relative to when state applies and when it's written, not touching serialization at all.

Narrowing to a reproduction

Once you know which rung and which of the two failure modes, shrink the repro before reading a single packet:

Cut to one object and one member. A whole scene desyncing is the same bug as one member desyncing, just noisier to look at.

Turn projection off - set the member's `TransmissionMode` to `Interval`. If the bug disappears, it's in the projection path; if it doesn't, projection was never the cause.

Raise loss deliberately (a `NetworkConditions` with a `PacketLossChance`, passed to `TransportManager.SetNetworkConditions`) to see whether the symptom is loss-shaped - arrives late, arrives out of order, never arrives - versus present on every send.

Only then read a packet. By this point you know which member, which tick range, and which of "never arrived" or "arrived wrong" you're chasing, so there's an actual question to answer instead of a blind capture.

Frequently asked questions

Short answers to the questions that come up before anyone has finished reading the rest of the docs. Each one links to the page with the full explanation.

Visibility and spawning

Why did an object stop moving instead of disappearing

Interest resolves to one of a few verdicts per connection, and "stopped" is not "unspawned." A stopped object is kept by the receiver and simply sent nothing further, which reads as frozen; an unspawned object is torn down. They are different outcomes of the same evaluation, not two names for one thing. See the interest and area-of-interest pages for what drives each verdict.

Why is nothing replicating at all

Check that the Nucleus source generator is wired into the project as an analyzer, not just referenced as an ordinary assembly. Without it no serializers are produced for your networked types, so there is nothing for the wire to send or read. See the source generator setup page.

Questions about control

Why can I see my object but not drive it

Observing and controlling are separate questions. Interest decides whether a connection is sent an object's state at all; control is a separate assignment made by the server over who is allowed to drive it. Being registered as an observer never implies being the controller. See the control assignment page.

Questions about timing

Why does it work in host mode and break with two processes

A host's own client and its server share one process, and packets between them still go through the transport's loopback path. The receiving half never reads that loopback copy back: it discards the arriving packet unread rather than deserializing it, so nothing on the deserialize-and-apply path runs for a host talking to itself. Testing exclusively in host mode never exercises that path at all, which is why a bug there only shows up once a second process is involved. See the diagnosing-desync page for the isolation steps.

Host-mode surprises

Why does the host see my object differently than my client does

The same loopback exemption above applies more broadly: a host's server half never goes through its own client's receive path, so anything gated behind deserialization, including any host-side reporting of what its own client would have received, has to be resolved separately rather than inferred from a packet the host never actually reads.

Bandwidth surprises

Why is my bandwidth number unchanged after I turned a feature off

The transport's byte and bit counters are running totals, accumulated since the connection started, with no per-object or per-feature attribution. Turning a feature off changes the rate at which the totals climb, not the totals themselves, so a snapshot taken once won't show it. Sample the counters over time (per second) and compare rates before and after the change, rather than reading a single cumulative value.

Questions about Free and Pro

Why does this type not exist in my build

Free and Pro are not a runtime switch. A Pro-only source file is excluded from the compilation entirely when building Free, so a Pro-only type is a compile error in a Free project, not something that fails or falls back at runtime. If your code references a type that isn't there, you're either building Free against Pro-only API, or your project is pulling in the wrong edition's files.

Filing a bug report

What to include

State the edition: Free or Pro. The edition is a compile-time choice, not a runtime flag — a Free build is produced with `-p:NucleusEdition=Free`, which drops every `*.Pro.cs` file from the compilation, so a report that says "Free" or "Pro" is telling us which files exist in the build, not a setting that could be misread.

List the DLL set you deployed, and confirm every one of those DLLs came from the same build. A mismatched set (an old engine DLL next to a new source-generator DLL, for example) produces failures that look like engine bugs but are actually version skew. If Unity is involved, give the Unity version. If it's a plain host, give the target framework you're building against (the engine itself targets `netstandard2.1`).

Before you file

Reproduce it outside host mode first. A host runs both the server and client sides of the connection in the same process, and that dual role masks or manufactures behavior that a real client/server split does not show. If a repro only fails when hosting, it usually isn't the bug it looks like — narrow it to a genuine two-process reproduction before filing.

Reproduce it again with logging set to `LoggerLevel.Information` and capture the output. Information is verbose enough to show the sequence of events leading into the failure without needing a rebuild.

Say which violations were raised, if any. The engine reports rule breaks through `IViolation` payloads dispatched by the violation system; if your repro triggers one, name it and include its context.

Building a minimal repro

Minimal means: one networked object, one member on it, and the smallest scene (Unity) or the smallest `Main` (plain host) that reproduces the failure. Strip everything not required to trigger it — other systems, other objects, other members — before filing.

A repro that only fails in host mode is usually not the bug you think it is. Confirm it reproduces with the server and client as separate processes before treating it as real.

Attaching evidence

Include:

- The log lines, full text, not paraphrased or truncated.
- Counter readings before and after the failure, if the behavior involves a count (received/sent packets, retries, spawned objects, etc.).
- The exact tick or sequence number at which the failure occurs, if you can identify it.

Evidence in this shape is what lets a report be reproduced from the outside, rather than argued about.

Diagnostics and Troubleshooting

Pages in this category:

- Violation reference
- Violations
- What the server trusts
- Logging
- Common errors and what causes them
- Network statistics
- Round-trip time. jitter and packet loss
- Technical limitations
- Diagnosing desync
- Frequently asked questions
- Filing a bug report
- Reading serialization history (and the raw-packet test hook)

Reading serialization history (and the raw-packet test hook)

When the log has run out

Some faults only show up as "the client has the wrong value" with nothing in the log to explain why. Before guessing, ask the server two separate questions: what did it decide to serialize, and — far more rarely — what actually left the socket. These are answered by different tools, and conflating them wastes time.

What the server believes it sent

`SystemManager` keeps a rolling history of which `NetworkSystem` ids it serialized as changed or spawned, per tick, bounded by `SystemManager.SerializationHistoryTickCount`. Anything older than that window is gone; there is no going further back.

- `IsSystemRecentlySerialized(uint systemId)` — was this system serialized within the window, as of the current tick.
- `GetRecentlySerializedSystemIds(HashSet<uint> collectedSystemIds)` — every system id serialized within the window.
- `TryGetSerializedSystemIds(uint tick, HashSet<uint> collectedSystemIds)` — every system id serialized on one specific tick; returns `false` if that tick has aged out of the window.
- `TryGetSerializedSystemIdsAfterTick(uint afterTick, HashSet<uint> collectedSystemIds)` — every system id serialized on any tick after a baseline; returns `false` if part of that range has already aged out, meaning the results are incomplete rather than empty.

```
HashSet<uint> changedSystemIds = HashSetPool<uint>.Rent();

bool isComplete = coreManager.SystemManager.TryGetSerializedSystemIdsAfterTick(lastKnownGo
odTick, changedSystemIds);

foreach (uint systemId in changedSystemIds)
{
    if (coreManager.SystemManager.TryGetSystemReference(systemId, out NetworkSystem networ
kSystem))
        Logger.LogInfo($"Serialized: {networkSystem.AsString()}");
}

HashSetPool<uint>.Return(changedSystemIds);
```

If `isComplete` comes back `false`, the window aged past part of the range you asked for — treat the collected ids as a lower bound, not the whole answer.

Turning ids into something readable

A raw `uint` system id or connection id in a log line is almost never enough to place a fault — it doesn't say which object, which peer, or whether the reference was even valid at the time. Resolve it first:

- `ConnectionExtensions.AsString(this Connection connection)` — describes a `Connection`, or returns `"IsNull"` if it is null.
- `NetworkSystemExtensions.AsString(this NetworkSystem networkSystem)` — describes a `NetworkSystem`, or returns `"IsNull"` if it is null.

Use these in every log line that names a connection or a system, not just when debugging. `TryGetSystemReference` (above) is how you go from a bare id back to the `NetworkSystem` these extensions describe.

Rule for reading a payload

An id inside a packet is a claim, not a fact. Never trust a sender id read out of a payload as "who sent this." The sender is whatever the engine resolved the connection to be at receipt — the `Connection` handed to you by the read path — never a value pulled out of the bytes themselves. If those two disagree, the resolved connection is the truth and the payload value is the anomaly you're investigating.

The raw-packet test hook

`TransportManager.RawPacketReceived` (`RawPacketReceivedHandler(Connection sendingConnection, Reader reader)`) fires only for `PacketType.Raw`. This is not a tap on ordinary arriving traffic.

`PacketType.Raw` is a DEBUG-only packet type — it doesn't exist in the enum in Release at all. The engine itself never sends one; the only producers are the transport test suites, which hand-write a `PacketType.Raw` header themselves to exercise a transport directly (see `Nucleus.Tests/Transports/SynapseTransportTests.cs`). If you want to see what a transport actually put on the wire, you construct and send a raw packet yourself and subscribe to this event to observe it arrive — you don't get it for free by listening in on real gameplay traffic.

```
fixture.CoreManager.TransportManager.RawPacketReceived += (sendingConnection, reader) =>
{
    // Read whatever fields you wrote into the raw packet, in the same order.
    uint id = reader.ReadUInt32();
    ArraySegment<byte> payload = reader.ReadArraySegmentWithCountAsBytes();
};
```

The Reader is reused the moment your handler returns

The `Reader` handed to `RawPacketReceived` belongs to the receive loop, not to you. It is reused on the next packet the moment your handler returns. Copy whatever values you need out of it inside the callback; do not stash the `Reader` reference itself and read from it later.

Round-trip time, jitter and packet loss

The measured figures

`Connection.RoundTripTimeMilliseconds` is a smoothed estimate of the link's round trip time, in milliseconds. It follows the same estimator RFC 6298 ?2.2/?2.3 define for TCP's retransmission timer: the first sample seeds the average outright, and each later sample moves it by an eighth, so one slow reply nudges the figure rather than redefining it. `Connection.RoundTripTimeDeviationMilliseconds` carries the spread the smoothing hides - the link's jitter - and is meant to be read alongside it.

Both are `UnsetRoundTripTimeMilliseconds` / `UnsetRoundTripTimeDeviationMilliseconds` (`uint.MaxValue`) until the link has a first sample. `Connection.IsRoundTripTimeUnset` is `true` exactly while `RoundTripTimeMilliseconds` equals `UnsetRoundTripTimeMilliseconds`. `uint.MaxValue` is used instead of zero because zero is a valid measurement - a loopback link genuinely rounds to it - so it can't also mean "unmeasured" without the two being indistinguishable.

Each peer measures its own view of the link: a client measures its link to the server and holds the figure on the server stand-in `Connection`; a server measures each remote client and holds the figure on that client's `Connection`. The figure carries a small positive bias and never under-reports, since a reply is only read when the receiving peer's loop next polls its transport.

Packet loss

`Connection.PacketLossPercentage` is the share of recent probes that went unanswered, read the way a ping tool measures it: from gaps in the echoed probe sequence. Every probe's reply carries the probe's sequence number, so a sequence that never comes back is a probe that did not round trip. This is round-trip loss, not one-way.

Because it counts sequence gaps rather than timing replies, a reply does not have to land inside the probe interval to count. A late or reordered reply still closes its gap and is not counted as lost, even if a newer probe has already gone out by the time it arrives. The figure is measured over the last 16 probes, so it tracks the link's current condition rather than its whole history and clears within about that many probe intervals once loss stops. It reads 0% on a loopback link, which has no wire to lose on.

The trust-the-client model

`Connection.ReportedRoundTripTimeMilliseconds` and `Connection.ReportedRoundTripTimeDeviationMilliseconds` are the raw figures the remote client stated for its own link - populated on a server's remote client `Connections`, unset on a client. They sit beside `RoundTripTimeMilliseconds` / `RoundTripTimeDeviationMilliseconds`, which the server also adopts these reported values into after its one-time bootstrap probe at authentication. From that point the server never probes the client again; it keeps the figure current from what the client reports on each of its own probes.

This is a deliberate trust boundary. A client that understates or fabricates its own round trip time or deviation only mishandles its own recovery patience - the server judges that client's own acknowledgments too harshly or too leniently based on the figure the client handed it. It cannot reach past its own stream: nothing the server decides about any other peer depends on what one client reports. Packet loss follows the same footing - a server presents a client's reported loss without measuring it itself, and nothing the server decides depends on that figure being honest.

Withholding state until a link is measured

`Connection.CanReceiveState` is `!IsRoundTripTimeUnset`. State is withheld from a connection until its link has been measured, because every timing decision the sender makes about a peer - chiefly how long to wait for an acknowledgment before serving recovery - is denominated in that measurement. Serving state first would mean judging a peer's first acknowledgments against a round trip time that doesn't exist yet, and recovering a peer that was never late.

An unmeasured peer is therefore silent by design, not broken. The wait is one round trip on the link's first, reliably-sent probe; a host's loopback link resolves without any wire round trip at all.

The probe machinery

`RoundTripTimePing` and `RoundTripTimePong` are the two messages that measure a link. Both carry timestamp-free incrementing sequences: `RoundTripTimePing.Sequence` identifies the probe, its `RoundTripTimePong` reply echoes that same sequence, and only the peer that sent the probe ever measures elapsed time - the two peers need no shared clock. A `RoundTripTimePing` also carries the sender's own last-measured `ReportedRoundTripTimeMilliseconds`, `ReportedRoundTripTimeDeviationMilliseconds`, and `ReportedPacketLossPercentage` for the peer to adopt.

Both sides probe: a client probes its server link on a cadence, and a server sends exactly one probe to each client, at authentication, then trusts what that client reports thereafter. `TransportManager.RoundTripTimeIntervalMilliseconds` sets that cadence, in milliseconds, and defaults to 1000 (`TransportManager.DefaultRoundTripTimeIntervalMilliseconds`). Setting it to zero stops the recurring probe; it does not affect the mandatory first probe of a link, since state is withheld until that first probe resolves.

RoundTripTimeDiscovered

`TransportManager.RoundTripTimeDiscovered` fires once per `Connection`, the moment its link is measured for the first time - on the transition out of `Connection.UnsetRoundTripTimeMilliseconds`, or immediately for a host's own loopback link, which has no wire to measure.

The shipped Unity HUD ping row (`NetworkRoundTripTimeStatisticProvider`) subscribes to this event to adopt its server-stand-in `Connection` the moment it's known, rather than polling the transports on every refresh for a link that might by now be measured. It also seeds itself once on `Awake` by checking already-connected transports, since the event can't fire retroactively for a subscriber that didn't exist yet.

Reading a wrong figure

None of these figures surface as a visible fault when they're off - they feed recovery patience, so a wrong reading shows up as traffic, not an error. A round trip time or deviation read too low makes the receiving side judge acknowledgments too harshly and serve recovery that wasn't needed, showing up as too much repair traffic. Read too high, a genuinely late peer waits too long before it gets served recovery at all, showing up as too little repair traffic and a slower recovery than the link's real condition would call for.

Logging

Using Unity? See Engine logging in the Unity console.

The default logger

A plain .NET host gets a `ConsoleLogger` automatically. `LoggingService` registers one in its static constructor, so `Console.WriteLine` output starts flowing before you write any setup code. That's fine when the engine runs behind a terminal. It's wrong for a service: nothing there reads stdout, so every log line is lost.

Swapping the logger

`LoggingService.UseLogger(ILogger)` is the one call that replaces the active logger. It sets the public `LoggingService.Logger` field and re-reads the current logging level, so a fresh sink starts filtering correctly from that point on.

```
LoggingService.UseLogger(new MyServiceLogger());
```

Code that needs to react when the logger changes can subscribe to `LoggingService.LoggerSet`, which fires with the newly registered `ILogger` instance.

```
LoggingService.LoggerSet += logger => Console.WriteLine($"Logger swapped to {logger.GetTyp
e().Name}");
```

Implementing ILogger

```
public interface ILogger
{
    LoggerSetting GetLoggerSetting();
    bool DisableUnconditionalDevelopmentStacktrace();
    void LogInformation(string message);
    void LogWarning(string message);
    void LogError(string message);
}
```

`LogInformation`, `LogWarning`, and `LogError` receive the formatted message string; route each to your host's own logging framework.

`GetLoggerSetting` declares which `LoggerLevel` your logger runs at in each environment (see below). `LoggingService` calls it whenever it needs to re-cache the active level.

`DisableUnconditionalDevelopmentStacktrace` returns `true` when your sink already captures its own stack traces, so the engine doesn't need to attach one unconditionally in development environments.

Levels and cost

`LoggerLevel` is `Error`, `Warning`, `Information`, or `Disabled`. `Error`, `Warning`, and `Information` run least to most verbose; `Disabled` turns logging off entirely rather than sitting at either end of that scale. `LoggerSetting` exposes it as three independent knobs:

Field	Applies to
`Editor`	Running inside the editor
`DevelopmentBuilds`	Development builds outside the editor

`ReleaseBuilds`	Release builds

`ConsoleLogger`'s setting runs `Information` in the editor and development, `Error` in release. Which of the three applies comes from CodeBoost's application state, and in a plain .NET process the default one, `IdeApplicationState`, always reports that it is running in the editor, in Debug and Release builds alike. A .NET host therefore logs at `Information` by default, even in a Release build. To log less, register a logger whose `GetLoggerSetting` returns a lower `Editor` level.

Checking a level before formatting a message avoids the cost of building a string nobody will read. `LoggingService` exposes `IsInformationEnabled`, `IsWarningEnabled`, and `IsErrorEnabled` for this:

```
if (LoggingService.IsInformationEnabled)
    LoggingService.LogInformation($"Tick {tick}: {expensiveDescription()}");
```

When a level is disabled, the guarded branch never runs, so the interpolated string is never built.

Choosing the level at runtime

Sometimes the level is not known until the code runs: a warning the first time, an error once it keeps happening. `Logger.Log` takes the level as an argument, and `Logger.IsEnabled` answers the same question as the three checks above for a level held in a variable:

```
LoggerLevel loggerLevel = desyncCount > 10 ? LoggerLevel.Error : LoggerLevel.Warning;

if (Logger.IsEnabled(loggerLevel))
    Logger.Log(typeof(MatchService), loggerLevel, $"Player {playerName} desynced {desyncCo
unt} times.");
```

`IsEnabled` is always false for `LoggerLevel.Disabled`, and `Log` with `Disabled` writes nothing, so `Disabled` doubles as "decided not to log".

When the decision and the write happen in different places, carry the line as a `LogEntry`: a level and a message together. `LogEntry.Information`, `LogEntry.Warning` and `LogEntry.Error` build one, and the default `LogEntry` is unset (`IsSet` is false) and logs nothing. The engine uses this for violations, where a handler can change or clear the line a raise site seeded (see Violations):

```
LogEntry logEntry = isLobbyFull ? LogEntry.Warning("Lobby is full; the join was refused.")
 : default;

// Later, wherever the line is written. An unset entry writes nothing.
Logger<MatchService>.Log(logEntry);
```

Pass a constant message to a `LogEntry` where you can: an entry built from a constant string allocates nothing, which matters on a path that runs per packet.

Set this before new CoreManager()

Swap the logger before you construct the `CoreManager`. The constructor already logs (a clamped tick rate, for one), and with the default provider the loop starts stepping the moment the constructor returns, so anything raised during bring-up goes to the old sink if you swap afterward. The level is re-read inside `UseLogger`, and again whenever CodeBoost's application state is set, so a host may register its logger and its application state in either order and still log at the level the environment calls for. Calling `UseLogger` again is what makes a level change in your own logger take effect.

```
LoggingService.UseLogger(new MyServiceLogger());
CoreManager coreManager = new();
// The constructor, AddTransportAsync and everything after this line log through MyService
Logger.
```

Network statistics

Using Unity? See Reading network statistics in Unity.

`TransportManager` keeps four cumulative counters. They answer two different questions, and reading the wrong one for the wrong question is the usual mistake.

The counters

Counter	Type	What it counts
`TotalBytesSent`	`long`	Transmitted packet payload bytes, counted at the send chokepoint. Every redundant resend counts again.
`TotalBytesReceived`	`long`	Received packet payload bytes, counted at the receive chokepoint. Every redundant arrival counts again.
`TotalStatePacketPayloadBitsSent`	`long`	Bits of serialized StatePacket payload sent, excluding padding and every packet header. Each serialized StatePacket counts once, no matter how many peers it reached.
`TotalStatePacketPayloadBitsReceived`	`long`	Bits of StatePacket payload received, excluding padding and every packet header. Every redundant arrival counts again, same as `TotalBytesReceived`.

All four are `public long` with a private setter, live on `TransportManager`, and are never reset by the engine.

```
long bytesSent = coreManager.TransportManager.TotalBytesSent;
long bytesReceived = coreManager.TransportManager.TotalBytesReceived;
long stateBitsSent = coreManager.TransportManager.TotalStatePacketPayloadBitsSent;
long stateBitsReceived = coreManager.TransportManager.TotalStatePacketPayloadBitsReceived;
```

Why the two families disagree

`TotalBytesSent`/`TotalBytesReceived` count everything that crossed the transmit and receive chokepoints: headers, framing, and every redundant resend or arrival. They answer "what did the link carry."

`TotalStatePacketPayloadBitsSent`/`TotalStatePacketPayloadBitsReceived` count only the state payload bits inside StatePackets, with headers and framing stripped out. The sent counter also counts each serialized payload once, regardless of how many connections it was transmitted to. They answer "what did replication actually cost."

A feature that fans one serialized StatePacket out to many peers, or that resends over an unreliable link, will move the byte counters more than the bit counters. That gap is expected, not a bug.

Reading a rate

The counters are cumulative for the manager's lifetime. To get a rate, sample a counter, wait an interval, sample again, and subtract:

```
long before = coreManager.TransportManager.TotalBytesSent;
// ... wait one second ...
long after = coreManager.TransportManager.TotalBytesSent;
long bytesPerSecond = after - before;
```

The state-bit counters work the same way; divide the delta by eight to get bytes.

No per-object attribution

There is no counter for what one object, one system, or one member cost. The counters are manager-wide totals. To isolate the cost of a feature, measure the counters with it on, then off, and compare. A test can read `TotalStatePacketPayloadBitsSent` and `TotalStatePacketPayloadBitsReceived` directly, turning a wire-cost regression into a failing assertion instead of a surprise.

Technical limitations

Some names exist in the code but nothing is wired to them

Some names exist in the engine but nothing wires them up yet. Treat them as documentation of intent, not as working settings.

These features were built and then deliberately removed

These were built, shipped for a time, and then deliberately torn out. If you find a doc, comment or stale reference describing one, it is not a bug you rediscovered — it is history.

- **Distance-driven send-interval bands** — a middle interest tier that paced a distant object's sends instead of stopping it outright was removed from the shipped interest types. Per-member send interval and transmission mode still exist, but neither is driven by distance.
- **Full-world rollback physics** — client-side resimulation that restored the whole physics world to a snapshot tick and re-stepped it replaying client state was removed once projected sync proved it could stay pop-free without that cost.

These features are planned but not built yet

Distinct from the permanent limits above: these are absent because the work hasn't happened, not because the design forbids them. They may ship later.

- **Network profiler window** — an editor window attributing traffic to messages, objects and members. The counters it would read already exist; the window itself does not.

A browser/WebGL transport has no register entry either way: it has neither shipped nor been recorded as planned. Server-side rewind, lag-compensated queries and hitbox history are not on this list because they are built, as the Pro feature described in How lag compensation works.

Violation reference

What a violation is

A violation is a struct implementing `IViolation`, raised through `ViolationManager` when a peer sends something the protocol does not permit. The offending operation is always rejected regardless of what happens next; the violation only decides the `ViolationAction` taken against the `Connection` that sent it: `Ignore` or `Kick`. Logging is separate from the action, carried on `ViolationContext<T0>.Log`, a `LogEntry` the raise site may seed and the deciding handler may change or clear. Register a `ViolationHandler<T0>` to override the default action per type, or subscribe a `ViolationDetectedHandler<T0>` (or `IViolationObserver`) to be told about one without changing the outcome.

There are fifteen shipped violation types. Two exist only in Pro.

Peer-behavior violations

These fire from state application, spawning, and RPC dispatch. All default to `Ignore`.

Type	Payload	Check first
`UncontrolledStateChangeViolation`	`SystemId`	Whether the sender actually holds control of that `NetworkSystem` right now; control handoffs racing a state send look identical to this from the receiving side.
`PredictedStateChangeViolation`	`SystemId`	Whether the member is really predicted; a controller never serializes a predicted member's state, so this usually means the member's transmission setup changed without the controller rebuilding.
`EmptyCollectionDeltaViolation`	`SystemId`, `MemberId`	A compliant sender cannot produce a collection delta with no operations, so first suspect a diverged serializer pair between the two builds, not the sender's intent.
`InvalidStateAckViolation`	`AckedTick`	Whether the two peers' tick counters have actually diverged (a restart, a save/load, a manual tick reset) rather than assuming a forged ack.

`RetentionExceededViolation`	`AckedTick`	The peer's latency against `SystemManager.StateRetentionMilliseconds`; a single raise is ordinary heavy loss, not abuse.
`RecoveryUnconfirmedViolation`	`UnconfirmedServeCount`	Whether the peer's connection is dropping the recovery serve itself (packet loss eating the recovery, not just the state it was fixing) before assuming it is stalling on purpose.
`UnauthorizedSpawnViolation`	`SystemId`, `SystemBundleId`, `SystemLocalId`	Whether the server just despawned that system id and the client's own recovery pass re-served it as a full before learning of the despawn; that path raises nothing, so a raise here means the id was never one the server despawned.
`RpcSendPermissionViolation`	`SystemId`	Whether permission was revoked while calls already sent were still in flight; a well-behaved client reaches this on that race alone.
`RpcFloodViolation`	`ReceivedRpcCount`, `AllowedRpcCount`	`RpcManager.MaximumInboundRpcsPerConne against what the client actually needs to send in one drain, before assuming a flood attempt.

Malformed-data violation

`MalformedDataViolation` fires on the server when reading data a client sent throws, so the data is not applied. A well-behaved client never sends data the server cannot read, so it defaults to `Kick` in a release build and logs `Sent data the server could not read.` as a warning. A debug build (`#if DEBUG`) raises it with `Ignore` and no log line, and the read site logs the exception instead, since there the usual cause is a serializer under development. It is never raised on a client, for data with no known sender, for a server Connection (a host that dialled its own address reads from one as a client), or for a host's own client; in those cases the read site logs the exception and nobody is kicked, so a host never drops its own link or its own player over a bad read.

Type	Payload	Check first
`MalformedDataViolation`	`Kind` (`State`, `FullBody`, `Recovery`, `ClientState`, `Message`, `Rpc`), `SystemId` (zero when not for a system)	Whether both builds share the same serializers and member layout for that `Kind`; a version mismatch between client and server builds reads exactly like a forged packet. Run a debug build to see the exception.

Protocol-direction violations

These fire when a client answers about a bundle or scene it was never asked about. The server drives every load and release with a reliable request; nothing in the engine sends these messages upstream, so a well-behaved client cannot produce any of the four. They default to `Kick`.

Type	Payload	Check first
`UnexpectedBundleRequestViolation`	`BundleId`	Nothing on a well-behaved client sends this upstream; if it fires, look for custom code calling the bundle-request path on the wrong side.
`UnsolicitedBundleReportViolation`	`BundleId`, `IsLoadReported`	Whether the server actually requested that bundle load, or actually recorded the client holding it before an unload report; a mismatched build id here looks identical to a forged report.
`UnexpectedSceneRequestViolation`	`SceneHandle`, `IsLoadRequested`	Nothing on a well-behaved client sends this upstream; look for custom code calling the scene-request path on the wrong side.
`UnsolicitedSceneReportViolation`	`SceneHandle`, `IsLoadReported`	Whether the server actually requested that load or release; a scene handle that was never requested and never recorded held (including the unset handle) lands here too.

Pro-only violations

`PacketTransformRejectedViolation` and `UnauthorizedDespawnViolation` do not exist in a Free build; their events live on Pro partials of `ViolationManager`.

Type	Payload	Default	Check first
`PacketTransformRejectedViolation`	`ChannelId`, `PayloadByteCount`	No explicit default is passed at the raise site, so it takes `ViolationManager.DefaultAction` (`Ignore`), and because it is the plain framework default it always logs a warning.	Whether the peer is running the same `IPacketTransform` configuration; a stray packet from something that is not this game reaches the port too, so this is not evidence of an attack against a legitimate peer.

`UnauthorizedDespawnViolation`	`SystemId`	`Ignore`	Whether the client and server builds actually agree on that prefab's removal policy; a version skew between them reaches this honestly.

Lag compensation violation

`LagCompensationViewClampedViolation` exists in every edition, but only lag compensation (Pro) raises it: a lag-compensated query raises it, at most once per query, when the shooter's view is older than the rewind window or newer than anything the server has recorded. The view is clamped into range and the shot is still judged. See The rewind window and the view clamp.

Type	Payload	Default	Check first
`LagCompensationViewClampedViolation`	`RequestedTick`, `ClampedTick`	`Ignore`	The shooter's latency against `LagCompensationManager.Maximum` a single raise is a link that spiked, while a steady stream is either a very slow link or a client asking to be judged against the oldest world it can get.

Reading a violation you just saw

Most types default to `Ignore` and stay silent unless something sets `Log` because a well-behaved peer can reach them through an ordinary race (a control handoff, a revocation, a recovery re-serve) or ordinary loss (`RetentionExceededViolation`, `RecoveryUnconfirmedViolation`, `PacketTransformRejectedViolation`). Treat a single raise of any of those as lossy-edge noise and do not kick for it.

The four protocol-direction violations are different: no engine path on a compliant client produces them at all, which is why they default to `Kick`. A raise there means either a deliberately crafted client or your own code driving the request/report path from the wrong side - check the latter first.

Violations

Using Unity? See Handling violations in Unity

What a violation is

A violation is a rejected protocol fault from a peer: writing state it doesn't control, sending a malformed delta, acknowledging a tick it was never sent, answering a report it wasn't awaiting, and similar. The offending operation is always rejected, regardless of what you decide to do about the Connection. `ViolationManager` only controls the consequence for the peer, not whether the bad data gets applied.

By default almost every violation is silent. Most types are raised with `Ignore`, so the fault is rejected and nothing is logged; the scene-protocol faults (and, in Pro, the bundle-protocol faults) kick the offender. Logging is separate from the action: a violation raised with the framework default still logs a warning ("The offending operation was rejected."), and any raise site or handler can set `context.Log` to log something of its own regardless of the action chosen. To see violations at all, register an `IViolationObserver`. If you want misbehaving clients kicked, or you want to feed violations into your own audit system, you register with `ViolationManager`.

Every violation passes through the same four stages

A raised violation passes through the same four stages, in order, every time:

The deciding handler for that violation type, if one is registered, settles the `ViolationAction`.

Global observers (`IViolationObserver`) are told the settled action, for every violation type.

Per-type subscribers for that one type are told the settled action.

Enforcement applies the settled action against the offending `Connection`.

Only the deciding handler can change the outcome. Observers and per-type subscribers receive the context by read-only reference after the action is already settled — they can log, alert, or count, but they cannot influence what happens.

A throwing stage does not break the pipeline. If the deciding handler throws, the violation falls back to its default action and log line, as though no handler were registered. If an observer throws, the observers after it are still told, and if a per-type subscriber throws, enforcement still happens. Each failure is logged as an error with the exception attached, so a bug in your handler shows up in the log, while a client that tripped a violation is still kicked rather than escaping because your code threw.

Deciding the action

Register at most one deciding handler per violation type through `ViolationManager.RegisterViolationHandler<T0>(ViolationHandler<T0> violationHandler)`. Registering a second handler for the same type replaces the first. Remove it with `TryUnregisterViolationHandler<T0>()`, which takes no argument — it always targets the one handler for `T0`.

```
coreManager.ViolationManager.RegisterViolationHandler<UncontrolledStateChangeViolation>((r
ef ViolationContext<UncontrolledStateChangeViolation> context) =>
{
    context.Log = LogEntry.Error("Wrote state it does not control.");
    return ViolationAction.Kick;
});
```

The handler receives a `ViolationContext<T0>` and returns a `ViolationAction`:

- `Connection` — the offending Connection, read-only.
- `Violation` — the typed violation payload, read-only.
- `Action` — seeded with the framework default for this raise; the handler can read it back and return it unchanged to accept the default, or return something else to override it.
- `Log`: seeded by the raise site, if it set one; the handler can leave it, replace it with its own `LogEntry`, or clear it with `context.Log = default` to silence it.

```
public delegate ViolationAction ViolationHandler<T0>(ref ViolationContext<T0> violationCon
text)
    where T0 : struct, IViolation;
```

The two actions

`ViolationAction` has two values:

- `Ignore`: no consequence, Connection stays intact.
- `Kick`: disconnects the Connection.

`ViolationManager.DefaultAction` is `Ignore`. Each violation type is raised with its own default, which overrides `DefaultAction` when no handler is registered for that type: most types default to `Ignore`, and the bundle- and scene-protocol faults (unsolicited bundle/scene reports, unexpected bundle/scene load requests) default to `Kick`. A registered handler can still return whatever action it wants regardless of that per-raise default.

Logging no longer rides on the action. A violation raised through `ViolationManager.DefaultAction` (the framework default) is logged as a warning with the message "The offending operation was rejected."; a violation raised with its own explicit default logs only if the raise site or the deciding handler set a `LogEntry` on `context.Log`. When `Log` is set, the framework logs it as `Violation [<TypeName>] from Connection [<connection>]: <message>` at the level the `LogEntry` carries.

Watching without deciding

To audit every violation type without deciding any of them, implement `IViolationObserver` and register it with `ViolationManager.RegisterViolationObserver(IViolationObserver violationObserver)`:

```
public sealed class ViolationAuditSink : IViolationObserver
{
    public void OnViolation<T0>(in ViolationContext<T0> violationContext) where T0 : struc
t, IViolation
    {
        // Log, count, alert — cannot change violationContext.Action.
    }
}
```

One registration hears every type, including types added later. Remove it with `TryUnregisterViolationObserver(IViolationObserver violationObserver)`.

To watch one type only, use `RegisterViolationDetectedHandler<T0>(ViolationDetectedHandler<T0> violationDetectedHandler)` or the matching per-type event (for example `UncontrolledStateChangeViolationDetected`) — both write the same subscription. A subscriber registered this way pays nothing for violation types it didn't ask about.

Only a peer running a server enforces a kick

`Kick` is enforced only on a peer that has started a server; a client that settles a violation on `Kick` against its own server Connection just logs it, since an impossible acknowledgment from its own server there is a tick-ordering hiccup, not abuse to punish. On a server, the kick is routed through `ServerManager.KickClient`, which purges what's queued to the offender, stops serving it state, and closes the link at the end of the frame — not a socket drop mid-tick.

What the server trusts

Observation gates everything

Every access check the engine makes, for a state write or for a remote call, starts with the same test: is the sending connection an observer of the system being addressed. That test runs first, before the access value and before whether the sender controls the object. A connection that does not observe a system is rejected outright, regardless of `StateWriteAccess` or `RpcSendAccess`.

This means a client cannot reach a system by guessing its id. An identifier that names something the client was never told about, or has since lost interest in, resolves to nothing on the server. Access - `Controller` or `AnyClient` - only decides what an observer may do once it has cleared this gate. It never substitutes for being told about the object in the first place.

The controller is checked next: a system's controller may always write to it or call it, under any access. An observer that is not the controller is admitted only when the access in force is `AnyClient`.

Nothing travels as a grant

The server tells each observer two things about a system: the access value in force, and a one-bit hint of whether that observer personally may act. Neither is a permission a client can spend on its own. The server re-checks every incoming write and every incoming call against its own copy of the access, on every packet, so a client that forges or replays a favorable-looking declaration gains nothing - the check that matters runs on the receiving side, not the sending side.

The capability hint exists only so a well-behaved client avoids sending something the server is going to refuse anyway. A client that ignores the hint and sends regardless is met with the same enforcement as one that never saw it.

Who actually sent it

An identifier a client places in a payload is a claim about itself, nothing more. What a handler is handed as the sender is the `Connection` the transport resolved for that packet - `RpcContext.SenderConnection` for a call, the equivalent resolved connection for a state write. That resolution happens below any code the client controls, so there is no field a client can set to speak as someone else.

For a call the server relays on to other observers, the relayed copy still carries the original sender's connection id, so attribution survives the hop; a client that receives a relayed call is being told who actually said it, not who forwarded it. Only the authority writes that original-sender id. A call arriving from a client is attributed to the client that sent it, whatever originator id its packet names, so a client cannot dress a call up as coming from another client to fool a handler, the relay's sender exclusion, or `RpcContext.SenderConnection`.

Connections are not players

A `Connection`'s `RemoteAddress` is where it connected from, observed by the server rather than asserted by the client, which is what makes it usable as part of an identity check - but it names a location, not a player, since multiple clients behind the same NAT share one address.

A `Connection.Id` is assigned per session and reused once a connection is gone; nothing about it survives a disconnect. Anything that needs to recognize the same player across a reconnect keys on `Connection.Identity` instead - the value an authenticator stamps in on approval, meant to persist across sessions. Correlating a returning `Identity` with what it previously owned is game code's job; the

engine holds no player-to-object mapping of its own.

Arbitration is not a security boundary

`StateWriteAccess.AnyClient` (Pro) lets every observer, not just the controller, write a system's state. A Free build has no `StateWriteAccess` at all: only a system's controller may write its state, so there is never a contested write to arbitrate.

Contested multi-writer arbitration - resolving which of several observers' simultaneous writes to a system wins - is a Pro feature. Where it exists, the resolution is not adversarially safe: it is arbitrary but stable, and a client that keeps reconnecting to change its own connection id can farm a favorable outcome under it. Arbitration decides whose write is shown when two are equally legitimate; it does not decide whether a write should have been allowed, and it is not a defense against a client trying to win by gaming the mechanism. Anything where that matters needs its own arbiter in game code, sitting on top of the access check, not the arbitration itself.

What Nucleus does not do

Nucleus does not encrypt transport traffic. That is not built in, and adding encryption on top is game code's responsibility.

If you need to protect the bytes themselves - encryption, compression, obfuscation, or a custom integrity check - the one seam for it is the packet-transform layer, `TransportManager.PacketTransform` (Pro-only): it rewrites every packet's payload on its way to and from the transport, on both peers, before anything else in the engine sees it. That layer is not otherwise documented on this page.

A shooter's view is bounded, not trusted

Lag compensation (Pro) judges a client's shot against the moment the client says its screen was showing. That view is the client's claim, so the server bounds it: a view older than `MaximumRewindMilliseconds`, or newer than anything the server has recorded, is clamped into range, still judged, and raises `LagCompensationViewClampedViolation`, ignored by default. Within the window a client can still name any moment, so the window itself is the limit on how far a shooter is favored. The ray is the client's claim too: clip it at walls on the server against the present-time scene, and the queries test it only against the hitboxes the server's own history holds, never against anything the client reports about a target. See The rewind window and the view clamp.

Enforcement only happens on the server

Every check above - observation, controller status, access, arbitration - runs on the server's copy of the system. A client build can run the same code and reach the same verdict locally for prediction or UI purposes, but that verdict changes nothing on the wire. A kick decided by a client, or a write a client's own code declines to send, is not enforcement; the server never asked for that answer and does not use it. The only check that matters is the one the server performs when the packet arrives.

Core API

Documentation for driving Nucleus directly from C#, with no game engine involved. The same subjects from the Unity integration live under Unity.

- Core, Managers and the Tick Loop
- Systems, Objects and Spawning
- Replicated State and Change Callbacks
- Control, Client State and Prediction
- Messaging and Remote Calls
- Interest Management and Scaling
- Networked Scenes, Content Bundles and Persistence
- Physics Synchronization
- Lag Compensation
- Transports, Connectivity and Hosting
- Serialization, Codegen and the Wire
- Diagnostics and Troubleshooting

What area of interest is

Without any condition, every system replicates to every client

Without any interest condition registered, Nucleus replicates every started NetworkSystem to every authenticated client, every tick. This is the broadcast-to-everyone default, and it is not something interest opts you into: it is what the engine does before interest exists at all.

Interest layers restriction over that default. It never grants a connection anything the default did not already give it. Register no condition anywhere and the evaluation pass does nothing at all - it costs one comparison per tick and stops.

Conditions only restrict

An `IInterestCondition` returns an `InterestResult` carrying an `InterestOpinion` for each effect it declared in its `Capabilities`. That opinion is one of three values: `None` (no opinion, leave it to the others), `Unmet` (veto the effect for this pair), or `Met` (satisfied, no restriction).

`None` is the fold identity. The resolver combines every stacked condition's opinion with a fold equivalent to AND, so an abstaining condition never fights another condition's verdict. Only `Unmet` ever restricts anything; `Met` and `None` are indistinguishable to the resolution and differ only in what the condition means to say. Nothing a condition returns can push a connection past the streamed default - the whole vocabulary is restriction.

Conditions can be registered globally on the `InterestManager`, covering every system that starts from then on, or per-system through the system's own registration. Both stack, and the most restrictive verdict per effect wins across all of them.

Resolution is per pair, inside serialization

Interest resolves one `(NetworkSystem, Connection)` pair at a time. The evaluation runs inside the serialization pass, ahead of the state passes, so a verdict that changes a pair's membership takes effect on the same tick it was decided - a system that just got culled for a connection does not serialize to it that tick.

Because resolution happens on this hot path, an `IInterestCondition` implementation must be a pure read: it must not start or stop systems, mutate observers, or block.

The two effects

`InterestEffect` has two flags, and a condition declares which of them it ever speaks to:

- **Spawn** - whether the system exists for the connection at all. An unmet spawn condition means the system never spawns there, and a system already spawned is despawned exactly as an ordinary despawn releases it.
- **Stop** - whether the system is still replicating for the connection. An unmet stop condition freezes the system for that connection while keeping its object retained, so it can stay visible without costing replication; only a later unmet spawn condition reclaims it.

See stopping-versus-culling for what each looks like from the receiving connection's side.

Cadence and stagger

`InterestManager.EvaluationCadenceTicks` sets how many ticks pass between evaluations of any one system's conditions; the default is 5. A value of 1 evaluates every system every tick. Systems are staggered across the cadence by their `Id`, so on any given tick roughly one cadence-th of the world is

evaluated rather than the whole of it landing on one tick.

The cadence is skipped for two cases that cannot wait for a stagger slot to come around: a connection that just authenticated is evaluated against every started system immediately, and a system that just started is evaluated against every authenticated connection immediately. Both land on the tick they happen, before anything serializes, rather than trickling in as each system's stagger slot arrives.

Evaluation costs one check per system per connection

The evaluation is O(systems ? connections). It is cadenced and staggered to keep that honest, which is fine for demo and mid scale; a world of thousands of systems and connections wants a spatial broadphase feeding it candidate pairs instead, and that is future work rather than something the current pass fakes.

A connection is not registered as an observer the moment it authenticates. Registration - and so interest evaluation - waits for its link to be measured (`Connection.CanReceiveState`), because every timing decision the sender makes about a peer is denominated in its round trip time. A connection authenticated before its link is measured is held back and registered the moment that measurement completes, typically one round trip later, and immediately for a host's own client.

Where to go next

On the API track, see registering-interest-conditions for `InterestManager.AddCondition` and per-system registration. On the Unity track, see the Unity Interest Manager and Network Interest Object components.

Distance interest conditions

Using Unity? See Limit what a client receives by distance.

What you subclass

`DistanceInterestConditionBase` culls or stops a `NetworkSystem` by how far it is from a connection's controlled objects. It derives from `InterestPositionConditionBase`, which owns the shared measuring logic: how a source position is looked up and cached per tick, and how the nearest controlled object is found. The base declares `Capabilities => InterestEffect.Spawn | InterestEffect.Stop` because both resolutions come from the one distance measurement, not two separate passes.

To write your own condition, subclass `DistanceInterestConditionBase` and implement `TryReadSourcePosition(NetworkSystem networkSystem, out Vector3 position)`. That is the only method an integration owes: a raw position read for the system in front of you, world space. Everything else — memoizing the read per pass, walking a connection's controlled objects, picking the nearest, and branching the ladder — is handled for you.

```
protected abstract bool TryReadSourcePosition(NetworkSystem networkSystem, out Vector3 pos
ition);
```

A ladder is built from a spawn cutoff and a stop cutoff

Construct the condition with two cutoffs:

```
public DistanceInterestLadder(float stopDistance, float spawnDistance)
```

`stopDistance` is the distance past which the system stops for the connection but keeps its spawned object. `spawnDistance` is the distance past which the system is despawned. `Resolve(float distance)` checks despawn before stop, so a distance past both cutoffs despawns rather than merely stopping — a nearer effect never masks a further, more restrictive one.

`DistanceInterestLadder.UnsetDistance` is `0f`. A cutoff set to it is disabled, not "zero range": leaving `stopDistance` or `spawnDistance` at `0` means that rung never restricts anything, regardless of measured distance.

What the distance is from

The measurement is always from a connection's controlled objects to the system being evaluated, never from a fixed camera or observer point. `InterestPositionConditionBase.TryGetNearestControlledDistance` walks the connection's controlled `NetworkSystems`, reads each one's position through the same `TryReadSourcePosition` hook, and keeps the nearest. Nearest wins across multiple controlled objects, so a system is interesting if it's near any of them — a player driving both a character and a vehicle stays covered.

How many controlled systems are measured is capped by `InterestManager.MaximumControlledInterestObjects`, defaulting to `DefaultMaximumControlledInterestObjects` (`1`). The cap counts NetworkSystems, not objects: a controlled prefab made of two systems takes two slots, and below the count the systems measured are arbitrary rather than the nearest, so size it to the total systems a player controls. Set it to `InterestManager.UnlimitedControlledInterestObjects` (`0`) to measure every controlled object instead.

```
public uint MaximumControlledInterestObjects = DefaultMaximumControlledInterestObjects;
```

A ceiling below the controlled count keeps an arbitrary object, not the nearest one. Controlled objects are held in a set, so a ceiling of one against a player driving three things measures whichever the set hands back first. Picking the nearest instead would mean measuring all of them — the exact cost the ceiling exists to avoid. Set the ceiling to the number of objects a player actually drives, or to `UnlimitedControlledInterestObjects`, and the question doesn't come up.

Unmeasurable pairs restrict nothing

When a distance can't be measured (the connection controls nothing yet, or a controlled object's position can't be read), the condition answers unmeasurable and restricts nothing, rather than culling. This is what a system whose object has been destroyed or has not finished spawning looks like: `TryReadSourcePosition` returns `false`, the pair is skipped, and interest for it is left alone. Culling on an unknown distance would empty the world for a player whose character hasn't finished spawning.

Distance conditions ship only in Pro

Every file behind this page — `DistanceInterestConditionBase.Pro.cs`, `InterestPositionConditionBase.Pro.cs`, and `DistanceInterestLadder.Pro.cs` — is a `.Pro.cs` file. A free build has the interest system but not distance conditions.

The host interest report

Using Unity? See Making a host see what its players see.

The report tells a host what its own client half would have received

A host runs the server and a client in the same process, so it already holds every object in the world. That leaves it no way to know what its own client half would actually have been sent, unless something resolves the question and hands back an answer. `InterestManager.HostInterestEnabled` does that: turned on, the interest pass also resolves the host's own client against the registered conditions, exactly as it would a remote client.

The answer is a report, never a decision. A host is the server, and nothing is culled, no observer is dropped, and no despawn is written for it because of this resolution. The object goes on existing, replicating, and being controllable exactly as it always did. All that changes is that the engine now tells you what a client in the host's position would be holding.

Reading the report

Each `NetworkSystem` exposes what its resolution settled on:

```
public InterestMembership HostInterestMembership { get; private set; }
```

And raises a change to it:

```
public event HostInterestChangedHandler? HostInterestChanged;

public delegate void HostInterestChangedHandler(InterestMembership previousInterestMembers
hip, InterestMembership currentInterestMembership);
```

`HostInterestChanged` is raised only on a host, and only while `HostInterestEnabled` is set. A dedicated server has no client half to answer for, and a client is already told the same thing directly by its own spawns and despawns, so neither ever raises it. It arrives on the network loop thread inside the serialization pass, on the interest cadence, and a handler is free to do engine work with it.

Which membership is worth acting on

`HostInterestMembership` reads `InterestMembership.Streamed` everywhere the question does not arise: on a client, on a dedicated server, and on a host with `HostInterestEnabled` cleared.

`InterestMembership.Unspawned` is the one that means the object would not be there at all, and is therefore the one worth hiding on. `InterestMembership.Stopped` means a client would still be looking at the object while receiving nothing further for it, so hiding on `Stopped` shows the host less than its players see.

The return guarantee

A system leaving the world, or the host's client half dropping its link, returns the membership to `InterestMembership.Streamed` and raises that too. A handler that hid something on `Unspawned` is always told to put it back, rather than being left holding an invisible object.

Why it is off by default

Resolving the host's own client costs the host connection an ordinary condition pass per evaluated pair, the

same one every remote client already pays. A world pays that cost for an answer it may have nothing to do with, so the report stays off until a game asks for it. Turn it on in a host world that should see what its players see, and leave it alone on a dedicated-server build, which has no client half to answer for.

The switch is read on every interest pass, not once at startup, so a world can turn it on or off mid-session. Turning it off returns each system's membership to `InterestMembership.Streamed`, and raises that change, as the system is next resolved.

Which conditions can honestly differ for a host

A condition that is structurally meaningless for a host still exempts itself. The bundle gate is the clearest case: a host's own client shares the process with the server and so already holds every content bundle the server loaded, so `BundleInterestCondition` abstains for it. It abstains for any Connection with `IsHostLoopback` set, and for one on a local, in-process transport such as Yak.

The scene condition is not one of those. A host is placed in scene instances by the same protocol as any other client and records the instances it was placed in, so the scene gate reads a host's real scene records and settles against them the same way it would for a remote client. What resolves honestly for a host is what can honestly differ for it: position and scene above all.

Interest Management and Scaling

Pages in this category:

- What area of interest is
- Distance interest conditions
- Registering interest conditions
- Writing an interest condition
- An object is missing, or will not go away
- Stopping versus culling
- What scales and what does not
- How distances are measured: pairwise and the spatial grid
- Interest level of detail
- Spawn pacing
- Supplying positions to the interest system
- The host interest report

Interest level of detail

Using Unity? See Level of detail in Unity

Level of detail is a measurement, not a behaviour

Level of detail is a measurement, not a behavior. Nothing in the engine acts on a band: it does not cull an object, it does not change what a system sends, and it does not touch the per-member send interval, which is configured separately. A band tells the game how far an object is from a peer, in three steps instead of a raw distance. What the game does with that reading is up to the game.

Building a ladder

`InterestLevelOfDetailLadder.TryCreate` validates two ascending rungs and builds the ladder they describe:

```
if (InterestLevelOfDetailLadder.TryCreate(nearDistance: 20f, farDistance: 60f, hysteresis:
 InterestLevelOfDetailLadder.DefaultHysteresis, out InterestLevelOfDetailLadder ladder))
{
    interestManager.SetLevelOfDetail(ladder);
}
```

`TryCreate` refuses:

- Either rung being non-finite, zero, or negative.
- `nearDistance` greater than or equal to `farDistance` — equal rungs would describe a band no distance can land in.
- A `hysteresis` that is `NaN`, negative, or greater than or equal to `InterestLevelOfDetailLadder.MaximumHysteresis` (`1f`).

`InterestLevelOfDetailLadder.DefaultHysteresis` is `0.1f`. On failure, `TryCreate` writes an unconfigured ladder to the `out` parameter and returns `false`.

Configuring the world

`InterestManager.SetLevelOfDetail` installs the ladder:

```
interestManager.SetLevelOfDetail(ladder);
```

The current ladder reads back from `InterestManager.LevelOfDetail`. Its `IsConfigured` property reports whether it holds two valid rungs, and `WidestDistance` is the far rung, unsquared, in metres. Passing an unconfigured ladder turns level of detail off: every pair then resolves to `InterestLevelOfDetail.Near`.

`InterestManager.LevelOfDetailCadenceTicks` sets how many ticks pass between one resolution of a system's band and the next; it defaults to `InterestManager.DefaultLevelOfDetailCadenceTicks` (`10`).

Opting a system in

A `NetworkSystem` is measured only if it opts in:

```
networkSystem.SetLevelOfDetailEnabled(true);
```

`NetworkSystem.IsLevelOfDetailEnabled` reports the current opt-in state. Turning it off drops any band already held for that system, so a later read answers `Near` rather than the last value measured before

tracking stopped.

Read the resolved band for a specific connection with `GetInterestLevelOfDetail`:

```
InterestLevelOfDetail band = networkSystem.GetInterestLevelOfDetail(connection);
```

There are three bands, and Near means no level of detail at all

`InterestLevelOfDetail` has three values: `Near` (0), `Medium` (1), and `Far` (2). `Near` is not the finest level of detail — it is the absence of level of detail entirely. An unconfigured ladder, a position that cannot currently be measured, and a peer's own controlled object all resolve to `Near`, on the same footing as an object nobody ever measures.

Hysteresis at a boundary

Each rung's release boundary is precomputed from `1 - hysteresis` at ladder construction, not tested against the raw rung distance every time. A rung is tested at its release boundary only while the object already sits in that band or a coarser one, so coarsening happens at the rung itself and refining happens a dead band inside it.

Without that dead band, an object sitting exactly on a rung — or a player walking back and forth across one — would flap between two bands on every resolution, and anything reading the band would see a stream of changes describing jitter rather than movement. With the default 10% hysteresis, a 40-metre rung holds a 4-metre release band: comfortably wider than the distance an ordinary walking speed covers between resolutions.

Level of detail ships only in Pro

Level of detail is part of Nucleus Pro.

Forcing a re-measure

`RefreshInterest` takes an `isLevelOfDetailRemeasured` argument to force a system's band to be re-resolved off its cadence slot, for a system that moved discontinuously and cannot wait out its stagger.

How distances are measured: pairwise and the spatial grid

Choosing a technique

`InterestManager.SetInterestTechnique(InterestTechnique interestTechnique, GridInterestAxes gridInterestAxes = GridInterestAxes.Xyz)` sets how every distance condition in a world resolves distance. It is one setting on the manager, not a flag per prefab, because the two techniques are built to reach identical verdicts: the difference is cost, never behavior.

```
interestManager.SetInterestTechnique(InterestTechnique.SpatialGrid, GridInterestAxes.Xz);
```

The current choice reads back from `Technique` and `InterestAxes`.

`InterestTechnique.Pairwise` is the default. It measures each connection's nearest controlled object directly, with no index in front of it. It is the right answer where most objects are in range of most players anyway, which is most worlds at demo and mid scale.

`InterestTechnique.SpatialGrid` narrows candidates through the shared `SpatialInterestGrid` first, then measures the survivors exactly against the same cutoffs and the same ladder. Because the exact measurement is identical either way, the grid can never produce a different verdict from pairwise — it only changes what gets measured.

When the grid pays off

The grid buys narrowing, and narrowing is only worth its own cost where the cutoff genuinely excludes most pairs. `InterestResolutionBenchmarkTests` resolves 1000 systems against 100 to 1000 players and measures both worlds:

- A world 400 metres across against a 25 metre cutoff: the grid runs at 1.4x to 1.7x the speed of pairwise.
- A world small enough that everything is in range of everything: the grid runs at 0.6x of pairwise, because the radius query buys no narrowing and the exact measurement still has to happen.

Pick the technique from the world's geometry, not from player count.

Restricting the axes

`GridInterestAxes` chooses which axes a distance is measured across: `Xyz` (the full volume, and the default), `Xy`, `Xz`, or `Yz`. It applies to both techniques, not just the grid, so a world that plays out on a surface — top-down or side-on — can measure only in that surface and stop treating a tower directly overhead as a step away.

The choice resolves to a per-axis multiplier (one on each measured axis, zero on the omitted one) rather than a branch, so nothing on the hot path tests which axes are live. `Xyz` resolves to the mask of ones, which leaves the arithmetic identical to a world that never chose a plane at all.

The technique is locked once a distance condition registers

`SetInterestTechnique` is refused, with a logged error, once any distance condition has registered in the world. A single pass cannot measure in two planes at once, and which one won would become an accident of evaluation order rather than a decision anyone made. Set the technique and the axes during setup, before anything carrying a distance rule spawns.

SpatialInterestGrid

`SpatialInterestGrid` is the engine-free index `InterestTechnique.SpatialGrid` narrows through. It knows nothing about transforms; it answers "which of these points lie within this radius of this point."

- `TryBeginPass(uint pass)` opens a pass, clearing the grid when the pass has moved on. It returns `true` when the caller must re-insert this pass's points, so several callers can share one grid within a pass and only the first pays to build it.
- `Insert(uint id, Vector3 position)` files one identified point for the current pass. A non-finite position is dropped rather than filed.
- `CollectWithin(Vector3 center, float radius, HashSet<uint> results)` adds the identifiers of every inserted point within `radius` of `center`. Candidates are tested exactly before being added, so a result is genuinely in range and not merely in a probed cell.
- `IsIndexed(uint id)` reports whether an identifier filed any point at all this pass, which is how a caller tells "indexed and out of range" apart from "never indexed."
- `ConfigureAxes(GridInterestAxes gridInterestAxes)` chooses the plane, taking effect at the next rebuild rather than mid-pass.

`CellSize` is a public read-back with a private setter, not a fixed constant. `DefaultCellSize` (32 units) is a floor: each `TryBeginPass` that rebuilds widens the cell size to the widest radius any query asked about in the previous pass. Correctness never depends on the cell size — a query probes every cell its bounding box touches and tests each candidate exactly — so a mis-sized cell only costs more probes or more candidates, never a wrong answer.

Scale caveat

Resolution is O(systems ? connections), cadenced and staggered across systems. That is honest for demo and mid scale. A world of thousands of systems and peers wants a true spatial broadphase feeding candidate pairs directly; this pass does not fake one, and that remains future work.

An object is missing, or will not go away

A client receives objects from a scene it never loaded

No rule gates scenes by default in a way you can see failing: an ungated spawn is not an error anywhere on the server, it simply lands in the wrong scene on the client. The Unity inspector for `UnityInterestManager` warns about exactly this when its rule list has no scene rule registered: "No rule gates scenes. A client will receive objects from scenes it never loaded, which puts them in the wrong scene rather than raising an error, and several open instances of one scene cannot be told apart by distance."

Fix: add the scene rule to the world's interest conditions unless you are arbitrating scene membership with a rule of your own.

Nothing is ever culled, however large the stop distance is

A distance condition reads positions through its own `TryReadSourcePosition` (the Unity integration's conditions read the linked GameObject's transform), not through the world reader registered with `InterestManager.SetPositionReader`. When it cannot measure a pair it answers unmeasurable, and an unmeasured pair is never allowed to restrict anything, so no cutoff culls or stops it. That happens when the connection controls nothing yet, so there is no point to measure from, or when `TryReadSourcePosition` returns `false` for the object or for every object the connection controls.

Fix: make sure each player controls the object it plays through, and that the condition's `TryReadSourcePosition` answers for both the evaluated object and the controlled ones. The world reader is used only by level of detail: without one, level of detail logs an error once and every pair reads as Near, but culling is unaffected.

An object disappears at range when it should only freeze, or the reverse

`DistanceInterestLadder.Resolve` returns an unmet spawn past the despawn cutoff and only an unmet stop past the stop cutoff — despawn always outranks stop from the same measured distance. Whether an unmet stop actually removes the object is `NetworkSystem.DespawnWhenStoppedEnabled`: off, the object freezes in place and keeps its last state; on, it is treated as culled once stopped.

On a grouped object (several systems sharing one `NetworkSystemGroup`), a stop is a vote, not a unilateral despawn: a member votes only when its stop is unmet *and* its `DespawnWhenStoppedEnabled` is set, and the object leaves only once every member votes to. A member that freezes instead of voting holds the object in place by abstaining, so one system in the group with `DespawnWhenStoppedEnabled` off is enough to keep a "should have despawned" object sitting there. An ungrouped system has no sibling to defer to, so its own unmet stop with the flag on despawns it directly.

Fix: check `DespawnWhenStoppedEnabled` on every system in the group, not just the one you're staring at.

Objects flicker in and out at a cull boundary

There is no hysteresis on a distance cull. `Resolve` is two bare distance-versus-cutoff comparisons with nothing damping the boundary, so a player oscillating around a cutoff between evaluations flips the verdict every pass. The only thing standing between a player's movement and a flicker is `EvaluationCadenceTicks`: distance is only re-measured once per that many ticks, so as long as a player cannot close the gap between "just inside" and "just past" the cutoff within one cadence, the flicker never has a chance to fire.

Hysteresis does exist in this codebase, but only on the level-of-detail ladder (a fraction-of-a-rung band a distance has to re-cross before it switches back). It cannot fix this: a level-of-detail band is only a reading for the game. It never changes what is sent or how often, nor whether an object spawns, stops, or despawns, so it has no bearing on a cull boundary.

Fix: widen the gap between the stop/despawn cutoff and how far a player can move in one `EvaluationCadenceTicks`, either by raising the cutoff, lowering the cadence, or both. There is no setting to smooth the boundary itself.

A new client sees nothing for a second or two

Registration as an observer waits for more than authentication. A connection authenticates before its round trip time is known, and it is held back from every started system until `Connection.CanReceiveState` is true — which happens once its link is measured, typically one round trip later, or immediately for a host's own client. A client stuck in this window is authenticated but registered against nothing, so it appears to see no objects at all for a moment.

This is expected and self-resolving; there is nothing to configure. If the gap never closes, check that the transport is actually completing a round trip for that connection.

A join or scene load arrives as one huge tick and drops

Spawn pacing exists to prevent exactly this: `InterestManager.MaximumSpawnsPerTick` caps how many objects may start replicating to one client on a single tick, deferring the rest to later ticks. Left at zero it admits everything at once — a full join or scene load lands as a single, maximally expensive tick, the worst one to lose a packet on.

To check whether the ceiling is actually the bottleneck, read:

- `InterestManager.TotalSpawnsDeferred` — total deferrals so far; a world that never reaches the ceiling stays near zero.
- `Connection.SystemsAwaitingSpawnAdmissionCount` — how many spawns a specific connection is still owed.
- A `LogWarning` fires once a connection has been owed spawns for `InterestManager.SpawnAdmissionStallWarningTicks` (90) ticks without one being admitted, naming the connection and the current `MaximumSpawnsPerTick`.

Fix: raise `MaximumSpawnsPerTick`, or spawn less at once. A `NetworkSystemGroup` is always admitted whole rather than starved, so also check for a single group larger than the ceiling — that alone will trigger the stall warning.

Setting the technique or axes logs an error and does nothing

`InterestManager.SetInterestTechnique` is refused, with a logged error, once any distance condition has already registered in that world — tracked internally the moment a position condition is added, globally or per-system. One technique and one plane govern a world for its whole lifetime because the shared spatial index applies its axis mask when a point is filed and again when it is queried; changing either mid-session would have one pass measuring in two of them.

Fix: call `SetInterestTechnique` during setup, before any system carrying a distance rule can spawn — never after.

A distance rule with no spawn range, only a stop range

This is specific to the authored Unity rule (`AuthoredDistanceInterestCondition`), not the underlying

condition type. The inspector-authored rule only exposes a stop distance; it always builds its condition with the despawn cutoff unset (`DistanceInterestLadder.UnsetDistance`), so an object authored this way never culls by distance at all, only stops.

The underlying type has no such limitation. `DistanceInterestCondition` (Unity's own binding of the engine-neutral `DistanceInterestConditionBase`, not the inspector-authored rule) takes a `stopDistance` and a `spawnDistance` independently in its constructor, and so does the `DistanceInterestLadder` beneath it. Construct one yourself in code (or register a global condition) to get culling by distance instead of authoring through the inspector.

Every distance lever on this page is Pro-only

Distance interest conditions, the spatial grid, level of detail, and spawn pacing are all Pro-only — their implementations live in files suffixed `.Pro.cs`: `InterestManager.Conditions.Pro.cs`, `DistanceInterestLadder.Pro.cs`, `DistanceInterestConditionBase.Pro.cs`, `SpatialInterestGrid.Pro.cs`, `InterestManager.LevelOfDetail.Pro.cs`, `InterestManager.SpawnPacing.Pro.cs`, and the Unity-side `AuthoredDistanceInterestCondition.Pro.cs` / `DistanceInterestCondition.Pro.cs`. On a free build these files are absent, so none of the above levers exist: no distance rule, no spatial grid, no level of detail, no spawn ceiling. The scene rule and the base interest system are not gated this way and ship in free.

If a build behaves as though one of these settings simply doesn't apply, confirm it's a free build before debugging further.

Registering interest conditions

Using Unity? See Network Interest Object

Two registration routes

`InterestManager.AddCondition` registers a condition globally. It is folded onto each `NetworkSystem` when that system starts, not consulted per pair, so it governs everything spawned from here on and leaves what is already started alone:

```
coreManager.InterestManager.AddCondition(interestCondition);
```

`NetworkSystem.AddInterestCondition` registers a condition against one system alone, a per-prefab rule as opposed to a world-wide one:

```
networkSystem.AddInterestCondition(interestCondition);
```

Both stack, and every condition can only restrict. `AddCondition` rejects a condition whose `Capabilities` is `InterestEffect.None`, logging an error, because nothing would ever consult it.

The two routes meet when a system starts: the globals registered on the `InterestManager` are folded onto the system and sorted there, by the effects each condition declares, alongside whatever was registered on the system itself. Register during setup and the distinction never arises; register mid-session and the already-spawned world picks the rule up as its objects respawn.

Removing conditions

`InterestManager.TryRemoveCondition` unregisters a globally registered condition and returns `true` when it was registered and removed. `NetworkSystem.TryRemoveInterestCondition` does the same for a system's own registration.

```
bool wasRemoved = coreManager.InterestManager.TryRemoveCondition(interestCondition);
bool wasRemovedFromSystem = networkSystem.TryRemoveInterestCondition(interestCondition);
```

Systems already started keep the fold they took at spawn and drop it when they next start. Removing the last condition of an effect leaves that effect's current resolution in place rather than restoring the default.

`InterestManager.HasConditions` is true while any condition is registered globally. A world driving interest purely through per-system conditions registers nothing globally, and `HasConditions` correctly stays false — that is not a fault.

Default conditions

`InterestManager.DefaultConditions` lists the conditions the engine registered on its own behalf, in registration order: the scene spawn gate (`SceneManager.SpawnGate`), registered globally so a client only ever sees the contents of a scene it has loaded. The bundle gate (`BundleManager.SpawnGate`, Pro) is not one of these — it is deliberately left unregistered, for the game to attach globally or per-prefab wherever bundle gating should apply. `DefaultConditions` is empty for a peer whose engine registered none.

`RemoveDefaultConditions()` unregisters every one of them, leaving the game's own registrations alone, and returns the number removed:

```
uint removedCount = coreManager.InterestManager.RemoveDefaultConditions();
```

It is the single call for a game that wants to arbitrate everything itself. Systems already started keep the fold they took at spawn and drop it when they next start, exactly as `TryRemoveCondition` does.

Showing an object only to its controller

Some objects belong to one player alone: an inventory, a private quest log, a hand of cards. `ControllerOnlyInterestCondition` shows a system only to the client that controls it, so the object never reaches any other client at all:

```
ControllerOnlyInterestCondition controllerOnlyInterestCondition = new();

inventorySystem.AddInterestCondition(controllerOnlyInterestCondition);
```

It is a Spawn condition: a client that does not control the object never holds it, rather than holding a stopped copy. When control moves, the next evaluation takes the object away from the previous controller and spawns it for the new one. A group is answered as one object, so every member of a group is shown to a client that controls any one of them. The condition holds no state, so one instance can be registered on any number of systems.

Register it per system. Registered globally with `AddCondition`, it scopes every object in the world, and a system no client controls is shown to no client, so every server-controlled object would disappear from every client.

Evaluation cadence

`EvaluationCadenceTicks` controls how many ticks pass between evaluations of any one system's conditions. It defaults to 5. Systems are staggered across the cadence by their `Id`, so each tick evaluates roughly one cadence-th of the world rather than spiking the whole of it onto one tick.

A joiner and a newly started system are exceptions: a client that just authenticated is evaluated against every started system, and a system that just started is evaluated against every authenticated client, both off the cadence and on the tick the event happens, rather than waiting for the stagger slot to come around.

Forcing a re-resolve

`RefreshInterest(NetworkSystem, isLevelOfDetailRemeasured)` re-resolves one system against every Connection at the head of the next pass, rather than when it next reaches its stagger slot:

```
coreManager.InterestManager.RefreshInterest(networkSystem, isLevelOfDetailRemeasured: true
);
```

Use it for a move the pass cannot anticipate: a teleport, a scene change, a reparent. Ordinary movement needs nothing, because the cadence is already measuring it. `isLevelOfDetailRemeasured` defaults to `true`; leave it set unless the object did not move.

`RefreshInterest(Connection)` is the other axis, for a change on the player's side rather than the object's: a client whose character was teleported, handed a different object to control, or moved to another scene.

```
coreManager.InterestManager.RefreshInterest(connection);
```

It re-resolves the connection against every evaluated system, not the whole world, so a world with nothing to resolve pays nothing for asking.

Registration timing

Registration as an observer waits on `Connection.CanReceiveState`, not just authentication. A client authenticates before its round trip time is known, and interest timing decisions are denominated in that figure, so a client authenticated before its link is measured is held back and registered the moment its round trip time is discovered — typically one round trip later, and immediately for a host's own client.

Spawn pacing

An unpaced world sends its biggest tick to the peer least able to take it

A tick's outbound to a connection is one MTU-segmented combined stream. A receiver can only read it once every segment has arrived, so losing one segment costs the whole tick. Redundancy multiplies that loss rather than avoiding it. An unpaced world spawns every system it can as soon as interest admits it, which means the biggest ticks — a client joining a large world, a scene binding, a bundle releasing a world's worth of withheld spawns — land exactly on the peer least able to absorb them: one still building its connection, with nothing yet to fall back on.

Spawn pacing turns that one enormous tick into several ordinary ones. Each is complete and recoverable on its own, and the receiver's instantiation cost spreads over several frames instead of landing in one.

This is a Pro feature.

The ceiling

`InterestManager.MaximumSpawnsPerTick` is the most `NetworkSystems` that may begin replicating to any one `Connection` on a single tick. Spawns beyond it are admitted on the ticks that follow, oldest first.

- `DefaultMaximumSpawnsPerTick` is `500`.
- `UnlimitedSpawnsPerTick` is `0` — set the ceiling to this and nothing is paced; every spawn is admitted on the tick it resolves, exactly as Free behaves.

The ceiling is per connection, because what it bounds is one peer's tick. It is measured per state flush, not per tick: everything admitted between two flushes rides the same combined stream regardless of how many loop steps it arrived over.

```
interestManager.MaximumSpawnsPerTick = 200;
```

Reading the queue

- `InterestManager.TotalSpawnsAdmitted` — spawns this peer has admitted to its clients, counting each system once per connection it was admitted to.
- `InterestManager.TotalSpawnsDeferred` — the running deferral count.
- `InterestManager.ConnectionsAwaitingSpawnAdmissionCount` — how many connections are currently owed at least one deferred spawn. Zero in a world that never reaches the ceiling, which is where a well-tuned ceiling spends nearly all of its time.
- `Connection.SystemsAwaitingSpawnAdmissionCount` — the backlog for one connection: it rises on a burst and falls by at most `MaximumSpawnsPerTick` each tick until that peer holds the world.
- `InterestManager.SpawnAdmissionStallWarningTicks` (`90`) — how many consecutive drains may admit nothing to a connection that is still owed spawns before the stall is logged. A backlog only stalls when the whole of every tick's budget is going to spawns resolved ahead of it — the world is spawning faster than the ceiling lets that peer receive.

`TotalSpawnsDeferred` counts deferrals, not deferred systems: a system is counted once per connection when it is first queued, not again on each pass it waits through. A world built out of `NetworkSystemGroups` reads high here by construction, because a grouped system is always queued so its group can be admitted whole, and so counts here even on a tick where it is admitted straight afterwards. Read

`TotalSpawnsDeferred` against `TotalSpawnsAdmitted` as a ratio of how hard the ceiling is working, not as a count of objects still waiting; use `ConnectionsAwaitingSpawnAdmissionCount` and `Connection.SystemsAwaitingSpawnAdmissionCount` for that.

Correctness rules

- **A group is admitted whole.** A `NetworkSystemGroup` is queued and admitted together, even when the ceiling has budget free for only some of its members. Splitting a group across ticks would hand the receiver half an object — the one state the interest cull deliberately never produces. If a group's member count exceeds `MaximumSpawnsPerTick`, it is still admitted whole rather than starved forever, and the manager logs a warning naming the group.
- **A peer's own controlled object never waits.** The system a connection controls skips the queue's arithmetic, including any group it belongs to — a client that cannot act on its own object for several ticks is worse than any tick pacing saves.
- **A peer that already holds the object is never withheld.** Re-priming an existing observer is not a spawn, so the ceiling never defers it.
- **A host's own client is never paced**, nor is an emulated peer — there is no wire and no segmented stream to protect for a loopback connection.
- **Nothing on the wire says a spawn waited.** A paced batch is an ordinary state tick; a receiver cannot tell a paced spawn from one that arrived late for any other reason. Pacing is a sender-side policy only.

Choosing a ceiling

Measure a real join or scene load's peak spawns-per-tick before and after setting `MaximumSpawnsPerTick`, rather than guessing a number. Set the ceiling above what ordinary play spawns at once, so pacing engages only on the bursts that would otherwise have cost a whole tick — a join, a scene binding, a bundle release — and leaves normal spawning untouched. If `ConnectionsAwaitingSpawnAdmissionCount` sits near zero outside those bursts, the ceiling is sized correctly; a stall warning in the log means it is set below what the world spawns and needs raising.

In Unity this ceiling is an inspector field

In the Unity integration, this ceiling is the **Maximum Spawns Per Tick** field on the Unity Interest Manager component — see Unity Interest Manager component.

Stopping versus culling

Using Unity? See Frozen or gone: what a stop does to the object

Interest can end replication for a connection two different ways, and they cost the receiver differently. `InterestEffect.Spawn` controls whether a `NetworkSystem` exists for a connection at all: while an unmet `Spawn` condition holds, the system does not spawn there, and one already spawned despawns, releasing the object exactly as an ordinary despawn does. `InterestEffect.Stop` is cheaper: while an unmet `Stop` condition holds, the system stops replicating to the connection, but the connection's engine object is retained. A distant object can stay visible while it stops costing anything to keep current, and it is only reclaimed if a `Spawn` condition later culls it too.

InterestMembership

`InterestMembership` is the resolved state of one `(NetworkSystem, Connection)` pair, as the stacked interest conditions settle it:

- `Streamed` — an ordinary delta observer, receiving the system's shared delta stream every tick. This is the deliberate default: a connection with no interest entry is streamed, so a world with no condition registered behaves exactly as it did before interest existed.
- `Stopped` — stopped for the connection by an unmet `Stop` condition; the connection holds no system, but its engine object was retained for a later re-entry.
- `Unspawned` — culled for the connection by an unmet `Spawn` condition; the connection holds neither the system nor its object.

Reclaiming a stopped object

`NetworkSystem.DespawnWhenStoppedEnabled` decides what a stop then costs the receiver, independent of whatever resolved the stop in the first place. Left false, a stopped system keeps its object on the receiver — cheap to undo, and the right answer for something a player is currently looking at. Set true, the system asks to be reclaimed rather than left frozen, which is the right answer for something a player has left behind: the object is a cost with nothing to show for it.

For a grouped object this is a vote, not a decision: the object is reclaimed only once *every* member is stopped and asking for it. One member that still wants to be heard, or one left to freeze rather than despawn, keeps the whole object where it is. An ungrouped system answers for itself, so its stop is a despawn outright.

The InterestStopResolving seam

Conditions can only restrict, so a rule authored for a whole object stops every system on it, and no system can hand itself back on its own. `NetworkSystem.InterestStopResolving` is the seam for the decision a condition cannot express: a subscriber can decline a stop no condition can undo, or impose one no condition asked for.

```
public delegate void InterestStopResolvingHandler(ref InterestStopContext interestStopCont
ext);
public event InterestStopResolvingHandler? InterestStopResolving;
```

`InterestStopContext` carries the pair being resolved:

```
public struct InterestStopContext
{
    public readonly NetworkSystem NetworkSystem;
    public readonly Connection Connection;
    public bool IsStopped;
}
```

It is passed by reference through every subscriber in subscription order, seeded with what the registered conditions resolved. Each subscriber sees what the one before it decided, and the last one to write settles the answer. A subscriber speaking for one system alone tests `NetworkSystem` first and leaves `IsStopped` untouched otherwise. Where `DespawnWhenStoppedEnabled` is set, declining a stop here keeps the whole object with that connection, since the group vote never counts it as stopped and asking.

When and for whom it is raised

`InterestStopResolving` is raised once per `(system, connection)` pair on the interest cadence, on the network loop thread, while the pair's spawn still stands and before the object's despawn is settled — the order that lets the answer feed into the despawn decision. Subscribers are held to what a condition is held to: pure, cheap reads that neither start nor stop systems, mutate observers, nor block.

It is never raised for an emulated connection or one sharing an in-memory transport. A host's own client is the one exception in the other direction: it is raised for while `InterestManager.HostInterestEnabled` is set, though a stop settled for it is only reported through `HostInterestMembership` rather than acted on.

Subscribing is itself a reason the pass walks the system. A `NetworkSystem` carrying no interest condition at all is normally skipped by the interest pass; adding an `InterestStopResolving` subscriber puts it back in front of the pass even with nothing else registered, and removing the last subscriber takes it back out.

Reacting when interest moves

Conditions and the stop seam decide interest; they must not act on it. To act when a pair actually moves (show a "player left your view" marker, start a server-side effect for a player who just gained an object, record analytics), subscribe to one of the two interest change events. Both are raised on the authority only, after the tick's state has been sent, on the network loop thread, so every move is settled when a handler reads it and a handler is free to do engine work: a system it starts goes live with the next tick.

For one object, subscribe to `NetworkSystem.InterestChanged`:

```
public delegate void InterestChangedHandler(Connection connection, InterestMembership prev
iousInterestMembership, InterestMembership interestMembership);
public event InterestChangedHandler? InterestChanged;

networkSystem.InterestChanged += OnInterestChanged;

void OnInterestChanged(Connection connection, InterestMembership previousInterestMembershi
p, InterestMembership interestMembership)
{
    if (interestMembership is InterestMembership.Unspawned)
    {
        // This client no longer has the object at all.
    }
}
```

A subclass of `NetworkSystem` can override `OnInterestChanged` instead of subscribing. The event is raised only for a move: an object that reaches a connection with no condition ever restricting it raises nothing, and neither does a system leaving the world or a connection leaving it, since their own events

already say so. It is never raised for a host's own client, which is reported through `HostInterestChanged` instead (see Host interest report).

For the whole world, subscribe to `InterestManager.InterestChanged`, which is raised once per tick that moved anything, with every move that tick:

```
public delegate void InterestChangesHandler(IReadOnlyList<InterestChange> interestChanges)
;
public event InterestChangesHandler? InterestChanged;

coreManager.InterestManager.InterestChanged += OnInterestChanges;

void OnInterestChanges(IReadOnlyList<InterestChange> interestChanges)
{
    for (int i = 0; i < interestChanges.Count; i++)
    {
        InterestChange interestChange = interestChanges[i];

        if (interestChange.InterestMembership is InterestMembership.Streamed)
        {
            // interestChange.Connection gained interestChange.NetworkSystem.
        }
    }
}
```

Each `InterestChange` carries the `NetworkSystem`, the `Connection`, the `PreviousInterestMembership` and the new `InterestMembership`. One call carries the whole tick, so a player joining and receiving a thousand objects costs one invocation rather than a thousand, which makes this the right event for a per-player visibility tracker. The list belongs to the engine and is cleared once your handler returns: read it inside the handler and copy whatever must outlive the call. Every recorded move is handed over, including one whose system has since left the world, so check `NetworkSystem.IsStarted` where that matters.

To read where a pair stands right now instead of waiting for a move, call `NetworkSystem.GetInterestMembership(connection)` on the authority. A connection the system holds no entry for reads `Streamed`.

Supplying positions to the interest system

Core ships no reader

The distance half of the interest system needs to know where a `NetworkSystem`'s object is, and core has no way to know that on its own. A distance condition reads positions through its own `TryReadSourcePosition` (see Distance interest conditions), so it does not use the reader on this page. Level of detail has no condition to ask, so it reads through the world's reader, registered on the manager. A bare `CoreManager`, with no engine integration attached, registers none: level of detail then logs an error once and every pair reads as Near. Scene-based interest and the distance conditions work with no world reader at all; only level of detail depends on one.

The interface

```
public interface IInterestPositionReader
{
    bool TryReadPosition(NetworkSystem networkSystem, out Vector3 position);
}
```

`Vector3` here is `System.Numerics.Vector3`, the type core replicates in. If your engine's own vector type differs, the reader is where you convert - core never sees the engine's type.

Return `false` rather than guessing for a system whose object has been destroyed or hasn't finished spawning. Don't reach for a stale or default position; an unmeasurable read must fail honestly so the caller can abstain instead of restricting on bad data.

This is the single thing an engine integration owes level of detail. Everything else - memoizing, the finiteness check, the once-per-pass discipline - lives on `InterestManager`.

Register the reader on the interest manager

```
public void SetPositionReader(IInterestPositionReader interestPositionReader);
```

Call `InterestManager.SetPositionReader` once, during setup, with your reader. Pass `null` to clear it as an integration tears down.

Why the manager holds it, not a condition

A condition is a per-prefab object that its system hands back to a pool when it's done. If the manager adopted a reader from a condition, it would keep that instance alive past its life and could end up reading positions through an object that has since been re-rented as something else. The position reader is a property of the world, not of any one object, so it's registered on the manager and outlives everything in it.

One read per pass

The manager caches each system's measured position for the length of one interest pass. A system read once by level-of-detail and again by a distance condition on the same tick is only read once, through whichever reader asks first; every repeated ask within that pass is answered from the cache. A failed read is cached too, so a destroyed or not-yet-spawned object doesn't get probed again and again within the same pass.

Because of this, your reader owes no memoization of its own. Write the raw read and let the manager do the rest.

A worked reader

The test harness backs its reader with a plain dictionary, which is a reasonable shape for anything that doesn't have a live scene graph to query:

```
public sealed class DictionaryInterestPositionReader : IInterestPositionReader
{
    private readonly Dictionary<NetworkSystem, Vector3> _positions = new();

    public void SetPosition(NetworkSystem networkSystem, Vector3 position) => _positions[n
etworkSystem] = position;

    public bool TryReadPosition(NetworkSystem networkSystem, out Vector3 position) => _pos
itions.TryGetValue(networkSystem, out position);
}
```

A system nobody has recorded a position for reports unmeasurable, which is the correct default: it's a failed read, not a zero position.

The Unity integration registers a position reader for you

The Unity integration registers a reader for you; you don't need to write one. See Limiting replication by distance for how to configure the distance conditions, which read positions on their own.

What scales and what does not

The four levers

There are four ways to spend less on replication, and they save different amounts because they cut at different points in the pipeline.

Don't send it at all. `InterestEffect.Spawn` governs whether a Connection is ever made an observer of a system in the first place. A condition that vetoes spawn keeps a peer from ever holding an object it has no reason to see. This is the cheapest lever there is: nothing is serialized, nothing is queued, nothing rides the wire.

Stop sending it. `InterestEffect.Stop` pauses replication to a peer that already holds the object without reclaiming it client-side. Cheaper than a fresh spawn later (no re-priming full state), but the object still exists on the peer and still occupies whatever bookkeeping a held object costs.

Send it less often. `SendInterval` paces a member's deltas: `SendInterval.Normal` sends every changed tick, a wider interval accumulates changes and rides at most one delta per interval. This only applies under `TransmissionMode.Interval`; a member on `TransmissionMode.Divine` governs its own rate through projection instead. A full serialization (spawn, resync, recovery, reconcile) always carries the current value regardless of the interval, so this lever never affects the moments that matter most.

Send it smaller. `TransmissionMode` and per-value compression level shrink each individual write without changing how often it happens. This is the smallest saving of the four, because it doesn't remove a send, a tick, or a peer's membership — it just makes the send that was already happening cost fewer bits.

Bit-level detail — exactly which encoding a value gets — is a measurement that feeds a choice among these four, not a fifth lever. You don't budget "smaller encoding" as its own line item; you use it to decide whether a lever above is worth pulling.

Three different bottlenecks

"Replication is too expensive" is not one problem. It's usually one of three, and each has its own fix.

Per-client egress. How many bytes one peer's connection pushes per second. This is what `TotalBytesSent` and the per-object/state-only bandwidth stat providers measure. The fix is fewer or smaller sends to that peer: spawn/stop culling first, interval and compression second.

Per-tick burst size. How much of that egress lands in a single tick rather than spread across several. A join, a scene load, a bundle release, or a peer walking back into range can put every system's priming full into one combined stream on one tick. A wide tick is all-or-nothing on an unreliable channel: one lost segment discards the whole tick, and a retained set resends at `EffectiveGlobalRedundancy` on top of that. `InterestManager.MaximumSpawnsPerTick` (default 500, 0 for unlimited) is the lever here — it spreads a burst of admissions across consecutive ticks instead of fixing the total bytes sent. It does nothing for steady-state egress; a world under the ceiling never touches it.

The O(systems ? connections) resolution pass. How much work the server itself does deciding who sees what, independent of what gets sent. Every registered `IInterestCondition` is evaluated per (system, Connection) pair on `InterestManager.EvaluationCadenceTicks`, and that cost scales with systems times connections regardless of how few of those pairs end up replicating. A world with no condition registered skips evaluation entirely and replicates broadcast-to-everyone. A world with conditions pays the evaluation cost even for pairs that resolve to "don't spawn." This is a CPU-time bottleneck, not a bandwidth one, and cadence/staggering is the only lever the engine currently has for it.

Fixing the wrong one does nothing: pacing spawns doesn't lower steady-state egress, and shrinking a delta's encoding doesn't stop a burst from landing in one tick.

Reading the cost you have

The bandwidth counters (`TotalBytesSent`, `TotalBytesReceived`, `TotalStatePacketPayloadBitsSent`, `TotalStatePacketPayloadBitsReceived`) and the per-object/state-only stat provider rows are how you read what a world is actually costing, before deciding which lever to pull. Every one of these figures covers the whole peer — total traffic for that connection, not a single object's line item — so a per-object number is a derived average, not something the wire counts directly. Divide by object count yourself, or use the shipped per-object stat provider row, which already does it. How those counters are wired up and sampled is covered under Diagnostics; this page is about which lever their numbers point you toward.

Burst versus steady state

A spike on a join and a steady per-tick overspend are different failures with different fixes. A join spike is bounded and temporary: it costs frames and, if unpaced, risks a lost-segment discard on a wide tick — the fix is `MaximumSpawnsPerTick`, and it doesn't touch what the world costs once everyone has finished joining. A steady overspend is what egress and resolution-pass measurements are for, and its fix is spawn/stop culling or interval/compression tuning, not pacing.

Redundancy is why a lost tick during a burst is worse than it looks. A retained set is resent `EffectiveGlobalRedundancy` times on the ticks that follow, so bytes a burst already spent get paid again on top of that window's ordinary traffic. A burst that loses a segment doesn't just retry once — it pays the full width of the burst again, redundancy times over, while steady-state traffic is running alongside it. This is the concrete reason pacing a burst is worth doing even though it doesn't move the steady-state number: a wide, unpaced tick multiplies its own cost on loss, a paced one doesn't.

What the engine does not have

Don't design a world around any of these — none of them exist:

- **No per-connection byte-budget scheduler.** There's no cap on how many bytes a peer's connection may spend per tick, and no priority or staleness scheduling deciding what gets sent first when a peer is behind. The spatial grid and per-connection spawn pacing exist; a bandwidth cap does not.
- **No per-object send priority.** Nothing lets you mark one object as more urgent than another when a peer is bandwidth-constrained. Every eligible object is served the same way.
- **No interest enter/exit callbacks.** Game code isn't notified when a peer gains or loses interest in an object. If you need to react to that transition, you have to infer it yourself.
- **No broadphase seam for interest.** There's no hook letting an integration hand the interest pass a pre-narrowed candidate set; evaluation is the O(systems ? connections) pass described above for every world. The spatial grid narrowing that exists today is internal to distance/grid conditions, not an integration seam.

Edition effects on the budget

Several of the levers above only exist in Pro: distance and spatial-grid interest culling (the conditions that drive `InterestEffect.Spawn`/`Stop`), level-of-detail interest, and per-connection spawn pacing (`MaximumSpawnsPerTick`) are all Pro-only. Free still has the four-lever framework and the broadcast-to-everyone default, but the conditions that make spawn/stop culling and burst pacing possible are Pro.

Compression is the other edition split. Free writes every value at its exact length; Pro adds bit-packing brackets that pack values more tightly. This is the one gate that forks the wire format itself — a Free and Pro peer serialize differently at this layer — so it is not something you can toggle per-world; it is fixed by which edition built the peer. This page states that the brackets exist and are Pro-gated; how the encoding works is out of scope here.

A worked budget for a small world

Say a world has 50 objects, each with one `NetworkMember` sending a full-precision value every tick at `SendInterval.Normal`, replicated to 20 connections.

Steady-state egress, ignoring redundancy and headers: 50 objects ? 20 connections = 1,000 per-peer-per-tick sends. If each delta costs roughly the same regardless of which peer receives it, the total is 20 connections' worth of "50 objects' deltas per tick" — the per-connection egress is what scales with object count, not the total across all connections, because each connection only ever carries its own 50-object slice.

Resolution-pass cost, if you add one distance condition: 50 systems ? 20 connections = 1,000 pair evaluations every `EvaluationCadenceTicks` ticks. At a cadence of 1 that's 1,000 evaluations/tick; staggering it across a cadence of, say, 10 spreads that to roughly 100 evaluations on any given tick, at the cost of resolving each pair up to 10 ticks stale.

Burst size on a join: a joining peer is registered against every started system at once — all 50 objects' priming fulls land on one combined stream unless spawn pacing caps it. At the default `MaximumSpawnsPerTick` of 500, a 50-object world never touches the ceiling; the burst is whatever 50 fulls cost, in one tick, with no pacing needed. That ceiling only starts mattering once system count times joining-peer count starts approaching it — this world is nowhere close.

The arithmetic is the budget: egress scales with objects ? connections that can see them, the resolution pass scales with objects ? connections regardless of visibility, and burst size scales with objects per joining peer against the pacing ceiling. Three different multiplications, three different levers, and a 50-object/20-connection world doesn't need any of them pulled yet.

Writing an interest condition

What a condition does

An `IInterestCondition` decides one aspect of what a `Connection` receives for a `NetworkSystem`, so replication cost scales with what each client can actually perceive rather than with the size of the world. A condition registers globally on the `InterestManager`, covering every system spawned from then on, or on a single `NetworkSystem` through `AddInterestCondition`. Conditions stack: each is evaluated once per (system, Connection) pair on the manager's evaluation cadence, and the pair's resolution folds every stacked opinion, most-restrictive-wins per effect. With no condition registered, replication behaves exactly as it did before interest existed: every started system, to every authenticated client, every tick.

The interface has exactly two members.

```
public interface IInterestCondition
{
    InterestEffect Capabilities { get; }

    InterestResult Evaluate(NetworkSystem networkSystem, Connection connection);
}
```

Capabilities: a promise, not a filter

`Capabilities` is declared once and states every effect this condition can ever return an opinion on. This is a promise, not a filter: an effect left out is never asked, so a condition that returns an opinion it did not declare has that opinion silently ignored. Declare every effect the implementation can ever return, and only those — declaring one the condition never speaks to costs an evaluation per pair for nothing. `InterestEffect.None` declares a condition that speaks to nothing, and is rejected at registration.

`InterestEffect` is a `[Flags]` enum with two members:

- `Spawn` — controls whether the system exists at all for the connection. While unmet, the system does not spawn there, and one already spawned despawns; the object is released on the receiver exactly as an ordinary despawn releases it.
- `Stop` — controls whether the system stays started for the connection. While unmet, the system stops for the connection but its engine object is retained, so a distant object can stay visible while it stops costing replication; the object is reclaimed only when a `Spawn` condition culls it too.

A condition that only ever has an opinion about spawning declares `InterestEffect.Spawn`; one that speaks to both declares `InterestEffect.Spawn | InterestEffect.Stop`.

Building a verdict

`Evaluate` returns an `InterestResult`, an independent opinion on any subset of the effects the condition declares, so one condition can veto spawn and stop from a single measurement.

```
public readonly struct InterestResult
{
    public readonly InterestOpinion SpawnOpinion;
    public readonly InterestOpinion StopOpinion;

    public static InterestResult None => default;

    public InterestResult WithSpawn(bool isMet);
    public InterestResult WithStop(bool isMet);

    public static InterestResult Spawn(bool isMet);
    public static InterestResult Stop(bool isMet);
}
```

- `InterestResult.Spawn(isMet)` — a spawn-only verdict; an unmet one alone culls the system for the pair.
- `InterestResult.Stop(isMet)` — a stop-only verdict; an unmet one stops the system for the pair while keeping its engine object.
- `WithSpawn(isMet)` / `WithStop(isMet)` — combinators that set the other axis on a copy, for a condition that speaks to both effects in one evaluation.
- `InterestResult.None` — a verdict abstaining on every effect, imposing no restriction and fighting no other condition. Return this for a pair the condition cannot measure, rather than guessing a false restriction.

InterestOpinion: the tri-state

Each axis of an `InterestResult` carries an `InterestOpinion`:

```
public enum InterestOpinion : byte
{
    None = 0,
    Unmet = 1,
    Met = 2,
}
```

`None` is the fold identity: the resolver ANDs every stacked opinion, so an abstaining condition never restricts. Only `Unmet` restricts; `Met` and `None` are indistinguishable to the fold and differ only in what the condition means to say. Stacked conditions fold most-restrictive-wins per effect — any unmet `Spawn` opinion from any condition keeps the system away from the connection, and any unmet `Stop` opinion stops it there, regardless of what the others said.

The purity contract

`Evaluate` runs on the network loop thread, inside the serialization pass, on the manager's evaluation cadence rather than every tick. It must be a pure read: it must not start or stop systems, mutate observers, or block.

A condition can only ever restrict

None of this can grant a connection more than the broadcast-to-everyone default. A condition narrows what a connection receives; there is no opinion that adds a connection beyond what it would already get with no conditions registered at all.

A worked condition

From the test suite, a condition that always approves spawn — the shape a non-distance rule (scene membership, team, bundle ownership) takes:

```
private sealed class TestAlwaysMetInterestCondition : IInterestCondition
{
    public InterestEffect Capabilities => InterestEffect.Spawn;

    public InterestResult Evaluate(NetworkSystem networkSystem, Connection connection) =>
InterestResult.Spawn(isMet: true);
}
```

And a condition driven by an external rule, evaluated per pair against a caller-supplied policy:

```
private sealed class ProbeInterestCondition : IInterestCondition
{
    public InterestEffect Capabilities { get; }
    private readonly Func<NetworkSystem, Connection, InterestResult> _evaluator;

    public ProbeInterestCondition(Func<NetworkSystem, Connection, InterestResult> evaluato
r, InterestEffect capabilities = InterestEffect.Spawn | InterestEffect.Stop)
    {
        _evaluator = evaluator;
        Capabilities = capabilities;
    }

    public InterestResult Evaluate(NetworkSystem networkSystem, Connection connection) =>
_evaluator(networkSystem, connection);
}

// Cull one connection from a system while every other connection keeps receiving it:
networkSystem.AddInterestCondition(new ProbeInterestCondition(
    (system, connection) => InterestResult.Spawn(connection != culledConnection)));
```

Register the condition on a single system with `NetworkSystem.AddInterestCondition`, or globally with `InterestManager.AddCondition` to cover every system spawned from then on.

The Unity page covers authoring the same condition in the inspector

A condition authored on a prefab derives from `AuthoredInterestCondition`, a public abstract `[Serializable]` class, and builds the runtime `IInterestCondition` from its `CreateCondition` method. Subclasses are discovered automatically by the add-rule menu; nothing has to be registered by hand.

Capturing a shot

Using Unity? See Firing and judging a shot.

Capture on the frame the shot is fired

The shooter's half of lag compensation is one call. On the frame the player fires, capture the shot:

```
LagCompensatedShot lagCompensatedShot = coreManager.LagCompensationManager.CaptureShot(ori
gin, direction);
```

`CaptureShot(Vector3 origin, Vector3 direction)` returns a `LagCompensatedShot` carrying the origin and direction unchanged and the view `CaptureView()` returns at that moment. The direction does not need to be of unit length. Taking the ray and the view in the same call means they can never come from different frames.

Capture on the firing frame, not when the next tick's client state is built. A tick spans several rendered frames, each drawing a different blend between two server ticks, and the view records which blend was on screen. A view taken a frame later names a different moment from the one the player aimed at.

Send the shot whole

Put the `LagCompensatedShot` itself in a message, a remote call argument or a client state member, and send it to the server:

```
using Nucleus.Managers.LagCompensation;
using Nucleus.Managers.Messages;
using Nucleus.Serializers;

[NetworkType]
public struct ShotMessage : IMessage
{
    public LagCompensatedShot Shot;
}

if (coreManager.TransportManager.TryGetServerConnection(out Connection serverConnection))
    serverConnection.SendMessage(Channel.Reliable, new ShotMessage { Shot = lagCompensated
Shot });
```

Do not copy the origin and direction into plain `Vector3` fields. A plain `Vector3` in a message is rounded to a thousandth per component, which can turn a unit direction by about 0.9 milliradians: roughly 4 cm at 50 m and 9 cm at 100 m, enough to lose a headshot at range with nothing to show why. A `LagCompensatedShot` is written through hand-written serializers, `LagCompensatedShotSerializers`, which the generator adopts for every field and argument of the type in every assembly, and which send each component of the origin and direction as its raw 32 bits. A whole shot costs 24 bytes for the ray plus a few for the view. An infinity or NaN in either vector arrives as zero, as every raw float the engine reads does, so no peer can put one into the server's queries.

`LagCompensatedShot` is compared bit for bit, view included, so a member holding one sees a new shot as a change.

On the server, the shot feeds a query directly

The shot's three fields, `View`, `Origin` and `Direction`, are the arguments a query takes:

```
LagCompensatedShot shot = shotMessage.Shot;

bool isHit = coreManager.LagCompensationManager.TryRaycast(messageContext.SenderConnection
, shot.View, shot.Origin, shot.Direction, maximumDistance, hitboxLayerMask, out LagCompens
atedHit lagCompensatedHit);
```

See Judging a shot with rewound queries for every query and what a hit reports.

A 2D shot captures with a Z of zero

A 2D game captures with a Z of zero and reads the shot back through its 2D properties, `Origin2D` and `Direction2D`, which the 2D queries take:

```
LagCompensatedShot lagCompensatedShot = coreManager.LagCompensationManager.CaptureShot(new
 Vector3(origin2D, 0f), new Vector3(direction2D, 0f));

bool isHit = coreManager.LagCompensationManager.TryRaycast2D(messageContext.SenderConnecti
on, shot.View, shot.Origin2D, shot.Direction2D, maximumDistance, hitboxLayerMask, out LagC
ompensatedHit lagCompensatedHit);
```

Capturing a view alone

A shot that is not a ray, such as a melee swing whose volume the server builds itself, only needs the view. `CaptureView()` returns the `LagCompensationView` on screen right now, and the server passes it to an overlap query. A `LagCompensationView` is a `[NetworkType]`, so it can ride a message, a remote call or a client state member directly; inside client state it changes only on ticks that fire, so delta compression makes it nearly free on every other tick. Its fraction is quantized to a byte, a resolution of 1/255 of a tick.

What a server or host captures

On a server or host, `CaptureView` returns the newest recorded tick with a fraction of zero, because a server's screen shows the present. A host's own shots are therefore judged against the present, which is what its player saw.

An empty view, `default`, has a `Tick` of `NetworkLoopManager.UnsetTick` and is judged as the present. A client captures an empty view before any state has arrived, and a free build always does.

How lag compensation works

> **Using Unity?** See Lag compensation in Unity.

A client aims at the past

A client draws the world a little behind the server. It holds received state back for interpolation, and every packet it draws already spent half a round trip reaching it. When the player fires at a moving target, the server has already moved that target on, and by the time the shot arrives it has moved again. Judge the shot against the server's present and a player who aimed perfectly misses whatever is moving.

Lag compensation lets the server judge a shot against the world as the shooter saw it. It is a Pro feature; see What a free build does below.

Nothing in the scene is moved

`LagCompensationManager`, reached as `CoreManager.LagCompensationManager`, does not rewind objects, colliders or the physics engine. It keeps its own history and does its own geometry:

Record. Each networked object that can be hit registers a hitbox root: a fixed list of hitbox shapes and an `IHitboxPoseSource` that reports where they are. While this peer runs a server, the manager asks every root for its poses once per tick and keeps them for `MaximumRewindMilliseconds`.

Capture. On the frame the shooter fires, it calls `CaptureShot`, which returns a `LagCompensatedShot`: the ray exactly as drawn, plus a `LagCompensationView` naming the server tick on screen and how far the screen had blended toward the next one. The shooter sends the shot to the server in a message or remote call.

Judge. The server passes the shot's view, origin and direction to a query such as `TryRaycast`. The query rebuilds every hitbox between the two recorded ticks the view names, blended exactly as the shooter's screen blended them, and tests the shot against those rebuilt shapes.

Because only the history is read, a query can run at any moment on the server, any number of times, without disturbing the live simulation.

Recording happens once per tick, after state is written

The manager records on the callbacks of `NetworkLoopSteps.LateSerialize`, which run only on frames that run a tick and after that tick's state has been written. The pose recorded for tick N is therefore the pose tick N's state carried to clients, which is exactly what a client will later draw for tick N.

Only a server records. A registration on a client is refused without an error, so integration code can register on every peer and let the server keep what it needs. When this peer's server stops, every history is released.

Each tick, every root's pose source is called on the network loop. A source that throws is logged, loses only its own slot for that tick, and every other root is still recorded.

A view is two ticks and a blend

A `LagCompensationView` is a `Tick` and a `Fraction`. `Tick` is the older of two server ticks the screen was blending between, and `Fraction` is how far the screen had blended toward the tick after it, from 0 (exactly `Tick`) to 255 (exactly the next tick). On a client it is read from the pair of server ticks the received state is being presented between and this frame's subtick fraction. On a server or host it is the newest recorded tick with a fraction of zero, because a server's screen shows the present.

A tick spans several rendered frames, and every frame draws a different blend, which is why a shot must be captured on the frame it is fired rather than when the next tick's client state is built. See Capturing a shot.

Blended hitboxes are bounded by every recorded pose

A query first rules whole roots out with a cheap test against a bounding sphere, and only rebuilds the hitboxes of a root the query can reach. Each tick's bounding sphere covers every hitbox the root registered, active or not. When a view falls between two ticks, the two ticks' spheres are merged, so the covering sphere holds every hitbox drawn anywhere between its two poses.

That matters because a blended view takes the active hitboxes from whichever of the two ticks is nearer, while drawing each hitbox between both ticks' poses. A hitbox inactive on the older tick and active on the newer one is drawn partly at its older pose; a sphere around the active hitboxes alone would leave that pose out and cull a shot that hits it. For the same reason a pose source writes every hitbox's pose every tick, active or not.

A fraction of exactly 0 or 255 reads that tick's poses as stored and blends nothing.

Targets sent on an interval are judged where they were drawn

A position sent every tick is drawn sweeping across one tick, so the shooter's view is right for it as it stands. A position sent with `TransmissionMode.Interval` across a window of several ticks is drawn sweeping across that whole window, which puts it that many ticks less one behind. Both peers build the member with the same interval, so the server shifts the view back for that target by itself without being told. A target with no `TransformComponent`, or one that has stopped, is judged at the view as reported.

Grouped objects are recorded once

A grouped object links several systems, but its hitboxes belong to the object. Recording them under every member would let one shot hit each hitbox once per member, so an integration registers a grouped object's hitboxes once, under one member. The Unity integration prefers the member carrying the object's `TransformComponent`, because the interval shift above needs that member's transform. When a query is told to ignore an object, naming any member of a group skips the whole group, whichever member the hitboxes were recorded under.

A stopped target stays hittable for the rest of the window

A root whose system stops, or returns to its pool, stops recording, but its history stays readable until its newest recorded tick leaves the rewind window. A target killed a moment before a shot arrives can still be hit by that shot. Its `LagCompensatedHit.NetworkSystem` is then `null`, and `LagCompensatedHit.SystemId` still names it.

A record belongs to one life of its system. A pooled system handed out again, or restarted under the same Id, starts a fresh record, so it can never be found through an old history.

Spawn compensation is the other half

Lag compensation judges a shot against the past. Spawn compensation solves the opposite half: it brings a freshly spawned object, such as a projectile, up to the present through the latency it lost on the way. Its cap, `MaximumSpawnCompensationMilliseconds`, lives on `LagCompensationManager` for that reason.

What a free build does

Every name on these pages exists in both editions, so the same code compiles against either. In a free build every `TryRegisterHitboxRoot` call is refused, with one logged error for the whole session, nothing is

recorded, `CaptureView` returns an empty view, `CaptureShot` still carries its ray exactly with an empty view, and every query finds nothing.

Lag Compensation

Lag compensation is a Pro feature. Pages in this category:

- How lag compensation works
- Registering hitboxes
- Capturing a shot
- Judging a shot with rewound queries
- The rewind window and the view clamp

Registering hitboxes

Using Unity? See Making an object hittable, where `NetworkHitboxRoot` does all of this for you.

A root is one object's set of hitboxes

The engine knows nothing about transforms, meshes or colliders. A networked object that can be hit registers one **hitbox root** with `LagCompensationManager.TryRegisterHitboxRoot`, giving it three things:

- the `NetworkSystem` the hitboxes belong to, one root per system;
- an `IHitboxPoseSource` that writes the hitboxes' world poses whenever the server records a tick;
- a `HitboxDefinition` for each hitbox, the fixed shape in its own unscaled frame, in the order the pose source writes them.

```
HitboxDefinition[] hitboxDefinitions =
[
    new(HitboxShape.Capsule, radius: 0.3f, size: Vector3.Zero, height: 1.8f, layer: 8),
    new(HitboxShape.Sphere, radius: 0.15f, size: Vector3.Zero, height: 0f, layer: 9),
];

bool isRegistered = coreManager.LagCompensationManager.TryRegisterHitboxRoot(networkSystem
, soldierHitboxes, hitboxDefinitions);
```

The definitions are copied, so the caller may reuse the array. A root holds between 1 and `LagCompensationManager.MaximumHitboxesPerRoot` (64) hitboxes, which is the width of the per-tick active mask.

`TryRegisterHitboxRoot` returns `false`, with a logged error, when the system or pose source is `null`, when the hitbox count is out of range, or when the system already has a root. It returns `false` without an error when this peer is not running a server, because only a server records; register once the server is running, for example when the system starts on the server. In a free build it returns `false` and logs once that lag compensation is not available.

How far back a root's history reaches is sized from `MaximumRewindMilliseconds` when the root registers; see The rewind window and the view clamp.

Describing a hitbox

`HitboxShape` has three 3D shapes and three 2D shapes, each of which a query can test exactly:

Shape	Described by
`Sphere`	`Radius`
`Box`	`Size`, the full size on each axis
`Capsule`	`Radius` and `Height`, the full height including both rounded ends, along the hitbox's own up axis
`Circle`	`Radius`, in the XY plane
`Box2D`	`Size`, reading X and Y only, in the XY plane
`Capsule2D`	`Radius` and `Height`, along the hitbox's own up axis, in the XY plane

`HitboxDefinition`'s constructor is `HitboxDefinition(HitboxShape shape, float radius, Vector3 size, float height, byte layer = 0)`; a field a shape does not use is ignored. `Layer` runs from 0 to 31, and a larger value wraps into that range. A query's layer mask, one bit per layer, chooses which layers it can hit, so a game can keep heads, bodies and shields on separate layers.

Scale is not part of the definition. It is recorded with every pose, so a hitbox whose object grows or shrinks is judged at the size it had on the tick being asked about.

Writing poses

`IHitboxPoseSource` has one method:

```
public ulong WriteHitboxPoses(Span<HitboxPose> hitboxPoses);
```

It is called on the network loop thread, once per recorded tick, with one `HitboxPose` slot per registered hitbox in registration order. It must not allocate. For each hitbox, write:

- `Position`, the world position of the hitbox's center, including any offset from whatever it follows;
- `Rotation`, the world rotation; a capsule's length runs along its own up axis, so a capsule lying along another axis turns its recorded rotation onto it;
- `Scale`, the absolute world scale in the hitbox's own frame; for a capsule, Y is the scale along its length and X and Z the two across it.

Return a mask with bit `i` set when hitbox `i` is active on this tick. An inactive hitbox is recorded but never hit on that tick. Write every hitbox's pose every tick even when it is inactive: a shot judged between two ticks blends both ticks' poses, and the bounding test that culls roots covers inactive hitboxes for that reason.

```
using System;
using System.Numerics;
using Nucleus.Managers.LagCompensation;

public sealed class SoldierHitboxes : IHitboxPoseSource
{
    public Vector3 Position;
    public Quaternion Rotation = Quaternion.Identity;
    public bool IsDowned;

    public ulong WriteHitboxPoses(Span<HitboxPose> hitboxPoses)
    {
        // The body capsule, centered 0.9 m above the soldier's feet.
        hitboxPoses[0].Position = Position + Vector3.Transform(new Vector3(0f, 0.9f, 0f),
Rotation);
        hitboxPoses[0].Rotation = Rotation;
        hitboxPoses[0].Scale = Vector3.One;

        // The head sphere, 1.65 m up.
        hitboxPoses[1].Position = Position + Vector3.Transform(new Vector3(0f, 1.65f, 0f),
 Rotation);
        hitboxPoses[1].Rotation = Rotation;
        hitboxPoses[1].Scale = Vector3.One;

        // A downed soldier is recorded but cannot be hit.
        return IsDowned ? 0UL : 0b11UL;
    }
}
```

A source that throws is logged and loses only its own slot for that tick; every other root is still recorded.

2D hitboxes lie in the XY plane

A 2D game registers `Circle`, `Box2D` and `Capsule2D` hitboxes and judges them with the 2D queries. A 2D shape is the slice the plane Z = 0 cuts through the matching 3D shape, so it is tested with the same exact math. Write its pose in that plane: a `Position` whose Z is zero and a `Rotation` about the Z axis only. It reads the X and Y of its `Scale`, and for a 2D capsule X is the one scale across it.

2D and 3D hitboxes are kept apart the way Unity keeps its 2D and 3D physics apart: a 3D query never touches a 2D hitbox, and a 2D query touches 3D hitboxes only when the game allows it. See 2D queries.

Ending a recording

A system that stops or returns to its pool is unregistered by the engine. Call `TryUnregisterHitboxRoot(networkSystem)` only to stop recording a system that keeps running; it returns `true` when the system had a root. Either way, what was already recorded stays readable until it leaves the rewind window, so a shot fired just before the change is still judged fairly.

A grouped object registers once

A grouped object's members share the object's hitboxes. Register one root for the whole object, under one member, rather than one per member, or a single shot will report each hitbox once for every root that recorded it. The member carrying the object's `TransformComponent` is the right one when there is a choice, because a target whose position is sent on an interval is judged where its transform was drawn, and only that member knows the interval. See How lag compensation works.

Judging a shot with rewound queries

Using Unity? See Firing and judging a shot.

Every query takes a shooter and a view

The server judges a shot with one of `LagCompensationManager`'s queries. Each one rebuilds every registered hitbox where the shooter saw it and tests the shot against those rebuilt shapes. They all begin with the same two arguments:

- `shooterConnection`, the `Connection` that fired, normally `MessageContext.SenderConnection`. Its view is checked against what the server could have shown it, and hitboxes on any object it controls are skipped, so a shot never hits its own shooter. Pass `null` for a shot the server fires itself.
- `lagCompensationView`, the view the shooter captured when it fired, normally `LagCompensatedShot.View`. An empty view, `default`, means the present.

```
private void OnShotReceived(in MessageContext messageContext, ShotMessage shotMessage)
{
    if (!_coreManager.TransportManager.IsServerStarted)
        return;

    LagCompensatedShot shot = shotMessage.Shot;

    if (_coreManager.LagCompensationManager.TryRaycast(messageContext.SenderConnection, sh
ot.View, shot.Origin, shot.Direction, MaximumDistance, HitboxLayerMask, out LagCompensated
Hit lagCompensatedHit))
    {
        // Apply damage to lagCompensatedHit.SystemId, scaled by which hitbox lagCompensat
edHit.HitboxIndex names.
    }
}
```

The layer mask has one bit per hitbox layer. A query never moves anything and allocates nothing, so it can run as often as the game needs.

The 3D queries

Query	Returns	Use it for
`TryRaycast`	the nearest hit	a hitscan shot
`RaycastAll`	every hit, nearest first	a piercing shot
`TrySphereCast`	the first hitbox a swept sphere touches	a thick shot that stops at the first thing it touches
`SphereCastAll`	every hitbox a swept sphere touches, nearest first	a thick piercing shot
`OverlapSphere`	every hitbox a sphere overlaps, nearest first	an explosion
`OverlapCapsule`	every hitbox a capsule overlaps, nearest first	a melee swing

`OverlapBox`	every hitbox a rotated box overlaps, nearest first	an area attack

The casts take their arguments in the order Unity's physics queries do: origin, then radius for a sphere cast, then direction, then maximum distance. A direction does not need to be of unit length. A query with no direction, no reach or a negative radius finds nothing.

The `All` forms and the overlaps take a `List<LagCompensatedHit>`, clear it, fill it and return the count. Keep one list and reuse it across calls so nothing is allocated:

```
private readonly List<LagCompensatedHit> _lagCompensatedHits = new();

int hitCount = lagCompensationManager.OverlapSphere(shooterConnection, view, blastCenter,
BlastRadius, HitboxLayerMask, _lagCompensatedHits);

for (int i = 0; i < hitCount; i++)
{
    float falloff = 1f - _lagCompensatedHits[i].Distance / BlastRadius;
    // Apply damage scaled by falloff to _lagCompensatedHits[i].SystemId.
}
```

Every geometric test is exact. A swept sphere touches a hitbox exactly where a ray from its center enters the hitbox grown by the radius on every side.

What a hit reports

`LagCompensatedHit` carries:

- `NetworkSystem`, the system the hitbox belongs to, or `null` when that system has stopped since the judged tick;
- `SystemId`, the Id the system had, which stays readable after it stops;
- `HitboxIndex`, the hitbox's index within its root, in registration order;
- `Layer`, the hitbox's layer;
- `Distance`, `Point` and `Normal`, the first contact.

What the contact means depends on the query:

- **Ray.** `Distance` is from the origin to the hit and `Point` is where the ray hit.
- **Cast.** `Distance` is how far the swept shape's center travelled before it touched the hitbox, and `Point` is the point on the hitbox it first touched.
- **Overlap.** `Point` is where the query first touches the hitbox as it fills out from its core: the hitbox's nearest point to a sphere's center or a capsule's segment, or the point of the hitbox inside a box nearest the box's center. `Distance` is how far that point lies from the core, never more than the radius for a sphere or capsule, so an explosion reads its falloff straight from it. When the core itself reaches the hitbox, `Distance` is zero and `Point` is the query's own origin: the sphere's or box's center, or where a capsule's segment, walked from its start, first enters the hitbox. Order those zero-distance capsule hits along the swing by the distance from the segment's start to each `Point`.

`Normal` is the hitbox's outward surface normal at the point, facing the query, and is never zero, so its negation always pushes the hitbox away from the query. At a box's edge or corner it is the direction there that faces the query.

An overlap's `Point` is a contact, not the hitbox's position. Code that needs where the hitbox itself was should look the hitbox up through its root.

Ignoring an object

Every query, 3D and 2D, has an overload that takes a `NetworkSystem` `ignoredNetworkSystem` straight after the shooter. Pass the root of the object to ignore, normally the shooter's own root object, or `null` to ignore nothing. Everything nested under it is ignored too:

```
bool isHit = lagCompensationManager.TryRaycast(shooterConnection: null, turretSystem, view
, muzzle, aim, MaximumDistance, HitboxLayerMask, out LagCompensatedHit lagCompensatedHit);
```

The shooter's connection already skips what that connection controls, so the overload is for the cases it cannot cover: a host firing for its own avatar, which passes a `null` shooter, or a turret or NPC the server fires for itself. Naming any member of a grouped object skips the whole object, because a grouped object records its hitboxes under one member, which need not be the member you hold. A history that has ended keeps the group its system had, so a shot judged just after the object despawned still skips it.

Every object parented under the ignored one with `SystemManager.EnsureSetSystemParent`, at any depth, is skipped as well, so a weapon or shield a character carries never blocks that character's own shot. This is why you pass the root: naming a nested object skips only that object and what is nested under it, and leaves its parent hittable. Detaching an object ends the skip from that tick on. One case is not covered: a nested object that despawned before the shot is judged has no live parent left to follow, so only its own system and group are matched.

A ray can start inside a hitbox

`CanQueriesStartInHitboxes` is on by default, matching Unity's `Physics2D.queriesStartInColliders`. A ray whose origin lies inside a hitbox, or a sphere or circle cast that starts touching one, hits it at distance zero, with its own origin as the point and a normal facing back along the query, and that hit is the nearest one.

Turn it off and a ray or cast ignores every hitbox it starts inside or touching: that hitbox is neither reported nor taken as the nearest hit, and the query goes on to whatever lies beyond. That suits a muzzle that sits inside some other object's hitbox. Overlaps are not affected; they still report every hitbox their volume touches, including one their center lies inside. The setting is read once per query, so it can change at any time.

```
coreManager.LagCompensationManager.CanQueriesStartInHitboxes = false;
```

2D queries

The 2D queries take `System.Numerics.Vector2` and run in the plane Z = 0, where `Circle`, `Box2D` and `Capsule2D` hitboxes lie:

Query	Mirrors
`TryRaycast2D`, `RaycastAll2D`	`TryRaycast`, `RaycastAll`
`TryCircleCast2D`, `CircleCastAll2D`	`TrySphereCast`, `SphereCastAll`, with arguments in `Physics2D.CircleCast`'s order
`OverlapCircle`	`OverlapSphere`
`OverlapCapsule2D`	`OverlapCapsule`
`OverlapBox2D`	`OverlapBox`, with the rectangle turned by `angleRadians` about Z, counterclockwise

`OverlapBox2D` takes its angle in radians; a Unity angle in degrees is multiplied by `Mathf.Deg2Rad` first. A

hit on a 2D hitbox has a `Point` and `Normal` with a Z of zero.

2D and 3D hitboxes are kept apart the way Unity keeps its 2D and 3D physics apart. A 2D query touches only circles, 2D boxes and 2D capsules, and a 3D query only spheres, boxes and capsules. A 3D query never touches a 2D hitbox, which has no depth for it to meet, so a 2D game that calls `TryRaycast` finds nothing.

A game that mixes the two can set `Can2DQueriesHit3DHitboxes` to let its 2D queries reach 3D hitboxes as well. A 2D ray and `OverlapBox2D` have no depth, so they touch a 3D hitbox only where the plane Z = 0 cuts it; an `OverlapBox2D` hit's `Point` keeps a Z of zero while its `Normal` is the 3D hitbox's own and can tilt out of the plane. A circle, a 2D capsule and a circle cast have their radius in every direction, so they reach a 3D hitbox as a ball or capsule lying in the plane, up to their radius either side of it. Like the setting above, it is read on every query.

Walls are not in the history

Only hitboxes are recorded, so a rewound query sees no walls. Clip the shot against the present-time world first, with your own engine's raycast against level geometry, and pass the distance to the nearest wall as the query's maximum distance. Make sure that raycast passes over the colliders of the networked objects themselves: a present-time collider on a target running toward the shooter stands nearer than its rewound hitbox and would turn a clean hit into a miss. The Unity integration ships this check as `LagCompensationWalls`.

The rewind window and the view clamp

Using Unity? See UnityLagCompensationManager.

How far back a shot can be judged

`LagCompensationManager.MaximumRewindMilliseconds` is how far back poses are kept and a shot may be judged. It defaults to `LagCompensationManager.DefaultMaximumRewindMilliseconds`, 500 milliseconds.

```
coreManager.LagCompensationManager.MaximumRewindMilliseconds = 300;
```

It is read when a root registers, which sizes that root's history to the window at the current tick rate plus two ticks, so a change applies to roots registered afterwards. Set it before objects spawn. The oldest tick a query will answer for is worked out from the same setting on every query.

An honest client's view stays inside the window unless its latency exceeds it. Size the window to the slowest link the game is willing to favor the shooter for: a longer window lets a player on a slow link land shots on targets that have long since moved on, which the targets experience as being hit behind cover.

A view the server could not have shown is clamped

A view older than the window, or newer than anything the server has recorded, is one the server could never have shown that shooter. Rather than refuse the shot, the query clamps the view into range and still judges it: an old view is judged at the oldest tick kept, and a view from the future at the newest tick recorded, both with a fraction of zero. A player on a very slow link still gets an answer, but cannot ask to be judged against an arbitrarily old world.

Each clamp raises `LagCompensationViewClampedViolation` against the shooter's `Connection`, at most once per query. It carries:

- `ReportedTick`, the tick the shooter reported;
- `ClampedTick`, the tick the shot was judged at instead.

It defaults to `ViolationAction.Ignore`. A shot fired by the server itself, with a `null` shooter, raises nothing.

Watching for clamps

Subscribe to `ViolationManager.LagCompensationViewClampedViolationDetected` to be told about each clamp without changing the outcome:

```
coreManager.ViolationManager.LagCompensationViewClampedViolationDetected += OnLagCompensat
ionViewClamped;

private void OnLagCompensationViewClamped(in ViolationContext<LagCompensationViewClampedVi
olation> violationContext)
{
    LagCompensationViewClampedViolation violation = violationContext.Violation;

    // Count clamps per violationContext.Connection; violation.ReportedTick and violation.
ClampedTick say how far out the view was.
}
```

The event is raised on the network loop, by the query that received the view. A single clamp is ordinary on a link that spiked. A steady stream from one connection is either a very slow link or a client asking to be

judged against the oldest world it can get. To act on it, kicking for example, register a deciding handler with `RegisterViolationHandler<LagCompensationViewClampedViolation>`; see Violations.

Targets that stopped stay in the window

A target that stops, dies or despawns stays hittable until its last recorded tick leaves the window, so a shot fired just before the target went away is still judged against it. Its hit names no `NetworkSystem`; read `SystemId` instead.

The spawn compensation cap lives here too

`LagCompensationManager` also holds the cap on spawn compensation, the other half of the same problem: `MaximumSpawnCompensationMilliseconds`, defaulting to `DefaultMaximumSpawnCompensationMilliseconds` (150) and held by its setter to `MaximumSpawnCompensationCeilingMilliseconds` (500). Lag compensation judges a shot against the past; spawn compensation brings a freshly spawned object up to the present. The setting exists in every edition so code that sets it compiles against any build.

Channels and What Is Guaranteed

Channel.Reliable and Channel.Unreliable

`Channel` is a two-value enum: `Reliable` (ordered and reliable) and `Unreliable`. Every message and every call is sent on one or the other, chosen at the send call:

```
connection.SendMessage(Channel.Reliable, myMessage);
mySystem.TrySendRpc(RpcTarget.Observers, Channel.Unreliable, myRpc);
```

The channel is not a Nucleus feature built on top of the transport - it is a request passed straight through to it. The transport is what makes `Reliable` actually reliable: nothing in `RpcManager`, `MessageManager`, or `Connection` retries a send, buffers it for redelivery, or checks that it arrived. Pick `Reliable` and the transport guarantees order and delivery. Pick `Unreliable` and it guarantees neither.

Calls and messages are not state

A call or a message is deliberately not state. It carries no acknowledgment, no redundancy, and no delta baseline, and it is never resent to reach eventual consistency. It either arrives once, or it does not arrive at all - there is no third outcome where the engine notices a loss and tries again.

That means the channel is the only reliability lever either mechanism has. If a call or a message must arrive, it has to ride `Channel.Reliable`; there is no other way to make that true. Conversely, sending something on `Channel.Unreliable` is an explicit statement that losing it occasionally is fine.

This holds identically for messages and calls. A message has no address beyond the connection it travels to; a call additionally names a `NetworkSystem` so the receiver resolves the object before a handler runs. Neither carries any redelivery machinery, so for both, choosing the channel is most of the decision about whether the send is safe to lose.

Batching and framing

Messages and calls are batched per `Connection` and sent once per frame, one outbound writer per channel. Each one is length-framed: a message's body is written as `PackedBytes` with a known bit count, and a call's envelope (`RpcPacket`) carries the same for its body.

That framing is what keeps one bad payload from taking down the rest of the batch. Decoding runs per message and per call, not around the whole drain, so a body that fails to decode leaves nothing misaligned behind it - the reader already knows where that body ends from its declared length, and every payload packed after it in the same batch still dispatches normally.

A large unreliable send stays unreliable

A message or call larger than the transport's maximum transmission unit keeps the channel you asked for. A reliable one is handed to the transport, which splits it and delivers every piece. An unreliable one is split by the engine into pieces that each fit one datagram, on every transport, and the receiving peer rebuilds it once all of its pieces have arrived.

A split unreliable payload arrives whole or not at all. Losing any one piece loses the payload, so the more pieces it needs, the more often it is lost, and the receiver drops it once it has waited five seconds for a missing piece. An unreliable payload too large to split at all, more than 255 pieces or close to 300 KB at the default transmission unit, is dropped with an error, so send data that large reliably.

When neither one is the right tool

Messages and calls both fire once and are gone - nothing records that they happened for a peer that starts observing afterward. If a value needs to be correct for a late joiner, it belongs in a replicated member instead, which shows a new observer the value as of the last write rather than replaying past sends. See State, a Message, or a Call? and How replicated state works.

Messaging and Remote Calls

Pages in this category:

- Choosing Who Receives a Call
- Channels and What Is Guaranteed
- System RPCs
- Typed Messages
- Seeing Your Own Call: RpcSelfDelivery
- Who May Send a Call, and How Many
- Handler Registration and Scope
- State, a Message, or a Call?
- When a Message or Call Reaches Its Handler
- Judging and Refusing a Relayed Call
- Sending a Message to Many Peers
- Messages and Calls That Do Not Arrive

When a Message or Call Reaches Its Handler

Outbound messages and calls leave once per frame

Messages and calls sent by your code do not go on the wire the instant you call the send method. They queue, and the engine sends the queue once per frame, on every frame, at `NetworkLoopSteps.LateSerialize`'s place in the order. On a frame that ticks, the send happens after the `LateSerialize` callbacks, so a message or call sent from a `LateSerialize` callback or any earlier step still leaves this frame. One sent from a `TickAdvance` or `LateVariableUpdate` callback leaves on the next frame. Messages and calls go out in the same send, so a message and a call sent in the same frame leave together.

Inbound calls: LateStateUpdate, deliberately after messages

Inbound calls are dispatched on `NetworkLoopSteps.LateStateUpdate`, after the tick's state has applied. That is later than inbound messages, and on purpose: state, including a spawn that arrived this same tick, has already landed by `LateStateUpdate`. A call addressed to an object that spawned in the same frame resolves against a system that already exists, instead of being stepped over because the object wasn't there yet.

Send order holds across types

A peer's messages reach their handlers in the order they were sent, regardless of how many different message types are mixed in. This isn't per-type ordering — `MessageManager` backs local delivery with one ordered list (`_pendingLocalMessages`) rather than one queue per type, so the drain walks sends in the order they happened, not grouped by type.

Host loopback: Immediate, and only Immediate

A host is one `CoreManager` holding both the server and client roles. When one half addresses a message or call to the other, `TransportManager.HostLoopbackDelivery` decides when that delivery runs. Only `HostLoopbackDelivery.Immediate` is settable from game code: the receiving half's handlers run at the send site, before the send call returns, on whatever loop step the sender happened to be on.

`HostLoopbackDelivery.Deferred` exists in the enum but is refused with a logged reason when a game tries to set it. It's kept only so the engine's own tests can measure what the alternative would cost — handlers running one step later, at `EarlyVariableUpdate` on the following frame, the same step a real remote peer's copy would land on. Deferred delivery is message-only; a call is always dispatched at the send site because its delivery has to run the server's own admission (resolve the system, check the sender's permission, decide relay) inline.

Re-entrancy

Because `Immediate` runs a handler at the send site, a handler that sends something back re-enters the same pass that sent the original message or call. The engine supports this. The one consequence worth knowing: if a handler reached this way starts a spawn, and the sending step is `LateSerialize` or `TickAdvance`, after the tick's state was sent, that spawn is deferred until the tick advances. That is the same deferral any other late spawn gets, not a special case for loopback.

A handler throw costs only its own payload

Both `MessageManager` and `RpcManager` wrap each individual message or call dispatch in its own `try`/

`catch`, not the whole batch. A handler that throws is logged and the payload is dropped; every other message or call in the same drain — from the same sender, in the same frame — still reaches its own handler. A batch-wide guard would have cost a client everything the server sent that frame over one bad payload; the per-payload boundary is why it doesn't.

Messages and Calls That Do Not Arrive

The message hash could not be found for Type

`Connection.SendMessage<T0>` looks up the type's wire hash before anything else, and if no usable hash comes back it logs `The message hash could not be found for Type [...]` and returns. Nothing is queued.

The hash is worked out from the type's full name, so a sender never has to register a type before sending it. The lookup fails in only two cases:

- **Another message type this peer has already used hashes to the same value.** This is the usual cause. The first time it happens a separate error names both types (see the next section), and from then on every send of either type logs the error above.
- **The type has no full name to hash.** That logs `The FullName for [...] could not be found.` first. An ordinary concrete message type always has one.

This check runs before the host-loopback branch is even considered, so a host sending to its own other half is refused the same way as every other peer: loudly, and before anything is written.

The fix for a collision is renaming one of the two types, or moving it to another namespace, because the hash follows the full name. Registering a handler does not help: the same lookup refuses the handler too.

Two types resolving to the same wire hash

`MessageManager` and `RpcManager` each remember which type first claimed each wire hash. If a second, different type produces the same hash, the registry does not pick a winner: it refuses both types, for sending, for registering a handler and for delivering a received body, for as long as they collide.

The alternative would be delivering the second type's body to the first type's handler, reading bytes as the wrong shape. The registry treats a collision as unresolvable and names both types when it happens, rather than let ambiguous data through.

A call that reaches nobody

The receiver looks up the call's target system before any handler runs. If the receiver does not hold that system, the call is dropped with no log and no violation: an interest cull, a despawn, or a spawn whose priming full has not landed yet all look identical from here, and none of them are faults.

A routed call (`RpcRoute.TargetRelay`) has one more check once the system is found, still before any handler runs: the connection it names must be connected and must actually observe the addressed system. A target that has left or no longer observes it is refused the same way, silently.

So when a call goes nowhere, check interest before framing: does the receiver still hold the system, and does the intended recipient still observe it. A wire-level bug would produce a decode error or a violation. Silence is interest.

The sender never asked to see its own call

`RpcSelfDelivery` defaults to `None`, meaning the sender does not see its own call at all. A sender that expects immediate local feedback and gets nothing has not lost a call; it never asked to see it.

- `RpcSelfDelivery.Immediate` runs the sender's own handlers at the send site, before the call has gone anywhere. It runs even if the server later refuses the call.
- `RpcSelfDelivery.OnDelivery` runs the sender's handlers only once the call has actually been

delivered (and not at all if refused). On the server's own send this behaves like `Immediate`, since the server is the delivery point.

If a sender's own handler never fires, check what `RpcSelfDelivery` the send asked for.

A client's call refused

Three separate gates can refuse a client's call, in order:

`RpcAccess`. The server checks the sender against the system's `RpcAccess`. The sender must observe the system, and then `RpcSendAccess.AnyClient` admits any client while otherwise only the system's controller is admitted. The same check applies to a remote client's call and to a host's own client half calling its own server half. `CanLocalClientSendRpc` is a separate, earlier gate on the sending side: it lets a client queue a call upstream at all even when it is not the controller, and does not enter the server's admission check. If the system's access does not admit the sender, the receive is refused, and the server raises `RpcSendPermissionViolation` with the `SystemId` of the addressed system.

A server handler returning `RpcRelayAction.Cancel`. The server runs every registered handler and folds their answers: any single `Cancel` settles it, and no later `Relay` can overturn it. A handler that throws is treated as `Cancel` too: the throw counts as a refusal to relay, and the handler collection logs it, but the remaining handlers still receive the call.

`MaximumInboundRpcsPerConnectionPerDrain`. The server bounds how many calls one connection may deliver in a single drain (default 64). Once a connection exceeds it, `RpcFloodViolation` is raised (once per drain) with the received and allowed counts, and the rest of that drain's calls from that connection are dropped.

`RpcSendPermissionViolation` and `RpcFloodViolation` tell these two failure modes apart: a permission violation means the system's access rule rejected the sender; a flood violation means the sender delivered more calls in one drain than the bound admits.

The real host symptom: not "fires twice"

The engine already prevents every double-fire shape a naive host implementation could produce. The actual symptom to watch for with a host is different: with `RpcSelfDelivery.Immediate` on an addressed send, the host's client half runs a copy of calls addressed to *other* players, not only the ones addressed to itself.

This falls out of what a host is: one peer running both the server and a client. A call routed to another connection is relayed by the server half; if that call was sent with `Immediate` self-delivery, the sender's own handlers already ran at the send site, on the same process the server is running on. The host's client half is not a separate machine, so it sees the immediate copy of every call its server half sends, not just the ones meant for it. Check the target of the call, not whether it "happened twice."

A message type the receiver never registered

A receiver with no handler registered for a type is not an error. The message packet carries its own byte count, so a peer with no parser registered for that hash skips the packet's bytes and moves to the next one. No error, no log.

This is why a missing handler looks exactly like silence: the send succeeds, the packet arrives, and the receiver quietly steps over it. If a message type seems to vanish in transit, check that the receiving peer registered a handler for it. The sender needs no registration of its own.

Handler Registration and Scope

Using Unity? See Registering Call Handlers Across a Component's Lifetime.

Scope decides who a call reaches

`RpcManager` keys registered handlers by call type. Without a scope, every handler for a type would run for every object of that type in the world, forcing each handler to open by asking whether the call was actually addressed to it. `IRpcScope` moves that check into the engine:

```
public interface IRpcScope
{
    bool IsRpcScopedTo(NetworkSystem networkSystem);
}
```

`NetworkSystem` implements it by scoping to itself:

```
public bool IsRpcScopedTo(NetworkSystem networkSystem) => ReferenceEquals(this, networkSys
tem);
```

A handler registered through `NetworkSystem.RegisterRpcHandler` is invoked only for calls addressed to that instance, by reference identity. This matters beyond filtering: a handler that is never invoked for a call cannot return `RpcRelayAction.Cancel` for it. Scoping is what stops one object's handler from being able to refuse the relay of a call addressed to another.

`IRpcScope` is a predicate rather than a property naming "the system this is for," deliberately. A scope is not required to own exactly one system - something that hosts several systems can answer the question for all of them, which a property could not do.

Registering directly with RpcManager

Most code registers through the `NetworkSystem` methods below, which handle scope and lifecycle automatically. `RpcManager` itself exposes the two underlying overloads:

```
public void RegisterRpcHandler<T0>(RpcReceivedHandler<T0> rpcReceivedHandler) where T0 : I
Rpc, new();

public void RegisterRpcHandler<T0>(RpcReceivedHandler<T0> rpcReceivedHandler, IRpcScope rp
cScope) where T0 : IRpc, new();
```

The first registers unscoped: the handler receives every call of that type in the world. The second scopes the handler to whatever `IRpcScope` is passed, which can own any number of systems.

Registering through a NetworkSystem

```
public void RegisterRpcHandler<T0>(RpcManager.RpcReceivedHandler<T0> rpcReceivedHandler) w
here T0 : IRpc, new();

public void UnregisterRpcHandler<T0>(RpcManager.RpcReceivedHandler<T0> rpcReceivedHandler)
 where T0 : IRpc, new();
```

Calling `RegisterRpcHandler` before the system has spawned is legal. The handler is tracked on the system; if the system is already started it registers with the manager immediately, otherwise the spawn registers every tracked handler as it goes live. From there the handler follows the system's lifecycle:

- Deregistered from the manager when the system despawns.
- Re-registered when the system respawns.

- Dropped from tracking only when the instance returns to the pool.

A handler is live with the manager exactly while the object is in the world, and a caller never has to time registration to the spawn.

What dispatch costs

A handler registered through a `NetworkSystem` is scoped to that system, and the engine files it by the system itself. A received call addressed to a system finds that system's handlers with one lookup, however many other objects handle the same call type, so a world of a thousand objects each handling the same call pays the same per call as a world of one. Only handlers registered with no scope, or against an `IRpcScope` that is not a `NetworkSystem`, are walked and asked one by one, because an arbitrary scope can only be asked. Prefer registering through the system unless the handler genuinely speaks for many systems.

Re-registration and unregistration are idempotent and identity-scoped

Re-registering the same handler is a no-op: `NetworkSystem` checks whether the handler already has a tracked record before adding another, matching `RpcManager`'s own de-duplication.

Unregistering removes by handler identity only, never by scope. A handler belongs to at most one scope, so removing it by identity cannot remove another scope's handler.

Registering or unregistering during a dispatch

Each call type's handler collection tracks a dispatch depth while it enumerates handlers for a received call. A registration or unregistration requested while that enumeration is in progress cannot touch the live collection, so it defers. The deferred change is applied only once the outermost dispatch unwinds - not a re-entrant one, since a handler may itself invoke a call of the type it is currently handling. This is what makes it safe for a handler to unregister itself from within its own invocation, which matters because a system auto-clears its handlers when it returns to the pool.

Judging and Refusing a Relayed Call

A handler decides whether a call goes any further

An RPC handler on the server does not just react to a call - it can decide whether that call goes any further. Every handler returns an `RpcRelayAction`, and the server applies that verdict before writing a single byte to the peers the call would otherwise reach.

```
public enum RpcRelayAction : byte
{
    Relay = 0,
    Cancel = 1,
}
```

`Relay` is zero. A handler that returns nothing meaningful, or that has no opinion about relaying, defaults to `Relay` - a handler with no opinion cannot accidentally suppress a relay. `Cancel` keeps the call on the server instead of passing it on to the system's other observers.

The verdict never travels. It is a decision the server makes locally, before anything is written, so a receiving peer has nothing to decode for it and no bit width is reserved for it on the wire.

Veto folding

A call type can have more than one handler. When it does, every registered handler runs, regardless of what the ones before it returned, and the verdicts fold into one outcome: any single `Cancel` settles it for the whole call.

The fold only ever downgrades toward `Cancel`. A handler that answers `Relay` after an earlier handler already answered `Cancel` cannot undo that refusal - a permissive handler running later cannot overturn an earlier one's veto. The rule needs no priority or ordering between handlers because it only moves one direction.

A throwing handler counts as Cancel

Each handler is invoked in its own try/catch, so one throwing handler does not stop the handlers after it from receiving the call. But the throw itself is treated as a refusal: a handler that threw never answered `Relay`, so the call fails closed rather than travelling on unjudged. A payload crafted to crash a moderating handler does not buy it a pass.

RpcContext tells a handler what this peer is to the call

Alongside the verdict, a handler receives an `RpcContext` describing what this peer is to the call it's judging:

```
public readonly struct RpcContext
{
    public readonly Channel Channel;
    public readonly NetworkSystem NetworkSystem;
    public readonly Connection? SenderConnection;
    public readonly Connection? TargetConnection;
    public readonly RpcDisposition Disposition;

    public bool IsSenderServer => SenderConnection?.IsServer ?? false;
}
```

`Disposition` is one of three mutually exclusive answers:

```
public enum RpcDisposition : byte
{
    Recipient = 0,
    Origin = 1,
    Routing = 2,
}
```

- **`Recipient`** - the call was addressed to this peer, whose part is to act on it. A server admitting a client's fan-out reads as this too, since it is one of the observers the call is for, even as its verdict also decides whether the call travels on.
- **`Origin`** - this peer sent the call and is seeing its own copy, either as the server's local invoke or as an echo a client asked for.
- **`Routing`** - the call is passing through this peer to someone else, and was never addressed here. This only happens on the server, and only while it carries a client's targeted call on to the client that call named.

A handler that sees `Routing` still runs, and still gets a vote. That is deliberate: a call the server has not read is a call it cannot judge. A handler in this position judges the call and returns a verdict without applying whatever effect it would normally apply for a call addressed to it. `RpcContext.TargetConnection` names which connection the routed call is bound for - it is set only while `Disposition` is `Routing`, and is null otherwise.

`Cancel` does not unwind what already happened

`Cancel` governs the onward send alone. It does not unwind the handling that produced it - every handler, including the one that returned `Cancel`, has already run against the call before the verdict is applied. This is what lets a moderating handler log or count the very call it just refused to forward: it read the call, acted on reading it, and only then stopped it from reaching anyone else.

```
private RpcRelayAction OnRpcReceived(in RpcContext rpcContext, MyRpc value)
{
    if (ShouldBlock(value))
    {
        // Still seen and logged, just not relayed.
        Logger.LogWarning(typeof(MySystem), $"Blocked call from {rpcContext.SenderConnecti
on?.AsString()}: {value}");
        return RpcRelayAction.Cancel;
    }

    return RpcRelayAction.Relay;
}
```

Seeing Your Own Call: RpcSelfDelivery

RpcSelfDelivery decides whether a sender sees its own call

`RpcSelfDelivery` decides whether the peer that sends a remote call also sees that call itself, and when. It is a single enum, not a set of flags, because the three answers are mutually exclusive: a sender either runs its own call at the send site, waits for the copy that comes back, or does neither. A pair of flags could ask for two of those at once, which would run the handlers twice.

`RpcSelfDelivery` says nothing about where the call is going; `RpcTarget` already does. It is only about the sender's own view of a call it is sending.

```
public enum RpcSelfDelivery : byte
{
    None = 0,
    Immediate = 1,
    OnDelivery = 2,
}
```

It is the last parameter on `NetworkSystem.TrySendRpc<T0>`, and defaults to `None`:

```
public bool TrySendRpc<T0>(RpcTarget rpcTarget, Channel channel, T0 rpc, RpcSelfDelivery r
pcSelfDelivery = RpcSelfDelivery.None) where T0 : IRpc, new()
```

The three values

None

The sender does not see its own call. This is the default, and it's what a call whose effect the sender already applied by other means wants.

Immediate

The sender's own handlers run at the send site, before the call has gone anywhere. This is what makes a client-fired effect feel instant. The price is that it runs whether or not the call survives: a client's call can still be refused by the server afterward, and `Immediate` has already run by then. Use it where responsiveness matters more than being right.

OnDelivery

The sender sees the call once it has actually been delivered, and not at all if it was refused. For effects that must agree on when something happens, or whether it happens at all, a client asking for `OnDelivery` sees its own call only if the server passed it on. A refusal then reads as nothing happening, rather than as an effect that has to be undone.

The server is itself the delivery point. Asking for `OnDelivery` on a server's own send is treated the same as `Immediate`, resolved at the point of send rather than at the call site. That collapse is what lets one `TrySendRpc` call read the same on both peers instead of branching on role, the same way `RpcTarget.To` does for the destination.

`OnDelivery` is meaningless with `RpcTarget.Server`, and the engine refuses the send outright rather than silently downgrading it: a call addressed to the server is not passed on anywhere, so there is nothing for it to come back from.

```
if (isOnDelivery && isAddressedToAuthority)
{
    Logger<NetworkSystem>.LogError($"[{typeof(T0).Name}] was not sent to NetworkSystem [{A
sStringInternal()}]: {nameof(RpcSelfDelivery)}.{nameof(RpcSelfDelivery.OnDelivery)} waits
for a call to be passed on, and one addressed to the authority stops there.");

    return false;
}
```

The host traps

A host runs both a server and a client in the same process, sharing one handler registry for a system. Two `RpcSelfDelivery` combinations need care as a result.

Immediate with RpcTarget.Server is refused

On a host, a call addressed to `RpcTarget.Server` is already invoked locally as part of the send: the host's client half asks its own server half, and that ask runs the handlers right there. Asking for `Immediate` on top of that would run the handlers a second time, so the engine refuses the send and logs why:

```
if (isServerStarted && isAddressedToAuthority && rpcSelfDelivery is RpcSelfDelivery.Immedi
ate)
{
    Logger<NetworkSystem>.LogError($"[{typeof(T0).Name}] was not sent to NetworkSystem [{A
sStringInternal()}]: a host's call to its own authority is invoked locally already, so {na
meof(RpcSelfDelivery)}.{nameof(RpcSelfDelivery.Immediate)} on top of it would run the hand
lers twice.");

    return false;
}
```

Immediate with RpcTarget.Observers cannot double-fire

Sending to `RpcTarget.Observers` with `Immediate` does not double-fire on a host either, but for a different reason: the fan-out walk skips the host's own client connection when the send has already been locally invoked, so it is served exactly once, either by the local invoke or by the fan-out, never both.

Immediate with RpcTarget.To: the real pitfall

The trap that actually costs you something is not about double-firing. On a host, the client half and the server half register handlers on the same system, so `Immediate` runs the send site's handlers against the host's client half regardless of who the call is addressed to. Sending with `RpcTarget.To` and `Immediate` makes the host's client half run a copy of every addressed send, including calls addressed to other players, not just calls addressed to the host itself.

In other words: the host sees answers meant for somebody else. `TryQueueToConnection` still honors the address and skips queuing a redundant packet to the host's own loopback connection, but it does not stop the local invoke that already ran:

```
if (isLocallyInvoked && connection is { IsHostLoopback: true, IsClient: true })
    return true;
```

A handler on a system a host client listens to needs to check who a call was actually for before treating `Immediate` delivery as its own.

Telling your own copy apart: RpcContext.Disposition

A handler doesn't have to guess whether it's looking at its own call. `RpcContext.Disposition` is an

`RpcDisposition` the engine sets locally, from what it knows the moment before the handler runs; none of it travels on the wire.

```
public enum RpcDisposition : byte
{
    Recipient = 0,
    Origin = 1,
    Routing = 2,
}
```

`Origin` means this peer sent the call and is seeing its own copy: a local invoke as the server, or the echo a client asked for through `RpcSelfDelivery.OnDelivery`. A handler reading `Origin` knows the effect is one this peer caused rather than one done to it, which is the distinction an effect that must not fire for its own author needs.

`Recipient` is the ordinary case, a call meant for the peer reading it. `Routing` is server-only, and marks a call passing through on its way to a client it names elsewhere, never meant for the server itself.

Who May Send a Call, and How Many

What send access controls

Every `NetworkSystem` has an `RpcSendAccess`, layered beside control exactly as `StateWriteAccess` is layered beside write control.

```
public enum RpcSendAccess : byte
{
    Controller = 0,
    AnyClient = 2,
}
```

`Controller` is the default: only the server, or the single controlling client, may send the system a call. `AnyClient` opens sending to the server and to any client that currently observes the system.

Ordinal 1 is deliberately vacant. The value is serialized into authored Unity scenes, so renumbering `AnyClient` would silently reinterpret every scene that already carries a value of 1 as something else.

Setting and reading access

The server sets access with `TrySetRpcAccess`:

```
public bool TrySetRpcAccess(RpcSendAccess rpcSendAccess)
```

It's server-only; a client call is rejected, since letting a client widen its own send access would make the permission client-settable.

The current setting is readable from `RpcAccess`. On the client side, the read that matters is `CanLocalClientSendRpc`: a one-bit capability hint the server most recently declared, telling the local client whether it personally may send. It exists so a well-behaved client never sends a call the server is going to refuse, and it's always `false` on the server.

Nothing travels as a grant

The server enforces every call from its own copy of `RpcAccess`. A client is told only which access is in force and whether it personally may send (`CanLocalClientSendRpc`); it is never handed anything resembling a permission token. A forged "I may send" claim from a client is inert, because the server re-checks every incoming call against `RpcAccess` and its own observer roster.

Access widens permission, it never replaces observation

`AnyClient` is scoped to observers, not to every client in the world. Losing interest in a system silently removes the right to call it along with the interest itself - `AnyClient` means you may call what you can see.

Kept separate from state write access

`RpcSendAccess` and `StateWriteAccess` are deliberately different knobs. "You may ask this object to do something" and "you may overwrite this object's state" are different grants - folding them into one would mean a game that wants observers to fire a call at an object has to open its state stream to them as well.

RpcSendPermissionViolation

A call sent to a system without permission raises `RpcSendPermissionViolation`:

```
public struct RpcSendPermissionViolation : IViolation
{
    public uint SystemId;
}
```

`SystemId` is the Id of the system the sender addressed without permission. It's raised at most once per drain, however many refused calls that drain carried.

The violation reaches a well-behaved client too, on a revocation race: permission is withdrawn while calls already sent are still in flight. That's why it's a violation the game decides on rather than a discard the framework settles alone. Subscribe through `ViolationManager.RpcSendPermissionViolationDetected`.

Capping calls per drain

`RpcManager` bounds how many inbound calls one connection may deliver in a single drain:

```
public uint MaximumInboundRpcsPerConnectionPerDrain = DefaultMaximumInboundRpcsPerConnecti
onPerDrain;

public const uint DefaultMaximumInboundRpcsPerConnectionPerDrain = 64;

public const uint UnsetMaximumInboundRpcsPerConnectionPerDrain = 0;
```

The default is 64. Setting the field to `UnsetMaximumInboundRpcsPerConnectionPerDrain` removes the bound entirely. Without a cap, a peer packing calls into a datagram costs the receiver a decode and a dispatch per call while costing the sender almost nothing.

Calls past the budget are dropped rather than dispatched, and the excess raises `RpcFloodViolation` once per drain:

```
public struct RpcFloodViolation : IViolation
{
    public uint ReceivedRpcCount;
    public uint AllowedRpcCount;
}
```

`ReceivedRpcCount` is how many calls the sender delivered in the drain; `AllowedRpcCount` is how many the receiver admits per drain. Subscribe through `ViolationManager.RpcFloodViolationDetected`.

Choosing Who Receives a Call

RpcTarget

`RpcTarget` names the recipients of a remote call. It is a struct with factory members, not an enum, because the targets do not all carry the same information — only a connection target needs a connection.

- `RpcTarget.Server` — the server itself, and nobody after it.
- `RpcTarget.Observers` — every connection that observes the system.
- `RpcTarget.ObserversExcept(Connection excludedConnection, bool isControllerExcluded = false)` — every observer except the named connection, and optionally except the system's controlling client.
- `RpcTarget.To(Connection connection)` — one named connection, from either peer.

The exclusions live on `ObserversExcept` rather than on the send call itself, because skipping a peer or the controller only means anything when fanning out. That keeps the whole question of who receives a call in a single argument.

Server is a destination, not a first hop

`RpcTarget.Server` sends to the server and stops there. Nothing forwards a call sent this way. If a client wants the other peers watching the object to see something, it addresses them directly with `Observers` or `To`, and the server still decides whether that arrives by returning `RpcRelayAction.Cancel` from a handler that has read it.

A target names recipients, it does not grant permission

Building an `RpcTarget` never checks who is allowed to use it. That check happens when the call is sent:

- `RpcTarget.Server` is a client's to use.
- `RpcTarget.ObserversExcept` is the server's alone.

A client that names `ObserversExcept` — or otherwise sends from the wrong peer — is refused at the send, and the reason is logged rather than thrown, since no legitimate engine path produces that call.

A client's Observers request is admitted, not guaranteed

`RpcTarget.Observers` works from either peer, but the two directions are not identical. The server holds the observers and serves them directly. A client holds none, so sending with `Observers` is a request for a fan-out: the server admits it exactly as it admits any other client call, and a handler may still refuse it.

To(Connection) names the recipient by reference

`RpcTarget.To(Connection connection)` reaches one connection from either peer. The server sends straight there. A client has no direct link to another client, so it names the recipient and the server passes the call on — running its own handlers first, so it can still refuse.

Either way the recipient is named by reference: every client holds a `Connection` for every peer the server has reported, so `To` never needs to look a peer up by id.

RpcRoute on the wire

`RpcRoute` is what actually travels with the call, so the receiver can tell a legitimate call from one the sender was never entitled to make:

Value	Meaning
`Server`	A client's call to the server itself, which stops there.
`Observers`	A call to every connection observing the system — sent by the server, or requested by a client. The one value legal in both directions, told apart by `Connection.IsServer` on the connection the call arrived on.
`Target`	The server's call to a single observing connection.
`TargetRelay`	A client's call to a single connection it named, which the server passes on once it admits it.

`RpcRoute.Target` is server-only. One arriving from a client is discarded and logged rather than raised, because no engine path produces it — reaching it would mean forgery.

When the server passes a `TargetRelay` call on, it rewrites the route to `Target`. The recipient therefore cannot tell a call the server routed for a client from one the server addressed to it directly.

Sending a Message to Many Peers

Sending to everyone

`ServerManager.SendMessage<T0>` sends a message to every authenticated client. A host's own client is included, delivered in process rather than over a socket. A client still awaiting an authentication outcome is left out.

```
CoreManager.ServerManager.SendMessage(Channel.Reliable, new SomeMessage());
```

There is no client-side equivalent that fans out to multiple peers. See The client direction below.

Sending to a chosen set

The overload taking an `IReadOnlyCollection<Connection>` sends to exactly the clients you supply, without the authentication filter the whole-session overload applies. A caller that assembled its own set has already decided who belongs in it.

```
CoreManager.ServerManager.SendMessage(Channel.Reliable, new SomeMessage(), myConnections);
```

Sending to everyone but one

`SendMessageExcept<T0>` sends to every authenticated client except the one named.

```
CoreManager.ServerManager.SendMessageExcept(Channel.Reliable, new SomeMessage(), excludedC
onnection);
```

The exclusion is matched on `Connection.Id`, not by reference, so a `Connection` from any lookup, or one held from an earlier one, still names the same player. `excludedConnection` may be null, and null excludes nobody.

Sending to a scene's occupants

`SendMessageToScene<T0>` sends to every client that has the given scene instance loaded.

```
CoreManager.ServerManager.SendMessageToScene(Channel.Reliable, new SomeMessage(), sceneHan
dle);
```

`SceneManager.CollectConnectionsInScene` decides who counts as an occupant: every active connection for which `connection.IsSceneLoaded(sceneHandle)` is true. A scene is the observer rule most games actually want, so it's worth one call instead of renting a list and doing the filtering yourself.

The refusal on a peer with no server

Each of these calls opens with `EnsureIsServerStarted`. On a peer that has not started a server, the call does nothing and logs an error naming the method that refused. This is a loud refusal rather than a silent no-op: a send from a peer that is not the server would otherwise walk an empty or stand-in-only connection set and go quietly nowhere, which reads as a lost message rather than as a call that was never valid.

The client direction

There is no client-side fan-out API, by design: a client only ever has one peer to send to, the server it's connected to.

```
if (CoreManager.TransportManager.TryGetServerConnection(out Connection serverConnection))
    serverConnection.SendMessage(Channel.Reliable, new SomeMessage());
```

`TransportManager.TryGetServerConnection` resolves the local server connection, then `Connection.SendMessage` sends on it directly.

State, a Message, or a Call?

The three shapes

A **replicated member** carries a current value. The server writes it, the wire keeps every observer's copy converging on it, and a late observer is brought up to date the moment it starts observing.

A **message** is addressed to a peer and forgotten. It has no address beyond the connection it travels to, no acknowledgment, and no record that it happened once the handler returns.

A **call** is addressed to a `NetworkSystem`. The receiver resolves that system before any handler runs, so a call for a system the receiver does not have is stepped over rather than delivered to the wrong place.

Each mechanism carries a different envelope

A message's envelope carries a hash of the message type (`ushort`), so the receiver knows which handler decodes the body. There is no address and no framing beyond that.

A call's envelope (`RpcPacket`) carries the addressed system's id (`SystemId`), a route (`RpcRoute`, telling the server whether it is a fan-out to observers or bound for one target), and the call's own type hash. The body itself is length-framed (`PackedBytes`) so an unresolvable call can be skipped without decoding it, and the packet after it still reads correctly.

A replicated member sends a delta against a baseline rather than a fresh value every time, so its ongoing cost is the size of what changed, not the size of the whole value.

The late-joiner rule

A replicated member shows a new observer the value as of the last write. That is what "current value" means: the member's job is to be caught up to, not to be replayed.

A message or a call that fired before a peer started observing is gone. Nothing records it for a peer that arrives later, and neither `RpcManager` nor `MessageManager` keeps a backlog to replay. If a peer needs to know something happened before it joined, that fact has to live in a replicated member, not in the message or call that first announced it.

This is usually what decides between the three: does a newly-arrived peer need to know this, or only peers who were already there to see it happen?

Wire-type hashes are per mechanism

Messages and calls each keep their own registry of type hashes, resolved once per type and cached. Two message types (or two call types) landing on the same hash are caught the first time the second one is used, whether to send it or to register a handler for it, and an error names both types. From then on both types are refused, rather than silently letting one type's body be decoded as the other's. A collision between a message type and a call type cannot happen this way, because the two registries are separate; it can only happen within one mechanism.

The same protection holds on receive: a message or call whose hash two types collide on is skipped rather than guessed at. A hash with no handler registered is not an error: a message is skipped because nothing is there to decode it, and a call is not decoded, though the server still relays it.

Picking one

- **A score.** Every peer, including one that joins mid-match, needs to see the current value. Replicated member.

- **A door's open state.** Same reasoning: a peer that walks up to a door five minutes after it opened needs to see it open, not miss the event that opened it. Replicated member.
- **A chat line.** Nobody who wasn't there needs to receive it later, and it isn't addressed to an object in the world, just to peers. Message.
- **A hit effect.** It plays once, on the object it happened to, and a peer who joins after the hit doesn't need to see it play. That address to a specific `NetworkSystem` is what makes it a call rather than a message.
- **A server answer to one player's request.** Addressed to that one player's system, resolved once, done. Call, sent with a routed `RpcRoute` rather than fanned out to every observer.

System RPCs

Using Unity? See Sending a Remote Call From a Component.

A system RPC is a remote call addressed to a `NetworkSystem`. The receiver resolves the object before any handler runs, so a call for a system this peer does not hold is stepped over whole instead of being decoded into nothing. Without that address, a game would have to put a system id in its own payload, re-resolve it on receipt, write its own fan-out, decide relay itself, and invent a permission check. A system RPC closes all of that.

A call is a struct that implements IRpc

A call is a plain `[NetworkType]` struct that implements `IRpc`:

```
[NetworkType]
public struct DamageRpc : IRpc
{
    public int Amount;
}
```

`IRpc` is a marker interface. The type the game declares is the type the wire carries; there is no generated stub between them. Every call requires an authenticated sender, with no per-type opt-out: a call is admitted against the addressed object's own permission, so there is no shape of call an unauthenticated peer can reach.

Sending a call

`NetworkSystem.TrySendRpc<T0>` sends a call to the peers a target names:

```
bool sent = TrySendRpc(RpcTarget.Observers, Channel.Reliable, new DamageRpc { Amount = 10
});
```

- `RpcTarget` is who the call is for: `RpcTarget.Server` (client to server only), `RpcTarget.Observers` (fan-out to every observer), `RpcTarget.ObserversExcept(connection, isControllerExcluded)` (server only), or `RpcTarget.To(connection)` (one named connection).
- `Channel` is the channel to send on, which a call keeps whatever its size; an unreliable call larger than the MTU is split by the engine and arrives whole or not at all.
- `RpcSelfDelivery` controls when this peer sees its own call back (`RpcSelfDelivery.None` by default).

The return value reports only what this peer can see: a client cannot know the server's observer set, so a call it routed returning `true` means it was queued, not that it was delivered.

Handling a call

Register a handler on the system so it runs only for that system's calls:

```
RegisterRpcHandler<DamageRpc>(OnDamageRpc);
// ...
UnregisterRpcHandler<DamageRpc>(OnDamageRpc);
```

A handler matches `RpcManager.RpcReceivedHandler<T0>`:

```
private RpcRelayAction OnDamageRpc(in RpcContext rpcContext, DamageRpc rpc)
{
    Health -= rpc.Amount;

    return RpcRelayAction.Relay;
}
```

Return `RpcRelayAction.Relay` to let the call pass on if it was going to; return `RpcRelayAction.Cancel` to keep it on this peer. Where nothing was going to be passed on, the answer is discarded, so `Relay` is always safe.

RpcContext

The handler's `in` `RpcContext` parameter carries the call's circumstances:

Member	Meaning
`Channel`	The channel the call travelled on.
`NetworkSystem`	The system the call is addressed to, already resolved.
`SenderConnection`	The peer the call originated from. Nullable: null only when neither role is running locally.
`TargetConnection`	The connection a routed call is bound for, or null unless this peer is a routing waypoint.
`Disposition`	What this peer is to the call: `RpcDisposition.Origin`, `RpcDisposition.Recipient`, or `RpcDisposition.Routing`.
`IsSenderServer`	Whether the server is what sent this call.

`Disposition` is the field worth branching on. When it is `RpcDisposition.Routing`, the call was addressed to a different peer and this one is only deciding whether it travels, so a handler should judge and return without applying the call's effect.

Typed Messages

Using Unity? See Typed Messages in Unity.

A message is a point-to-point payload sent to one peer. It is never routed onward and never echoed back to whoever sent it. Declare a type, register a handler for it, and send it with `Connection.SendMessage`.

Declaring a message type

A message type implements `IMessage`:

```
public interface IMessage
{
    public bool IsAuthenticationRequired => true;
}
```

`IsAuthenticationRequired` is the only member. It defaults to `true`, so an ordinary type needs no override: the sender must be authenticated or the message is refused. This is read once per type, from a throwaway instance, when the type's handler is first registered.

Only types that take part in an authentication handshake override it to `false`, because they must arrive before anyone is authenticated. The engine ships none of its own: the `AuthenticationResponse` it sends on approval answers `true` like everything else, because by the time it is sent authentication has already happened. Write your own authenticator's pre-auth message types (the client's token, say) to answer `false`, or the server refuses them from the unauthenticated client trying to authenticate with them. The check is on the sender: a message from the server reaches a client that is not yet authenticated whatever its type says, so a refusal notice sent with `DenyClient` needs no override.

Registering a handler

```
public void MessageManager.RegisterMessageHandler<T0>(MessageReceivedHandler<T0> messageRe
ceivedHandler) where T0 : IMessage, new();
public void MessageManager.UnregisterMessageHandler<T0>(MessageReceivedHandler<T0> message
ReceivedHandler) where T0 : IMessage, new();
```

The handler shape:

```
public delegate void MessageReceivedHandler<T0>(in MessageContext messageContext, T0 messa
ge) where T0 : IMessage, new();
```

It returns nothing. A call's handler answers whether the server may pass the call on; a message is never passed on, so there is no verdict for a handler to give.

```
private void OnChatMessage(in MessageContext messageContext, ChatMessage message)
{
    // handle it
}

coreManager.MessageManager.RegisterMessageHandler<ChatMessage>(OnChatMessage);
```

Unregister the same delegate instance with `UnregisterMessageHandler<T0>` when you stop listening.

Sending a message

```
public void Connection.SendMessage<T0>(Channel channel, T0 data) where T0 : IMessage, new(
);
```

This sends `data` to that one `Connection`, on the given `Channel`. It is never delivered back to the sender,

even when the sender and receiver are the same peer's own other half (a host talking to itself is delivered in process, not echoed).

Reading MessageContext

```
public readonly struct MessageContext
{
    public readonly Channel Channel;
    public readonly Connection SenderConnection;
    public bool IsSenderServer => SenderConnection.IsServer;
}
```

- `Channel` — the channel the message travelled on.
- `SenderConnection` — the peer that sent it. **Never null**, unlike a call's `RpcContext.SenderConnection`: every message reaches a handler either off a link, where the receive pass names the connection it arrived on, or through this peer's own loopback delivery, which resolves the connection the receiving half would have seen.
- `IsSenderServer` — whether the server sent this. On a host, this is the only honest way to tell which half a message is from: a host is both roles at once, so its own `TransportManager.IsServerStarted` says nothing about any particular message.

What happens when the type's hash is refused

A type needs no registration to be sent: its wire hash is worked out from its full name. `SendMessage<T0>` looks that hash up before anything else, ahead of the loopback branch. If the hash is refused, because another message type this peer has used hashes to the same value or the type has no full name, it logs an error and returns without sending, on every peer alike, host included. There is no quiet no-op on any one peer. See Messages and Calls That Do Not Arrive for the fix.

Separately, the receiving side gates on authentication: if `IsAuthenticationRequired` is `true` and the sender isn't authenticated, the message is refused and the sender is disconnected through `ServerManager.KickClient`, not merely dropped.

Channels and What Is Guaranteed

Channel.Reliable and Channel.Unreliable

`Channel` is a two-value enum: `Reliable` (ordered and reliable) and `Unreliable`. Every message and every call is sent on one or the other, chosen at the send call:

```
connection.SendMessage(Channel.Reliable, myMessage);
mySystem.TrySendRpc(RpcTarget.Observers, Channel.Unreliable, myRpc);
```

The channel is not a Nucleus feature built on top of the transport - it is a request passed straight through to it. The transport is what makes `Reliable` actually reliable: nothing in `RpcManager`, `MessageManager`, or `Connection` retries a send, buffers it for redelivery, or checks that it arrived. Pick `Reliable` and the transport guarantees order and delivery. Pick `Unreliable` and it guarantees neither.

Calls and messages are not state

A call or a message is deliberately not state. It carries no acknowledgment, no redundancy, and no delta baseline, and it is never resent to reach eventual consistency. It either arrives once, or it does not arrive at all - there is no third outcome where the engine notices a loss and tries again.

That means the channel is the only reliability lever either mechanism has. If a call or a message must arrive, it has to ride `Channel.Reliable`; there is no other way to make that true. Conversely, sending something on `Channel.Unreliable` is an explicit statement that losing it occasionally is fine.

This holds identically for messages and calls. A message has no address beyond the connection it travels to; a call additionally names a `NetworkSystem` so the receiver resolves the object before a handler runs. Neither carries any redelivery machinery, so for both, choosing the channel is most of the decision about whether the send is safe to lose.

Batching and framing

Messages and calls are batched per `Connection` and sent once per frame, one outbound writer per channel. Each one is length-framed: a message's body is written as `PackedBytes` with a known bit count, and a call's envelope (`RpcPacket`) carries the same for its body.

That framing is what keeps one bad payload from taking down the rest of the batch. Decoding runs per message and per call, not around the whole drain, so a body that fails to decode leaves nothing misaligned behind it - the reader already knows where that body ends from its declared length, and every payload packed after it in the same batch still dispatches normally.

A large unreliable send stays unreliable

A message or call larger than the transport's maximum transmission unit keeps the channel you asked for. A reliable one is handed to the transport, which splits it and delivers every piece. An unreliable one is split by the engine into pieces that each fit one datagram, on every transport, and the receiving peer rebuilds it once all of its pieces have arrived.

A split unreliable payload arrives whole or not at all. Losing any one piece loses the payload, so the more pieces it needs, the more often it is lost, and the receiver drops it once it has waited five seconds for a missing piece. An unreliable payload too large to split at all, more than 255 pieces or close to 300 KB at the default transmission unit, is dropped with an error, so send data that large reliably.

When neither one is the right tool

Messages and calls both fire once and are gone - nothing records that they happened for a peer that starts observing afterward. If a value needs to be correct for a late joiner, it belongs in a replicated member instead, which shows a new observer the value as of the last write rather than replaying past sends. See State, a Message, or a Call? and How replicated state works.

Adapting your engine

IPhysicsBody

`IPhysicsBody` is the full surface the core reads from and writes to a body. Every member is required except the two default methods, which you may override.

Member	Type	Contract
`Position`	`Vector3` (get/set)	World-space position.
`Rotation`	`Quaternion` (get/set)	World-space rotation.
`LinearVelocity`	`Vector3` (get/set)	Linear velocity.
`AngularVelocity`	`Vector3` (get/set)	Angular velocity, as an axis scaled by radians per second.
`IsSleeping`	`bool` (get/set)	Reads whether the body is sleeping; setting `true` puts it to sleep, `false` wakes it.
`IsKinematic`	`bool` (get)	True when the body is kinematic and does not respond to forces.
`Gravity`	`Vector3` (get)	The gravitational acceleration actually acting on this body.
`Drag`	`float` (get)	The body's linear drag factor.
`Mass`	`float` (get)	The body's mass.
`AddForce(Vector3 force)`	method	Applies a continuous force to the body's center of mass for the current physics step.

Gravity and Drag

`Gravity` is not a global constant, it's what's actually acting on this specific body right now. If the body has gravity disabled, return `Vector3.Zero`, not the world's gravity vector. The trajectory projector integrates against exactly what this getter returns, so an adapter that always returns the world gravity will make a gravity-disabled body fall under projection while the engine holds it still.

`Drag` has to be in the same units the trajectory projector integrates against — whatever your engine's own linear damping field means, return it unconverted.

IsKinematic

`IsKinematic` is the follower's short-circuit. A kinematic body doesn't respond to forces or velocity, so there's nothing for convergence to nudge — the follower checks this flag and skips its own correction work rather than fighting an engine that's going to ignore it anyway.

Default snapshot methods

`IPhysicsBody` ships default implementations of `CaptureSnapshot` and `ApplySnapshot` built from the properties above; a plain adapter never needs to override them. Override `ApplySnapshot` only when your engine needs extra bookkeeping on a snapshot write beyond setting `Position`, `Rotation`,

`LinearVelocity` and `AngularVelocity`.

ISweepProvider

Optional. Implement it only if your engine can sweep a shape along a segment against static geometry:

```
bool TrySweep(IPhysicsBody physicsBody, Vector3 fromPosition, Vector3 toPosition, out Swee
pHit sweepHit);
```

Return `true` when the sweep hit blocking geometry, and fill `sweepHit` with `Position` (the body-center position at which the sweep was blocked), `Normal` (the surface normal at the contact) and `Fraction` (how far along the sweep the contact occurred, between zero and one). With no `ISweepProvider` registered, swept projection features stay unavailable and Nucleus falls back to the closed-form projector alone.

The numeric boundary

The core replicates entirely in `System.Numerics`. Your adapter is the only place engine types cross into that world, so it's also where you convert between your engine's vector/quaternion types and `Vector3`/`Quaternion`.

A minimal worked adapter

`Nucleus.Tests.Physics.FakePhysicsBody` is the adapter the test suite runs against — plain fields behind the interface, relying on the interface's default snapshot methods and no sweep provider at all. It's the floor: five settable properties, four read-only ones and one method satisfy `IPhysicsBody`.

Unity's adapters, as reference

`UnityPhysicsBody` (`Nucleus.Integrations.Unity/Physics/UnityPhysicsBody.cs`) wraps a `Rigidbody` and shows both contracts applied for real: `Gravity` returns `Vector3.Zero` when `Rigidbody.useGravity` is off, `Drag` maps to `Rigidbody.linearDamping`, and velocity writes are skipped on a kinematic body. Unity's sweep adapter is `UnitySweepProvider.Shared`, a static singleton you can reference directly.

How physics replication works

The loop

Every physics body Nucleus replicates follows the same loop, regardless of which engine drives it.

The controlling peer simulates the body normally and, once per tick, captures the stepped result: world position, world rotation, and a set of condition flags. That capture rides the wire as ordinary replicated members, alongside the tick it was captured at.

Every other peer holds a proxy for that body. Each physics step, the proxy rebuilds the latest server snapshot from those members, projects it forward by however many ticks old it is, and converges its own simulated body onto the projected result. The proxy is never teleported onto the raw received pose; it is steered toward a moving target that accounts for the snapshot's age.

Velocity is derived, not sent

Linear and angular velocity never cross the wire. Each side derives them independently from the change in position and rotation across the last two committed ticks, divided by the time between those ticks.

This halves what a moving body costs: a body in motion would otherwise carry position, rotation, *and* both velocities every tick it changes. A resting body carries nothing new at all, since an unchanged capture doesn't restamp its tick.

A proxy keeps simulating

A proxy body is never written directly onto the received pose. It keeps stepping through the same physics engine as the controller, with its velocity steered toward the projected target and its pose blended or corrected against divergence. Contacts, stacks, and collisions involving a proxy stay real on every peer, because every peer is still actually simulating it — not just moving a transform to match a number that arrived.

What's on the wire, and what isn't

Three things replicate: world position, world rotation, and a small set of condition flags (whether the body is sleeping, whether it's kinematic, and whether it has been captured at all). A capture tick rides alongside them so a receiver can anchor its projection to the snapshot's exact age.

No engine type ever crosses that boundary. The replicated members and the snapshot they rebuild are expressed in `System.Numerics` — `Vector3` and `Quaternion` — never an engine-specific vector, quaternion, or rigidbody handle. The core has no dependency on any particular physics engine; an integration is responsible for reading its engine's body into that shape and writing the projected result back onto it.

Position and rotation ride Divine or Interval transmission like any other member; the choice is per-member, not physics-specific. Divine can go quiet for stretches where Interval would still send every tick. Divine projection is Pro: a free build accepts `Divine` and replicates the member with ordinary deltas, as `Interval` does.

What isn't built

This model does not include:

- Replay-correct collision callbacks

This is planned, not hidden. A resimulated, full-world rollback approach to physics was tried and archived;

the model here (capture, project, converge) is what shipped in its place.

Judging a shot against where targets were is a separate feature that does not touch physics bodies at all: lag compensation (Pro) keeps its own hitbox history and its own geometry. Catching a freshly spawned projectile up through the latency it lost is spawn compensation (Pro).

Where core ends and integration begins

Everything above lives in the core engine, engine-agnostic and expressed in `System.Numerics`. Turning it into motion on an actual rigidbody — reading a body's state each tick, applying a projected target, wiring up sleep and kinematic flags — is an integration's job. Nucleus ships a Unity integration that does this for Unity's physics.

The core-side member and convergence types are the entry point for a custom or unsupported engine. The Unity-side components are the entry point for Unity.

Physics Synchronization

Pages in this category:

- NetworkPhysicsComponent
- PhysicsConvergence
- How physics replication works
- Replicate a physics body
- Adapting your engine
- Trajectory projection
- Locally predicted bodies
- Swept projection targets

Locally predicted bodies

Using Unity? See Driving a predicted body.

A body this peer is driving does not have to ride the server's velocity like a proxy does. Setting `IsLocallyPredicted` on that body's `PhysicsConvergence` switches it from following to reconciling: it keeps responding to local client state instantly, and only trims the small gap that opens up against the server's snapshot.

What IsLocallyPredicted changes

It is easy to assume a predicted body keeps the velocity the local simulation produced. It does not. `PhysicsConvergence.Step` assigns `physicsBody.LinearVelocity` from the projected snapshot unconditionally, the same as for a proxy. The source rejects the alternative deliberately: if the body kept its own velocity and only added a position bias on top, the two would drift apart with nothing to bleed the difference off, and a reversal in the server's motion would detonate that standing offset into a lurch.

So the counter-velocity assignment is shared between proxy and predicted paths. Three things are actually different for a predicted body:

- **Projection cap.** A proxy's snapshot is projected forward by at most `Settings.MaximumProjectionSeconds`. A predicted body uses `PredictionMaximumProjectionSeconds` instead, deliberately larger, because it is re-integrating its own client state across a full round trip rather than extrapolating a stranger's motion a short distance ahead.
- **No distance teleport.** A proxy that diverges past `Settings.TeleportDistance` snaps its whole pose onto the target at once. A predicted body is exempt. A driver's own client state is never expected to jump; any gap closes smoothly through the reconcile below instead.
- **Reconcile mechanism.** Past `Settings.PositionThreshold`, a proxy blends its pose toward the target (or applies a `BlendPerTick` velocity bias, depending on `ConvergenceBlendMode`). A predicted body instead applies a `PredictionCorrectionRate` velocity bias once it is past `Settings.PositionThreshold` or `Settings.RotationThreshold` — never before.

PredictionCorrectionRate

`PredictionCorrectionRate` (default `0.1f`) is the fraction of the position or rotation gap the correction takes off per tick, once past threshold. It scales the same way as a proxy's `BlendPerTick`, but applies only past `PositionThreshold` and `RotationThreshold` rather than continuously, so a well-predicted body that stays inside those thresholds is never nudged at all.

Setting it to `0` disables the correction entirely. The body becomes pure prediction: it no longer reconciles a genuine divergence, and since a predicted body is also exempt from the distance teleport, nothing ever pulls it back onto the server. This is rarely what you want — use it only to isolate prediction drift while debugging, not as a running configuration.

Higher values snap the body back to the server harder and sooner; lower values let the prediction ride looser and correct gently.

PredictionMaximumProjectionSeconds

`PredictionMaximumProjectionSeconds` (default `1f`) is the predicted counterpart to `Settings.MaximumProjectionSeconds`. A proxy's short cap keeps an extrapolation from running too far ahead of the last received fact. A predicted body instead re-integrates its own client state across the

whole round trip it has driven ahead of the server, so its window has to be allowed to span that round trip: capping it at the proxy's short bound would truncate the replay, land the target short of the actual prediction, and drag the body backward through the very reconcile it exists to avoid.

Choosing ticksToProject

`Step` takes `ticksToProject` as a parameter, not a constant. For a predicted body, base it on the measured round trip rather than a fixed tick count: `Connection.RoundTripTimeMilliseconds` gives that peer's current estimate. The instant-response feel a predicted body has comes from projecting the snapshot a full round trip forward on every step, not from retaining the body's own local velocity — the velocity assignment happens either way.

```
public uint RoundTripTimeMilliseconds { get; private set; }
```

Clearing IsLocallyPredicted

Flip `IsLocallyPredicted` back to `false` the moment control leaves the body: a body no longer driven locally has no client state to re-integrate, and left predicted it would keep skipping the teleport and the proxy reconcile it needs once it goes back to following the server.

`Reset()` clears `TargetError`, `HasTeleportedThisStep`, and `IsLocallyPredicted` together, for pool reuse or respawn.

Out of scope

Client state, the reconcile against confirmed server state, and the replay window that produces `ticksToProject`'s upper bound are a separate subject, covered on the control and prediction pages.

NetworkPhysicsComponent

This component replicates a body through ordinary member replication

`NetworkPhysicsComponent` replicates a physics body's kinematic state through ordinary member replication. The server captures its body into typed members each tick; proxies rebuild the latest server `PhysicsSnapshot` from those members as the projection source. Only position and rotation ride the wire — linear and angular velocity are derived from their per-tick change when a snapshot is built, so a moving body carries roughly half the members it otherwise would.

Replicated members

Member	Type	Wire settings
`WorldPosition`	`NetworkMember<Vector3>`	`CompressionLevel.Aggressive`, `TransmissionModeDefaults.Motion`
`WorldRotation`	`NetworkMember<Quaternion>`	`TransmissionModeDefaults.Motion`
`BodyFlags`	`NetworkMember<int>`	default
`CaptureTick`	`NetworkMember<int>`	default

`WorldPosition` and `WorldRotation` are built at `TransmissionModeDefaults.Motion`, so they divinely project wherever that mode is available.

`BodyFlags` carries a `PhysicsBodyFlags` value as an int member, since byte members do not currently ride the replication pipeline; the delta encoding keeps the wire cost equivalent.

`CaptureTick` is the tick the carried state was captured at, counted on the capturing peer's own tick counter: the server's for a body the server controls, the controlling client's for a body a client controls. It lets receivers anchor reconciles and projections to the snapshot's exact age instead of estimating it. It is carried as a signed int so the delta encoder can represent a decrease: a pooled component reused for a fresh body resets below the delta baseline, and an unsigned delta of that decrease would wrap to a 64-bit value and corrupt the stream. It holds `NetworkLoopManager.UnsetTick` until a ticked capture runs, and it is only re-stamped when the captured state actually changed, so a resting body stays wire-quiet instead of dirtying the member every tick.

PhysicsBodyFlags

Replicated body condition bits packed into `BodyFlags`.

Flag	Value	Meaning
`None`	`0`	No condition bits are set.
`Sleeping`	`1 << 0`	The body is sleeping.
`Kinematic`	`1 << 1`	The body is kinematic and does not respond to forces.
`Captured`	`1 << 4`	The flags were written by a real body capture.

`Captured` is never set on an unprimed member, so it doubles as the proof that real state has arrived — proxies read it as `HasReceivedState` rather than relying on the rotation member, which reads as identity when unset and cannot be distinguished from a genuinely captured identity rotation.

Derived reads

Property	Reads
`IsBodySleeping`	`BodyFlags` has `PhysicsBodyFlags.Sleeping` set.
`IsBodyKinematic`	`BodyFlags` has `PhysicsBodyFlags.Kinematic` set.
`HasReceivedState`	`BodyFlags` has `PhysicsBodyFlags.Captured` set.
`HasAdoptedState`	True once `TryAdoptInitialState` has placed a proxy body onto the first received state.

Methods you can call

```
public void CaptureBody(IPhysicsBody physicsBody, uint tick = Managers.NetworkLoop.Network
LoopManager.UnsetTick)
```

Called once per tick, before serialization, by the peer that controls the body (the server, unless a client controls it), to capture the body's current state into the replicated members. `CaptureTick` is only re-stamped when the captured position or rotation would actually serialize a change at that member's wire accuracy, not on every exact inequality, which would re-stamp on sub-quantum float jitter and drag the whole component onto the wire each tick even though nothing the receiver could observe had moved. This is what keeps a resting body wire-quiet.

```
public PhysicsSnapshot BuildSnapshot()
```

Rebuilds the latest server snapshot from the replicated members, called by proxies as the projection source. Linear and angular velocity are not replicated: they are derived from the change in `WorldPosition` and `WorldRotation` across their last two committed ticks, halving the moving wire cost in exchange for the small finite-difference error a one-tick-old average velocity carries into the projection.

```
public bool TryAdoptInitialState(IPhysicsBody physicsBody)
```

Applies the first received state directly onto the body, once. A wire-spawned proxy instantiates at the prefab pose, and there is no local trajectory worth preserving before the first fact from the server — converging from the prefab pose would visibly glide the body toward its real position instead of starting it there. Call this each proxy step before convergence; the step that adopts should skip its convergence. Returns `false` once `HasAdoptedState` is already true, or until `HasReceivedState` is true.

PhysicsSnapshot

The value every method above trades in — remote server state, local body state, and projected targets are all expressed as this one type.

Field	Type	Meaning
`WorldPosition`	`Vector3`	The world-space position of the body.
`WorldRotation`	`Quaternion`	The world-space rotation of the body.

`LinearVelocity`	`Vector3`	The linear velocity of the body.
`AngularVelocity`	`Vector3`	The angular velocity of the body, as an axis scaled by radians per second.
`IsSleeping`	`bool`	True when the body is sleeping; a sleeping snapshot projects to itself.

Velocity is derived, not replicated

Neither linear nor angular velocity is a replicated member. Both are computed from the per-tick change of `WorldPosition` and `WorldRotation`:

- Linear velocity is `(WorldPosition.Value - previousPosition) / seconds`, using the last two committed position samples and the tick gap between them.
- Angular velocity is the rotation vector carrying the previous rotation onto the current one, divided by the same elapsed seconds.

A sleeping body, or one that does not yet have two committed samples, derives zero velocity in both cases.

PhysicsConvergence

Using Unity? See ProjectedRigidbody.

`PhysicsConvergence` is the per-body follower behind Projected physics sync. It takes the latest server snapshot, projects it to now, rides its velocity, and blends or teleports a diverged pose back onto it. Call `Step` once per physics step for each proxy body, before the engine simulates.

Step

```
public void Step(IPhysicsBody physicsBody, in PhysicsSnapshot remotePhysicsSnapshot, float
 tickDelta, float ticksToProject)
```

Reads the body's current position, rotation and velocity, plus `remotePhysicsSnapshot` (the last snapshot received from the server), `tickDelta` (the duration of one tick), and `ticksToProject` (how many ticks that snapshot lags local time). Writes the body's `LinearVelocity`, `AngularVelocity`, and, on a blend or teleport, its `Position` and `Rotation`.

Call `Step` before your own world step for that tick, so the follower's velocity and any position/rotation write are in place before the engine integrates. If `physicsBody.IsKinematic`, `Step` returns immediately without touching the body.

PhysicsConvergenceSettings

`Settings` on `PhysicsConvergence` holds the tuning. `new PhysicsConvergenceSettings()` gives the recommended defaults, shown below.

Field	Default	Meaning
`PositionThreshold`	`0.1`	Positional divergence in meters past which the body blends back toward the target. Below it, the counter-velocity alone holds the offset steady.
`RotationThreshold`	`5`	Rotational divergence in degrees past which the body blends back toward the target.
`BlendPerTick`	`0.1`	Fraction of the remaining gap closed per tick while over threshold, from 0 (never) to 1 (onto the target in one tick).
`ResidualCloseRate`	`0.5`	Per-second rate at which a sub-threshold positional residual is bled off, as a velocity bias toward the target. Zero holds the offset steady instead.

`ResidualCloseMaximumSpeed`	`0.5`	Body speed (m/s) above which the residual close is skipped in favor of the plain deadband, so a freshly bumped or still-moving body isn't fought.
`TeleportDistance`	`4`	Positional divergence in meters past which the body snaps its whole pose onto the target instead of blending. Zero disables the teleport.
`MaximumProjectionSeconds`	`0.2`	Cap, in seconds, on how far a stale snapshot is extrapolated before the blend takes over.
`GravityProjection`	`GravityProjection.Auto`	Whether gravity participates in the projection.
`BlendMode`	`ConvergenceBlendMode.Position`	How an over-threshold divergence is closed; see below.

ConvergenceBlendMode

`Position` writes the pose directly toward the target each step (a per-step lerp/slerp). `Velocity` applies the same closing fraction as a velocity bias the engine's solver integrates instead.

The two converge identically in free space — same fraction of the gap closed per tick. They differ once the body touches something: `Position` bypasses the solver, so a blend into geometry penetrates and then depenetrates. `Velocity` is mediated by the solver, so the same blend against geometry resolves as a bounded contact press, and never yanks a body out of a stack or through what it's resting on.

The three bands

Each step, `Step` measures the divergence and picks one of three responses:

- **Over `TeleportDistance`** — the whole pose (position and rotation) snaps onto the target instantly. `HasTeleportedThisStep` is set. This only applies to an ordinary proxy; a locally predicted body is exempt (see below).
- **Over threshold, under teleport distance** — position and rotation each blend toward the target at `BlendPerTick`, per `BlendMode`.
- **Under threshold** — the position is left untouched, but if the body's own speed is below `ResidualCloseMaximumSpeed`, a velocity bias of `ResidualCloseRate` is added toward the target so a slow, never-sleeping body's residual still closes through the solver. A body moving faster than `ResidualCloseMaximumSpeed` — freshly bumped or still settling — skips the residual close and falls back to the plain deadband: the counter-velocity holds it steady until the next real divergence, so the gentle close never fights a new contact.

Reads and reset

- `TargetError` — the positional error from the most recent step (target position minus body position), read-only.
- `HasTeleportedThisStep` — true when the most recent step snapped the pose instead of blending it,

so a driver can collapse a smoothed visual's interpolation window and render the jump instantly.

- `Reset()` — clears `TargetError`, `HasTeleportedThisStep`, and `IsLocallyPredicted`, for pool reuse or respawn.
- The kinematic short-circuit: if `physicsBody.IsKinematic`, `Step` returns before reading or writing anything else on the body.

TargetProvider

By default `Step` projects the snapshot with a closed-form trajectory. Setting `TargetProvider` (an `IProjectionTargetProvider`) replaces that projection with whatever the provider computes instead — for example a target that sweeps against geometry rather than following a pure ballistic path. See swept projection targets for that provider.

Locally predicted bodies

`IsLocallyPredicted`, `PredictionMaximumProjectionSeconds`, and `PredictionCorrectionRate` only apply to a body this peer is driving with its own client state rather than following from the server. See locally predicted bodies.

Replicate a physics body

Using Unity? See Replicate a rigidbody.

`NetworkPhysicsComponent` replicates a simulated body's position and rotation through the normal member pipeline. The server captures its body into the component after it steps; every other peer rebuilds a snapshot from the component and steps a convergence follower toward it before it steps. Neither side needs an engine object — this page uses nothing but plain C# and `IPhysicsBody`.

Getting the component

Add `NetworkPhysicsComponent` to a system the same way you add any other component, by renting it:

```
if (!NetworkSystemPool.TryRent<NetworkSystem, NetworkPhysicsComponent>(coreManager, out Ne
tworkSystem? networkSystem, canStartSystem: true))
    return;

networkSystem.TryGetComponent(out NetworkPhysicsComponent physicsComponent);
```

A wire-spawned system that arrives on a receiving peer has no GameObject or other engine object to hang a reference off, so there is nothing to look up by name or tag. Declare a standing watch instead, and the pool hands you the component when a matching system arrives:

```
NetworkSystemPool.Watch<NetworkSystem, NetworkPhysicsComponent>(
    coreManager,
    networkSystemAcquiredHandler: system =>
    {
        system.TryGetComponent(out NetworkPhysicsComponent physicsComponent);
        // physicsComponent is fully initialized here.
    });
```

The watch matches on composition — any `NetworkSystem` carrying a `NetworkPhysicsComponent` — not on an identifier learned in advance.

The server half: capture

After your own world step, call `CaptureBody`:

```
physicsComponent.CaptureBody(physicsBody, tick: currentTick);
```

This reads `Position`, `Rotation`, and the sleeping/kinematic flags off `physicsBody` into the component's replicated members. The `tick` argument is stamped onto `CaptureTick` only when the capture actually changed something a receiver could observe — a resting body stays wire-quiet — and it is what lets a receiver anchor a snapshot to its exact age instead of estimating it. Omit it and no attribution is carried.

The follower half: converge

On every other peer, before your world step, rebuild the latest snapshot and hand it to a `PhysicsConvergence`:

```
PhysicsSnapshot snapshot = physicsComponent.BuildSnapshot();
float ticksToProject = GetTicksToProject(physicsComponent);

convergence.Step(physicsBody, in snapshot, tickDelta, ticksToProject);
```

`BuildSnapshot` rebuilds position and rotation straight from the replicated members; linear and angular velocity are derived from the change between the last two committed ticks, so they never ride the wire

themselves. `Step` projects that snapshot forward by `ticksToProject` ticks and steers `physicsBody` toward it — riding its projected velocity, teleporting on a large divergence, blending a smaller one back in. `Step` does nothing to a kinematic body.

`ticksToProject` is the snapshot's age in ticks. Peers do not share a tick counter, so it is not this peer's tick minus `CaptureTick`: measure it in the server's numbering, with `Connection.TryGetServerTickEstimate` on the connection to the server.

```
private const float FallbackTicksToProject = 2f;

private float GetTicksToProject(NetworkPhysicsComponent physicsComponent)
{
    uint captureTick = (uint)physicsComponent.CaptureTick.Value;

    if (captureTick != NetworkLoopManager.UnsetTick
        && coreManager.TransportManager.TryGetServerConnection(out Connection serverConnec
tion)
        && serverConnection.TryGetServerTickEstimate(coreManager.NetworkLoopManager.Tick,
out uint serverTick)
        && serverTick > captureTick)
        return serverTick - captureTick;

    // No attributed capture or no estimate yet: project a short fixed distance.
    return FallbackTicksToProject;
}
```

The estimate is of the server's tick, so this measures the age of a body the server captures, which is the setup this page describes.

Where each half runs in the loop

Both halves are ordinary `INetworkLoopStepCallback` subscribers, split across the two fixed-update steps that bracket your world step:

```
public class PhysicsDriver : INetworkLoopStepCallback
{
    public NetworkLoopSteps GetNetworkLoopSteps() => NetworkLoopSteps.EarlyFixedUpdate | N
etworkLoopSteps.LateFixedUpdate;

    public void OnNetworkLoopStep(NetworkLoopSteps networkLoopStep, StepDelta stepDelta)
    {
        if (networkLoopStep is NetworkLoopSteps.EarlyFixedUpdate)
        {
            // Follower: converge before the world steps, so the step integrates the corre
cted velocity.
            if (!isAuthority)
            {
                PhysicsSnapshot snapshot = physicsComponent.BuildSnapshot();
                float ticksToProject = GetTicksToProject(physicsComponent);
                convergence.Step(physicsBody, in snapshot, tickDelta, ticksToProject);
            }

            WorldStep(); // your simulation
        }
        else if (networkLoopStep is NetworkLoopSteps.LateFixedUpdate)
        {
            // Authority: capture after the world has stepped, so the capture describes th
is tick's result.
            if (isAuthority)
                physicsComponent.CaptureBody(physicsBody, tick: currentTick);
        }
    }
}
```

`EarlyFixedUpdate` is before the world step; `LateFixedUpdate` is after it. A follower converges on the early side so the step it is about to take integrates the corrected velocity; the server captures on the late side so what goes out describes the tick that just ran.

Priming a freshly observed body

A proxy that has just started watching a system has no local trajectory worth preserving — it sits at whatever pose it was constructed with, and converging from there would visibly glide it toward the real position. Call `TryAdoptInitialState` each follower step before `Step`, and skip the convergence on the step it succeeds:

```
if (!physicsComponent.TryAdoptInitialState(physicsBody))
{
    PhysicsSnapshot snapshot = physicsComponent.BuildSnapshot();
    float ticksToProject = GetTicksToProject(physicsComponent);
    convergence.Step(physicsBody, in snapshot, tickDelta, ticksToProject);
}
```

`TryAdoptInitialState` only does anything once: it returns `false`, and does nothing, until `HasReceivedState` first turns true (the component has captured at least once) and it has not already adopted (`HasAdoptedState`). After that first successful call every later step falls through to ordinary convergence.

A runnable sketch

`Nucleus.Tests/Physics/FakePhysicsBody.cs` is a minimal, engine-free `IPhysicsBody`: plain fields for position, rotation, and velocity, with an `Integrate` method standing in for a real simulation step. It is

exactly what the snippets above are written against, and `Nucleus.Tests/Physics/NetworkPhysicsComponentTests.cs` exercises the whole round trip — capture, `BuildSnapshot`, tick attribution, and `TryAdoptInitialState` — against real rented systems, with no Unity dependency anywhere.

For adapting a real physics engine's body to `IPhysicsBody`, see the Unity integration's rigidbody page linked above.

Swept projection targets

The seam

`PhysicsConvergence.Step` computes where a proxy body should be this step by projecting the latest server snapshot forward. By default that projection is the closed-form `TrajectoryProjector`, which has no idea what geometry exists between the body's old position and its new one. Set `PhysicsConvergence.TargetProvider` to an `IProjectionTargetProvider` and the follower defers to it instead — `ComputeTarget` receives the body, the remote snapshot, the tick delta, the ticks to project, and the resolved `GravityProjection`, and hands back the `PhysicsSnapshot` to steer onto. The controller doesn't know or care how the target was produced.

```
public interface IProjectionTargetProvider
{
    void ComputeTarget(IPhysicsBody physicsBody, in PhysicsSnapshot remotePhysicsSnapshot,
 float tickDelta,
        float ticksToProject, GravityProjection gravityProjection, out PhysicsSnapshot tar
getPhysicsSnapshot);
}
```

`TargetProvider` is left `null` by default in the core API — an ordinary proxy just rides the closed-form target. The Unity integration does not leave it null (see below).

SweptTargetProvider

`SweptTargetProvider` wraps the closed-form target with a sweep against your collision world. It first projects the snapshot the ordinary way, then, if the resulting displacement clears `MinimumSweepDistance` (default `0.05f`), asks an `ISweepProvider` to sweep the body's shape from the old position to the projected one. If the sweep reports a blocking hit, the target clamps to the hit position instead of the original projected position — the target can never land past geometry the sweep found in the way.

```
public sealed class SweptTargetProvider : IProjectionTargetProvider
{
    public float MinimumSweepDistance = 0.05f;
    public bool ReflectEnabled;
    public float Restitution = 0.6f;

    public SweptTargetProvider(ISweepProvider sweepProvider) { ... }
}
```

Three knobs control it:

- **MinimumSweepDistance** (`0.05f`) — projections shorter than this skip the sweep entirely; a short hop can't tunnel through anything worth querying for, so the cost is only paid when it might matter.
- **ReflectEnabled** — when on, the velocity at a clamped target reflects about the contact normal, scaled by `Restitution`. Off by default; the reflection is only accurate for simple bouncy shapes.
- **Restitution** (`0.6f`) — the bounciness applied by the reflect tier when it's enabled.

Install it on the follower:

```
convergence.TargetProvider = new SweptTargetProvider(mySweepProvider);
```

`ISweepProvider` is the one capability it needs — an engine integration supplies `TrySweep`, sweeping the body's collision shape between two points against static geometry. In Unity that's `UnitySweepProvider`,

built on `Rigidbody.SweepTest`.

When it earns its cost

A sweep is an extra collision query every physics step the provider runs, so it isn't the default. It's worth turning on when a projected proxy can plausibly be pulled through something it should have bounced off: fast bodies whose per-tick displacement is large relative to nearby geometry, thin walls or floors that a coarse projection can straddle in one step, or any proxy that visibly tunnels through a surface during a fast pass. A slow body drifting across open space has nothing to gain from it.

Writing your own provider

`SweptTargetProvider` assumes the swept-and-clamped model: stop at the first contact, optionally bounce. That's not always right — you might want a stepped re-integration, a provider that ignores certain layers, or something with no relation to sweeping at all. Implement `IProjectionTargetProvider` directly for that. The one rule that must hold regardless of what the provider does internally: it has to produce a `PhysicsSnapshot` the follower can steer onto — a valid position, rotation, and the linear/angular velocity `PhysicsConvergence.Step` will assign straight onto the body. Anything else about how that snapshot is derived is the provider's own business.

Unity always wires it up

`ProjectedRigidbody.BindSystem` installs `SweptTargetProvider(UnitySweepProvider.Shared)` on every body it binds, unconditionally. A gravity projection on a proxy with no sweep target would otherwise carry it straight through the floor between ticks; the Unity integration doesn't leave that as an opt-in. Leaving `TargetProvider` null is the ordinary case only for the core API used outside the Unity integration — a Unity project always has a swept provider installed underneath it.

Trajectory projection

TrajectoryProjector

`TrajectoryProjector` is a static, engine-independent class that projects a physics body's position, velocity, and rotation forward by an arbitrary number of ticks. It is closed-form: the result equals what a stepped simulation would reach after that many ticks of Euler integration with gravity and linear drag, computed in one call rather than by looping a tick at a time. It has no upper bound on how far ahead it will project; nothing in the class itself caps `ticksAhead`.

ProjectPosition

```
public static Vector3 ProjectPosition(float tickDelta, float ticksAhead, Vector3 position,
 Vector3 velocity, Vector3 gravity, float drag)
```

Projects a position forward by `ticksAhead` ticks. `ticksAhead` can be fractional; a fractional value projects partially into the tick it lands in. `drag` at zero or below projects without drag. Throws `ArgumentOutOfRangeException` if `ticksAhead` is negative.

ProjectVelocity

```
public static Vector3 ProjectVelocity(float tickDelta, float ticksAhead, Vector3 velocity,
 Vector3 gravity, float drag)
```

Projects a linear velocity forward the same number of ticks, under the same gravity and drag. Same negative-`ticksAhead` guard as `ProjectPosition`.

ProjectRotation

```
public static Quaternion ProjectRotation(float tickDelta, float ticksAhead, Quaternion rot
ation, Vector3 angularVelocity)
```

Projects a rotation forward along a constant angular velocity, given as an axis scaled by radians per second. A zero angular velocity returns the input rotation unchanged.

ProjectSnapshot

```
public static PhysicsSnapshot ProjectSnapshot(in PhysicsSnapshot physicsSnapshot, float ti
ckDelta, float ticksAhead, Vector3 gravity, float drag, GravityProjection gravityProjectio
n)
```

The one-call form: projects a full `PhysicsSnapshot` (position, rotation, linear velocity, angular velocity) forward by `ticksAhead` ticks, resolving gravity participation through `gravityProjection` before calling `ProjectPosition`, `ProjectVelocity`, and `ProjectRotation` internally. This is what the convergence follower calls each tick.

A sleeping snapshot (`physicsSnapshot.IsSleeping`) is returned untouched — no projection math runs on it.

ExtractRotationVector

```
public static Vector3 ExtractRotationVector(Quaternion fromRotation, Quaternion toRotation
)
```

The inverse of `ProjectRotation`: given two orientations, returns the shortest-arc rotation between them as a world-space axis-angle vector in radians. Dividing the result by the seconds it spans gives an angular velocity that `ProjectRotation` consumes directly; multiplying it by a per-second gain gives the angular

velocity that closes the gap as a correction. A degenerate (near-identity) arc returns `Vector3.Zero`.

GravityProjection

```
public enum GravityProjection : byte
{
    Apply = 0,
    None = 1,
    Auto = 2,
}
```

Controls whether `TrajectoryProjector` folds gravity into a projection.

- **Apply** — gravity is always applied.
- **None** — gravity is never applied.
- **Auto** — gravity is applied only when the snapshot is already descending fast enough to be a real fall, not solver jitter or quantization noise on a resting or supported body. A slow or upward-moving snapshot projects without gravity even under `Auto`, so a resting body is not projected into the ground.

`Auto` is a black-box test against the snapshot's own descent speed; it does not read a "grounded" flag or any collision state.

Who calls this, and who clamps it

`TrajectoryProjector` takes `ticksAhead` as given and projects that far, unbounded. The cap comes from the caller: `PhysicsConvergence.Step` computes a maximum projection time — `Settings.MaximumProjectionSeconds` normally, or `PredictionMaximumProjectionSeconds` for a locally predicted body — divides it by `tickDelta` to get a tick count, and clamps the requested `ticksToProject` into `[0, maximumTicks]` before ever calling `ProjectSnapshot`. The projector itself never sees or enforces that limit.

This split is what lets the projector serve other callers directly, without inheriting convergence's clamp: a spawn-time lead (projecting a freshly received snapshot forward by the ticks already elapsed since it was sent), a diagnostic that reports where a body will be N ticks from now, or a visual predictor that projects further ahead than convergence would ever allow for smoothing purposes. Any of these call `ProjectSnapshot` (or the individual `ProjectPosition`/`ProjectVelocity`/`ProjectRotation` methods) with whatever `ticksAhead` and `GravityProjection` fit the use, independent of `PhysicsConvergence`.

Unity: Gravity Enabled and Maximum Projection Time

On `ProjectedRigidbody`, the **Gravity Enabled** inspector checkbox and **Maximum Projection Time** field are the two knobs over this math. Gravity Enabled maps to `GravityProjection.Auto` when checked and `GravityProjection.None` when unchecked. Maximum Projection Time is the seconds value passed as `Settings.MaximumProjectionSeconds`, which `PhysicsConvergence.Step` uses to clamp `ticksToProject` for a proxy body — a predicted body ignores it and uses its own round-trip-spanning maximum instead. Neither field exposes anything beyond what `TrajectoryProjector` and `GravityProjection` do here; the inspector is a thin front end over this same projection.

The content bundle protocol

Using Unity? See Content bundles in Unity.

A content bundle is a `ushort` identifier for a shard of content that is not in the base build: a set of prefabs a client might not have installed yet. Nucleus carries the identifier and the load/unload protocol between the peers; it ships no loader of its own. `Connection.BaseBuildBundleId` (0) means the content shipped in the build, and is always considered loaded.

Registering a loader

Implement `IBundleLoader`:

```
public interface IBundleLoader
{
    Task<bool> LoadBundleAsync(ushort bundleId);
    Task<bool> UnloadBundleAsync(ushort bundleId);
}
```

`LoadBundleAsync` must register the bundle's prefabs before the task completes, since a completion tells the server it may spawn into the content immediately. `UnloadBundleAsync` releases it. Both may complete asynchronously, on any thread; the outcome is queued and drained back onto the network loop, so nothing a loader does off-thread races the outbound writers.

Register the loader with `BundleManager.SetBundleLoader`:

```
coreManager.BundleManager.SetBundleLoader(myLoader);
```

The engine consults exactly one loader per peer. Core ships no implementation; mapping a bundle identifier to an asset bundle, an addressable catalog entry, or a download is entirely the game's concern.

Gating spawns on content

`NetworkSystem.RequiredBundleId` is the bundle a spawn needs. It defaults to the bundle the prefab itself was built into (`PrefabBundleId`), but can be overridden with `SetRequiredBundleId(ushort bundleId)`. The sentinel `UnsetRequiredBundleId` (`ushort.MaxValue`) clears an override and falls back to the prefab's own bundle.

`BundleManager.SpawnGate` is a `BundleInterestCondition` (`Capabilities => InterestEffect.Spawn`) that withholds a system's spawn from a Connection that does not yet hold `RequiredBundleId`. It is not registered for you; attach it at whatever scope fits:

```
// Globally, for a world where most content hotloads:
coreManager.InterestManager.AddCondition(coreManager.BundleManager.SpawnGate);

// Or only on the prefabs that actually ship in a bundle:
networkSystem.AddInterestCondition(coreManager.BundleManager.SpawnGate);
```

The condition is stateless and safe to register against any number of systems. Without it, a spawn for content a client lacks arrives anyway and fails at instantiation; with it, the object simply does not exist for that client until the bundle is confirmed, and then spawns normally.

Getting the bundle loaded

With `BundleManager.AutomaticRequestOnBlockedSpawnEnabled` (true by default), a spawn the gate blocks also asks that client to load the missing bundle, so content hotloads on demand and the object

appears the moment the client confirms. Clear the flag to drive every load from game code instead; the gate still withholds spawns but never asks on its own.

To ask explicitly:

```
coreManager.BundleManager.RequestBundleLoad(connection, bundleId);
```

This is server-side, and dedupes per Connection and bundle — a repeat while a request is outstanding is dropped.

A peer can also load or release a bundle on itself, independent of any request from the other side:

```
bool loaded = await coreManager.BundleManager.LoadBundleAsync(bundleId);
bool unloaded = await coreManager.BundleManager.UnloadBundleAsync(bundleId);
```

This is the route for content a peer decides it needs on its own — a server hotloading a zone it is about to spawn from, or a client preloading ahead of a gate. It runs through the registered `IBundleLoader`. On a client it is only a preload: the load is reported to the server only if the server has already asked for that bundle and is still waiting on the answer. Otherwise the content loads locally and nothing is sent, and the server still asks for the bundle when a spawn needs it; the client's answer to that request is what the server records. This is also the call a bundle-backed scene load makes to bring its content in.

Reporting outcomes

`BundleManager.NotifyBundleLoaded(ushort bundleId)` and `NotifyBundleUnloaded(ushort bundleId)` report an outcome to the server. `LoadBundleAsync`/`UnloadBundleAsync` call these automatically; call them directly only when a game drives loading itself outside an `IBundleLoader`. Either way, a load report is sent only in answer to a request the server made, and an unload report only for a bundle the client has already told the server it holds; anything else stays on the client.

On the server, `Connection.IsBundleLoaded(ushort bundleId)` (Pro) reports whether a client holds a bundle. `Connection.BaseBuildBundleId` (0) is always loaded, since it shipped inside the build.

Watching the protocol

Two events on `BundleManager`:

```
coreManager.BundleManager.BundleLoadRequested += OnBundleLoadRequested;      // client: se
rver asked for a bundle
coreManager.BundleManager.BundleLoadStateChanged += OnBundleLoadStateChanged; // server: a
 client's bundles changed
```

Or implement `IBundleObserver` and register it with `RegisterBundleObserver`:

```
public interface IBundleObserver
{
    void OnBundleRequested(Connection connection, ushort bundleId);
    void OnBundleLoadStateChanged(Connection connection, ushort bundleId, bool isLoaded);
}
```

`OnBundleLoadStateChanged` fires only on an actual change, after the Connection's held set and interest have already been updated, so a handler reading `Connection.IsBundleLoaded` inside it sees the new state. This side is mostly a server concern: the server is the side that tracks what each client holds.

Violations

The server drives every load: it asks, then awaits the answer. Two violations catch a client that does not play by that:

- `UnsolicitedBundleReportViolation` — a load report for a bundle the server never asked for, or an unload report for a bundle the server does not record the client holding. This is how a spoofed load report is caught.
- `UnexpectedBundleRequestViolation` — a bundle load *request*, a message only the server ever sends, arriving at the server instead.

Both default to a kick.

Pro and Free

`BundleManager` is Pro. In a Free build, `CoreManager` has no `BundleManager` member at all. `IBundleLoader`, `IBundleObserver`, `BundleInterestCondition` and `Connection.IsBundleLoaded` are Pro as well. The two violation types, `Connection.BaseBuildBundleId`, and `NetworkSystem.RequiredBundleId` with `SetRequiredBundleId` live in plain files usable in either edition.

Networked Scenes, Content Bundles and Persistence

Pages in this category:

- How networked scenes work
- Opening and closing scene instances
- Placing clients in scenes
- Registering a scene loader
- Choosing what a save keeps
- Scene events and observers
- Moving and carrying objects between scenes
- Refusing a scene load
- Saving and loading a world
- Scene visibility: what a client in a scene can see
- The content bundle protocol
- Scenes and bundles: things that look broken
- Writing a world store

Moving and carrying objects between scenes

Moving a system to another scene

`SystemManager.EnsureMoveSystemToScene(NetworkSystem networkSystem, uint destinationSceneHandle)` is the one call that moves a spawned object between scene instances. It is server-only, and it returns `false` without moving anything if the system is not started and registered, if it is a scene object or has no platform identity (only a dynamically spawned object carries a scene handle to move), or if `destinationSceneHandle` is `NetworkSystem.UnsetSceneHandle`.

```
coreManager.SystemManager.EnsureMoveSystemToScene(asteroid, destinationSceneHandle);
```

If `networkSystem` belongs to a group, every member is restamped with the new scene together; a group shares one engine object and moves as a unit. The object's controller is carried along: the move requests the scene load for the controlling client before this tick's interest pass runs, so the client is walked into the destination scene rather than losing the object out from under it.

Reading a system's scene

`NetworkSystem.SceneHandle` names the scene instance the system currently belongs to. `NetworkSystem.UnsetSceneHandle` (zero) means the scene every peer already booted into - the boot scene needs no handle, since nothing has to carry a client into a scene it already holds.

`NetworkSystem.SceneChanged` fires whenever the handle changes: on the server as it performs the move, and on a client as it applies the change.

```
networkSystem.SceneChanged += (previousSceneHandle, currentSceneHandle) =>
{
    // previousSceneHandle: the scene the system was in before the move.
    // currentSceneHandle: the scene it is in now.
};
```

When the destination can't take it

Carrying an object into another scene means walking that object's controlling client into the destination scene along with it. Two different things can go wrong on that client's end, and they are not the same failure:

- **A carry failure.** The client's scene load for the destination scene fails outright. The object has lost the controller that was meant to follow it.
- **A refusal.** The client's scene loader will not hold another scene at once, so the load is refused rather than failing - the fix is placing the client with `SceneReplaceMode.AllScenes` instead of loading additively. Nothing about the object or its content was wrong, and the object is still perfectly reachable once the client is placed correctly. A refusal deliberately does not trigger carry-failure resolution; despawning or surrendering control here would punish the object for a load the engine itself asked for additively.

Deciding what happens on a carry failure

`SceneCarryFailureAction` is the outcome for an object whose controlling client failed to load the destination scene:

- `SurrenderControl` - return the object to the server's control and leave it where it is. The default. Destroys nothing and strands nothing: the client stops driving something it can't reach, and the object stays live for whoever is actually in the scene.

- `Despawn` - despawn the object (the whole group, since a group is one object). What a player's avatar usually wants, since the client that was going to drive it never arrived.
- `Retain` - leave the object exactly as it is, controller included. The engine does nothing further, so the game is expected to resolve the failure itself, typically by moving the object back and asking the client into that scene instead.

The action is resolved once, per carried object, on the failing edge, from three layers:

`SceneManager.SceneCarryFailure` - the world-wide default. Defaults to `SceneCarryFailureAction.SurrenderControl`.

`NetworkSystem.SceneCarryFailureOverride` - a nullable per-object override, for the cases where the blanket default reads wrong (a player's avatar in a world where everything else should surrender control, say). Null takes the manager's default.

A registered `ISceneCarryFailureResolver` - decides case by case, and outranks both of the above.

```
public SceneCarryFailureAction? SceneCarryFailureOverride { get; set; }
```

Resolving case by case

```
public interface ISceneCarryFailureResolver
{
    SceneCarryFailureAction ResolveSceneCarryFailure(
        Connection connection,
        NetworkSystem networkSystem,
        uint sceneHandle,
        SceneCarryFailureAction defaultSceneCarryFailureAction);
}
```

`defaultSceneCarryFailureAction` is what would happen without this resolver - the object's override, or the manager's default if it has none. Returning it unchanged is how a resolver declines a case, so a resolver only has to speak about the objects it cares about; several resolvers can be registered at once and each hands the next the answer so far.

Register and unregister a resolver on the `SceneManager`:

```
coreManager.SceneManager.RegisterSceneCarryFailureResolver(resolver);
coreManager.SceneManager.TryUnregisterSceneCarryFailureResolver(resolver);
```

In Unity

A root GameObject bound to its scene follows the move automatically. A parented GameObject is refused and stays in its parent's scene. An object whose scene binding is not yet resolved is parked until it resolves. See the Unity scene events page for the detail.

Opening and closing scene instances

Opening an instance

`SceneManager.TryOpenScene(ushort sceneId, SceneScope sceneScope, out uint sceneHandle, SceneReplacePolicy sceneReplacePolicy = SceneReplacePolicy.Released)` opens a live instance of a scene on the server and returns a handle for it. There is an awaitable form, `OpenSceneAsync(ushort sceneId, SceneScope sceneScope, SceneReplacePolicy sceneReplacePolicy = SceneReplacePolicy.Released)`, which does the same thing and completes once the scene has loaded.

```
if (sceneManager.TryOpenScene(myDungeonSceneId, SceneScope.Connections, out uint sceneHand
le))
{
    // sceneHandle identifies this instance.
}

uint sceneHandle = await sceneManager.OpenSceneAsync(myLobbySceneId, SceneScope.Global);
```

Opening a scene id twice gives two handles and two independent worlds; nothing about opening the same id again reuses or merges state with the first instance. Handles are never recycled, so a handle you have seen always refers to the same instance, and a new instance never reuses an old, closed handle's number.

Scope: who gets placed

`SceneScope` decides who the instance is for, and it cannot be changed after opening:

- `SceneScope.Connections` - private to whichever clients the game explicitly places into it later. Opening places nobody by itself.
- `SceneScope.Global` - every client belongs in it. Opening places every already-connected client into it immediately, and a client that authenticates after the instance is already open is placed into it as part of authentication.

A `SceneScope.Connections` instance places nobody until the game asks for it. A `SceneScope.Global` instance is the one case where opening itself places clients, on top of the join-time placement every scope shares.

Replace policy: what survives a replacement

`SceneReplacePolicy`, chosen at open time, decides what happens to this instance when a client already in it is later replaced out by a different placement:

- `SceneReplacePolicy.Released` (the default) - a replacing placement releases this instance along with everything else the client holds.
- `SceneReplacePolicy.Retained` - a replacing placement leaves the client in this instance and only releases the rest.

An explicit close or unload request always takes effect regardless of replace policy; the policy only exempts an instance from being swept aside by a *different* placement.

Reading back what is open

- `OpenSceneHandles` - the handles of every currently open instance.
- `TryGetOpenScene(uint sceneHandle, out ushort sceneId)` - which scene id a handle belongs

to.

- `TryGetSceneScope(uint sceneHandle, out SceneScope sceneScope)` - the scope it was opened with.
- `TryGetSceneReplacePolicy(uint sceneHandle, out SceneReplacePolicy sceneReplacePolicy)` - the replace policy it was opened with.

Closing an instance

`EnsureCloseScene(uint sceneHandle)` and the awaitable `CloseSceneAsync(uint sceneHandle)` are the ordinary way to end an instance: occupants are asked to leave before the instance itself comes down.

If an instance goes away without being closed through the manager - unloaded out from under it - call `TryNotifySceneUnloadedUnexpectedly(uint sceneHandle)` to tell the manager. It despawns the instance's systems and removes its occupants, since nothing else will.

Placing clients in scenes

Using Unity? See UnitySceneManager component.

Placing a client in a scene instance is what entitles it to see the objects inside. `SceneManager` drives every load and unload; the client never decides on its own.

Loading a connection into a scene

`TryRequestSceneLoad(Connection connection, uint sceneHandle, SceneReplaceMode sceneReplaceMode = SceneReplaceMode.None, uint systemId = NetworkSystem.UnsetId)` asks one client to load a live scene instance. It is deduped per connection and scene: a scene the client already holds, or has already reported it cannot load, is not asked for again. It returns `true` only when a request was actually sent.

```
coreManager.SceneManager.TryRequestSceneLoad(connection, sceneHandle);
```

There is also an overload for several clients at once:

```
uint requestedCount = coreManager.SceneManager.TryRequestSceneLoad(connections, sceneHandl
e, SceneReplaceMode.None);
```

`TryRequestSceneLoad(IReadOnlyList<Connection> connections, uint sceneHandle, SceneReplaceMode sceneReplaceMode = SceneReplaceMode.None)` places each client independently: one refused by a validator, already in the scene, or holding a recorded failure does not stop the rest. It returns how many clients a request was actually sent to, which tells you when the placement did less than you asked.

Removing a connection from a scene

`RequestSceneUnload(Connection connection, uint sceneHandle)` asks a client to release a scene instance it holds, or is on its way into. The scene a client booted into (`NetworkSystem.UnsetSceneHandle`) cannot be released this way; every peer is considered to be in it and none can report leaving it.

The client stops being served that scene's objects immediately, not when it confirms the release. `RequestSceneUnload` marks the client as leaving and drops it as an observer of that scene's systems before the unload message even goes out, so the interest gate withholds anything new for that scene right away, and existing observers are torn down as part of the call rather than left standing until the client answers.

Replacing what a connection holds

`SceneReplaceMode` controls what a load does to the scenes a client already holds:

- **`None`** (the default) - keeps everything the client holds and adds this scene to it.
- **`AllScenes`** - releases every server-opened scene the client holds or is loading, then loads this one. The target is validated before anything is released, so a refusal leaves the client where it was.
- **`AllScenesAfterLoad`** - loads this scene first, and only releases the client's other scenes once it confirms this one and acknowledges the state that went out with the confirmation. This is what to use when an object needs to move with the client rather than be rebuilt: releasing first would destroy the client's copy of everything in the scene it's leaving before the move can ride across.

The replace form exists because the removal half is easy to forget: passing `SceneReplaceMode.None`

and manually calling `RequestSceneUnload` for every other scene the client holds means finding all of them yourself and getting the ordering right. `AllScenes` and `AllScenesAfterLoad` fold that bookkeeping into the load call itself.

Where a newly authenticated client lands

`SceneManager.JoinPlacement` (type `JoinScenePlacement`) decides what a client is placed into the moment it authenticates, before the game asks for anything:

- **`EveryOpenScene`** (the default) - every live scene instance the server has open, whatever it was opened for. This overrides `SceneScope`, so a `SceneScope.Connections` arena still gets every new client while this is set.
- **`GlobalScenes`** - every `SceneScope.Global` instance, and nothing else. This is the answer once a world starts instancing: shared scenes opened `Global` still place everyone, but `Connections` instances place nobody.
- **`None`** - nothing. The game places every client itself, entirely through `TryRequestSceneLoad`.

Reading occupancy

`Connection.IsSceneLoaded(uint sceneHandle)` answers the question for one client: it returns `true` for `NetworkSystem.UnsetSceneHandle` (the scene every peer booted into) and otherwise reports whether that client has confirmed holding the scene.

`SceneManager.CollectConnectionsInScene(uint sceneHandle, List<Connection> connections)` answers it for a whole scene, adding every client that has confirmed the load to the supplied list. A client that has been asked and hasn't answered yet is not counted, and neither is one whose load failed. Asked about `NetworkSystem.UnsetSceneHandle` it adds every connected client.

Load requests that go unanswered

`SceneManager.LoadRequestTimeoutSeconds` (default 180, three minutes) bounds how long the server waits for a client to answer a load request. The request itself rides the reliable channel, so it always arrives; the timeout is for a client whose loader never finishes - a scene stuck fetching, a hung load, a client that stopped answering.

Setting it to zero or less turns the timeout off and the server waits indefinitely. When it expires, the request is cancelled and latched as a failure exactly as a client-reported failure is, and an answer that arrives afterward is discarded rather than acted on.

Placing a client from a blocked spawn

`SceneManager.AutomaticRequestOnBlockedSpawnEnabled` defaults to `false`. When `true`, a client would be placed in a scene the moment a spawn is withheld for want of it. It's off by default here, unlike the equivalent flag on the bundle manager (which defaults to `true`), because being in a scene is what *entitles* a client to its objects in the first place - inferring placement from a withheld spawn would pull every connected client into every open scene. Leave placement to a deliberate `TryRequestSceneLoad` call unless every client genuinely belongs in every scene you open.

Registering a scene loader

> **Using Unity?** See UnitySceneLoader component.

The core engine carries a scene identifier and nothing else, no path, no build index, no asset reference. It has no notion of what a "scene" is on any given platform, so it ships no `ISceneLoader` implementation. Until one is registered, scene requests do nothing.

The interface

```
public interface ISceneLoader
{
    uint MaximumConcurrentScenes { get; }

    Task<bool> LoadSceneAsync(uint sceneHandle, ushort sceneId);

    Task<bool> UnloadSceneAsync(uint sceneHandle);
}
```

`LoadSceneAsync` loads one live instance of the scene identified by `sceneId`, keyed under `sceneHandle` for every scene object that registers into it. `UnloadSceneAsync` releases the instance at that handle. Both return a task resolving to `true` on success.

Both methods may await freely. The network loop never blocks on them, and whatever the task resumes with is marshalled back onto the loop itself, so an implementation can await a download, an asset load, or any engine operation without worrying about which thread it ends up on.

Registering it

```
sceneManager.SetSceneLoader(mySceneLoader);
```

One loader per `SceneManager`. There is no default and no fallback, register it before any scene load is requested.

Concurrency limit

`MaximumConcurrentScenes` declares how many live scene instances this peer can hold at once. A loader that replaces the running world with each load can only ever hold one; a loader that adds a scene alongside what is already there can hold as many as it wants. Set it to `SceneManager.UnlimitedConcurrentScenes` (0) for no limit.

The engine enforces this limit itself, before the loader is ever consulted. A request that would exceed it never reaches `LoadSceneAsync`; it comes back as `SceneLoadOutcome.Refused` immediately.

Mapping ids to content

The wire never carries a path or an asset reference, only the `ushort sceneId`. The mapping from that id to actual content (a scene asset, an address, a bundle) is entirely the loader's own. Nothing in the core engine constrains how that table is built or where it lives.

The Unity integration supplies its own scene loader

The Unity integration ships two implementations of this interface, `NetworkSceneLoader` and `UnitySceneLoader`, so a Unity project does not need to write its own.

Scene events and observers

Events on SceneManager

`SceneManager` raises seven events. Each has its own delegate.

Event	Delegate	Raised on	When
`SceneLoadRequested`	`SceneLoadRequestedHandler(uint sceneHandle, ushort sceneId, bool isLoadRequested)`	client	The server asks the client to load or release a scene, before the loader is consulted.
`SceneLoadStateChanged`	`SceneLoadStateChangedHandler(Connection connection, uint sceneHandle, bool isLoaded)`	server	A client's answer changes which scenes it holds.
`SceneLoaded`	`SceneLoadedHandler(uint sceneHandle, ushort sceneId)`	client	This client's registered `ISceneLoader` finishes a load the server asked for, after the outcome has been reported to the server. Not raised on the server, a host included, for the server's own copy: server code awaits `OpenSceneAsync` instead, which returns the handle once the scene is loaded there, or `NetworkSystem.UnsetSceneHandle` if it could not be.
`SceneUnloaded`	`SceneUnloadedHandler(uint sceneHandle, ushort sceneId)`	client	This client's registered `ISceneLoader` finishes a release the server asked for, after the report is sent. Not raised on the server when it closes its own copy; await `CloseSceneAsync` there instead.

`SceneLoadFailed`	`SceneLoadFailedHandler(uint sceneHandle, ushort sceneId)`	This peer's own load fails, after the failure has been reported to the server. Also raised on the server when `OpenSceneAsync` could not load the server's own copy.
`ClientSceneLoadFailed`	`ClientSceneLoadFailedHandler(Connection connection, uint sceneHandle)`	A client reports it could not load a scene, after the pending request has been cancelled.
`SceneLoadProgressed`	`SceneLoadProgressedHandler(uint sceneHandle, ushort sceneId, float progress)`	This peer's own load advances, for a loading screen. Purely local; progress never reaches the wire.

```
sceneManager.SceneLoadRequested += (sceneHandle, sceneId, isLoadRequested) =>
{
    // Show a loading screen, then load or release sceneId.
};

sceneManager.ClientSceneLoadFailed += (connection, sceneHandle) =>
{
    // Decide what happens to a client that could not fetch the scene.
};
```

`SceneLoadStateChanged` fires only on a genuine change to what a client holds, and only once the record and interest re-resolution have already happened.

ISceneObserver: the read-only alternative

`ISceneObserver` covers the same ground as three of the events, for code that wants to watch without adding its own event subscription per manager instance:

```
public interface ISceneObserver
{
    void OnSceneRequested(Connection connection, uint sceneHandle, bool isLoadRequested);
    void OnSceneLoadStateChanged(Connection connection, uint sceneHandle, bool isLoaded);
    void OnSceneLoadFailed(Connection connection, uint sceneHandle);
}
```

Register any number of observers with `sceneManager.RegisterSceneObserver(observer)`, and remove one with `TryUnregisterSceneObserver`. Both the events above and the matching `ISceneObserver` callback are raised for the same occurrence, so use whichever form suits the call site. Registering the same observer twice is ignored; registering null is logged as an error.

Reporting from a loader that is not the shipped one

A registered `ISceneLoader` reports automatically. A game driving its own loading calls these instead:

- `NotifySceneLoaded(uint sceneHandle, uint systemId = NetworkSystem.UnsetId)` — this

client now holds the scene instance.

- `NotifySceneUnloaded(uint sceneHandle)` — this client has released a scene instance.
- `NotifySceneLoadFailed(uint sceneHandle, uint systemId = NetworkSystem.UnsetId)` — this client could not load a scene instance.
- `NotifySceneLoadProgress(uint sceneHandle, ushort sceneId, float progress)` — reports progress through a load, purely local. Nothing is sent; the server learns only whether the scene did or did not load, never how far along it got.

All three report methods are safe to call from any thread, and each is only sent in answer to a request the server actually made — a self-driven load the server never asked for loads locally and stays silent.

Clearing a stuck client

A reported failure latches, so the server stops asking a client that cannot fetch the content. Re-arm it once whatever blocked the load has been addressed:

```
bool cleared = sceneManager.TryClearSceneLoadFailure(connection, sceneHandle);
```

Check whether a client is currently latched with `connection.HasSceneLoadFailed(sceneHandle)`.

SceneLoadOutcome

A client reports one of four outcomes for a scene instance it was asked to load or release:

- **Loaded** — the client now holds the instance and can be served the objects inside it.
- **Released** — the client has released the instance it was asked to release.
- **Failed** — the client could not load the instance: content would not fetch, the scene would not resolve, or the load threw.
- **Refused** — the client's `ISceneLoader` is already at the limit set by `ISceneLoader.MaximumConcurrentScenes`; the loader was never consulted.

`Refused` and `Failed` both surface through `ClientSceneLoadFailed` and `ISceneObserver.OnSceneLoadFailed`, and neither raises `SceneLoadStateChanged` — the client holds nothing it did not hold before, so nothing about what it holds changed. Only `Failed` runs scene-carry-failure resolution for objects being carried into that scene; a refusal says only that the loader will not hold another scene at once, which the game fixes by moving the client with `SceneReplaceMode.AllScenes` rather than adding to what it holds, so nothing about the object itself is wrong.

Violations

A client that answers about a scene it was never asked about did not merely send a stray message — it is trying to place or remove itself from scenes on its own, which is how it would serve itself content it was never admitted to. The server drives every load and release itself, so these settle as violations rather than ordinary reports.

- **UnsolicitedSceneReportViolation** — a client reports loading a scene it was never asked to load, or releasing one it was never asked to release. Carries `SceneHandle` and `IsLoadReported` (true for a claimed load, false for a claimed release).
- **UnexpectedSceneRequestViolation** — a client sends a scene load request, the message that only ever travels server to client. Carries `SceneHandle` and `IsLoadRequested`.

Both default to a kick.

Refusing a scene load

What a validator does

`ISceneLoadValidator` gives the server a chance to refuse a client before it is ever asked to load a scene instance:

```
public interface ISceneLoadValidator
{
    bool CanLoadScene(Connection connection, uint sceneHandle, ushort sceneId);
}
```

- `connection` is the client the server is about to ask.
- `sceneHandle` identifies the live scene instance.
- `sceneId` identifies the scene asset that instance was opened from.

Return `true` to let the request go out, `false` to refuse it. A validator is a predicate and nothing else: it must not open scenes, send messages, or otherwise act on what it's deciding.

Register one with `SceneManager.RegisterSceneLoadValidator`, and remove it with `TryUnregisterSceneLoadValidator` when it no longer applies:

```
sceneManager.RegisterSceneLoadValidator(myValidator);

// later
sceneManager.TryUnregisterSceneLoadValidator(myValidator);
```

Several validators, first refusal wins

Any number of validators may be registered. When `TryRequestSceneLoad` is about to ask a client into a scene, the server walks the registered list in order and stops at the first one that returns `false`. If none are registered, or none refuse, the request proceeds. There's no voting and no override: one refusal is final, and validators after it aren't even asked.

This means validator order can matter if two validators disagree about edge cases, but it never matters for the common case of independent entitlement checks — each one only needs to say yes or no about its own rule.

The client never finds out

A refusal happens before the request is sent. Nothing crosses the wire: no rejection message, no scene reference, no acknowledgment that a scene existed at all. From the client's point of view a refused scene is indistinguishable from one that was never offered.

This is deliberate, not an oversight to work around. It's what makes `ISceneLoadValidator` the right place for entitlement rules — a party requirement, a ticket, a subscription, a ban — where the answer has to come from the server and can't be inferred or forged by a client that was told "no" and why.

Where a validator is the wrong tool

Capacity. How many live scene instances a peer can hold at once is `ISceneLoader.MaximumConcurrentScenes`, a property of the loading strategy itself, not a per-client decision. Don't write a validator that counts scenes and refuses past some number — that logic already belongs on the loader.

Per-object visibility. Deciding which objects inside a scene a given client can see is interest's job, not the scene load's. A validator only answers whether the client is placed into the scene instance at all; it has no say over what becomes visible once they're in.

A validator runs once per gated placement, for every client

`IsSceneLoadAllowed` runs the registered list once per gated placement — one call per validator until a refusal or the list ends. With no validators registered it's a single empty-list check, effectively free. A validator that does real work (a database lookup, a network call to an entitlement service) pays that cost on every scene placement it's asked about, for every client, so keep it to an in-memory check wherever possible, or cache the result behind whatever state already tracks the entitlement.

Registering a scene loader

Using Unity? See UnitySceneLoader component.

The core engine carries a scene identifier and nothing else, no path, no build index, no asset reference. It has no notion of what a "scene" is on any given platform, so it ships no `ISceneLoader` implementation. Until one is registered, scene requests do nothing.

The interface

```
public interface ISceneLoader
{
    uint MaximumConcurrentScenes { get; }

    Task<bool> LoadSceneAsync(uint sceneHandle, ushort sceneId);

    Task<bool> UnloadSceneAsync(uint sceneHandle);
}
```

`LoadSceneAsync` loads one live instance of the scene identified by `sceneId`, keyed under `sceneHandle` for every scene object that registers into it. `UnloadSceneAsync` releases the instance at that handle. Both return a task resolving to `true` on success.

Both methods may await freely. The network loop never blocks on them, and whatever the task resumes with is marshalled back onto the loop itself, so an implementation can await a download, an asset load, or any engine operation without worrying about which thread it ends up on.

Registering it

```
sceneManager.SetSceneLoader(mySceneLoader);
```

One loader per `SceneManager`. There is no default and no fallback, register it before any scene load is requested.

Concurrency limit

`MaximumConcurrentScenes` declares how many live scene instances this peer can hold at once. A loader that replaces the running world with each load can only ever hold one; a loader that adds a scene alongside what is already there can hold as many as it wants. Set it to `SceneManager.UnlimitedConcurrentScenes` (0) for no limit.

The engine enforces this limit itself, before the loader is ever consulted. A request that would exceed it never reaches `LoadSceneAsync`; it comes back as `SceneLoadOutcome.Refused` immediately.

Mapping ids to content

The wire never carries a path or an asset reference, only the `ushort sceneId`. The mapping from that id to actual content (a scene asset, an address, a bundle) is entirely the loader's own. Nothing in the core engine constrains how that table is built or where it lives.

The Unity integration supplies its own scene loader

The Unity integration ships two implementations of this interface, `NetworkSceneLoader` and `UnitySceneLoader`, so a Unity project does not need to write its own.

How networked scenes work

Three identifiers, never conflated

A networked scene involves three different numbers, and mixing them up is the most common source of confusion:

- **Scene id** (`ushort`) — which scene asset. The same id always resolves to the same content, through whatever `ISceneLoader` is registered.
- **Scene handle** (`uint`) — which live instance of that asset. Opening the same scene id twice gives two handles and two independent worlds.
- **Scene object id** (`uint`) — which object inside a scene file. It is stamped by editor tooling and is only unique within that one scene asset.

None of these substitutes for another. The id tells you what to load, the handle tells you which loaded copy you mean, and the object id only makes sense once you already know which instance it lives in.

SceneObjectKey

An object authored into a scene is identified by the pair of scene handle and scene object id together, `SceneObjectKey`. A `SceneObjectId` alone is ambiguous the moment two live instances of the same scene asset are open, since both instances stamped their objects from the same file and both objects carry the same id. Pairing it with the handle is what lets a peer holding several scene instances at once bind a spawn to the right one.

`NetworkSystem.SceneHandle` reports which instance an object belongs to. Its unset value is `NetworkSystem.UnsetSceneHandle` (zero), which means the object belongs to no server-opened scene — the scene a peer booted into, rather than one `SceneManager` opened and tracks.

The server drives everything

`SceneManager` opens instances, places and removes clients, and closes instances. A client never decides any of this for itself: it loads a scene only when asked, releases one only when asked, and reports back only what it was asked to report. That is what makes the server's record of who holds what trustworthy enough to serve objects from — a client cannot inflate what the server believes it holds by loading or reporting on its own initiative.

`SceneManager.TryOpenScene` and `OpenSceneAsync` open instances. `TryRequestSceneLoad` and `RequestSceneUnload` place and remove clients. `EnsureCloseScene` and `CloseSceneAsync` close instances. See Opening and closing scene instances and Placing clients in scenes for the full API.

Every instance coexists, nothing is replaced

Open scene instances are tracked by handle in a table the server owns, and closing one only ever removes that one entry. Nothing about opening a scene id again reuses or merges state with an instance already open for that id. Handles are handed out from a counter that only climbs and is never recycled, so a handle you have seen always names the same instance for as long as anything remembers it, and a later instance can never be confused with an earlier, closed one.

This is what makes scene stacking possible: opening one asset several times gives you several coexisting instances, each with its own handle, and the handle is the only thing that tells them apart — they still share the same world coordinates. The shipped Unity scene loader (`UnitySceneLoader`) implements this by loading each instance additively (`LoadSceneMode.Additive`), so several copies of one Unity scene can

be open in the same process at once. That is the loader's choice of how to realize the model, not a rule the model itself imposes.

SceneScope: who an instance is for

`SceneScope` is decided once, when the instance is opened, and cannot change afterward:

- `SceneScope.Connections` — nobody is placed until `TryRequestSceneLoad` asks for it. The instancing case: an arena, a dungeon, a match.
- `SceneScope.Global` — every connected client is placed immediately, and every client that authenticates afterward is placed as it arrives. The single-world or lobby case.

Because scope is per instance rather than per asset, one scene asset can run as a shared `Global` lobby while a dozen other instances of a different (or the same) asset run as private `Connections` matches, all open at once.

The wire carries a number

A scene is never sent as a path or a name. Every message that names a scene carries the scene id or the scene handle — plain numbers — and the registered `ISceneLoader` is what turns an id into actual content on each peer.

Where to go next

- UnitySceneLoader component and UnitySceneManager component for the Unity setup.
- Opening and closing scene instances and Placing clients in scenes for the API.

Scene visibility:what a client in a scene can see

The gate

`SceneManager` owns a `SceneInterestCondition` called `SpawnGate`. It has `Capabilities => InterestEffect.Spawn`, and `SceneManager` registers it on startup as one of the engine's default conditions (listed in `InterestManager.DefaultConditions`), so it applies to every `NetworkSystem` without any per-game setup. A game that wants to arbitrate scene membership itself can pull it back out with `InterestManager.RemoveDefaultConditions`.

The rule it enforces: a `NetworkSystem` whose `SceneHandle` names a live scene instance is withheld from a `Connection` that does not hold that scene. `NetworkSystem.UnsetSceneHandle` (a system that belongs to no server-opened scene) is exempt and is always held by everyone, so a world that never opens a scene pays almost nothing for this gate being on.

Spawning and ongoing replication

`SpawnGate` only carries `InterestEffect.Spawn`, but it is not just a spawn-time check. The gate stops serving a scene to a client the moment `SceneManager.RequestSceneUnload` asks it to leave, not when the unload finishes, even though `Connection.IsSceneLoaded` keeps answering true until the client confirms. `RequestSceneUnload` drops the connection as an observer of that scene's systems immediately, before any unload confirmation comes back. `SceneInterestCondition.Evaluate` mirrors this: while that unload request is outstanding, the condition returns `InterestResult.Spawn(isMet: false)` instead of abstaining, withholding the object rather than re-serving it into a scene the client is on its way out of.

So "the gate only restricts spawning" is only true for a client that fully holds the scene start to finish. A client mid-leave is cut off before the unload is confirmed.

Local and emulated peers

The condition abstains for a local peer, returning `InterestResult.None` immediately. A local peer is not a host's own player connection. It is an emulated `Connection`, or any peer sharing an in-memory Transport (test setups, most commonly), and it is deliberately distinct from `Connection.IsHostLoopback`, which is what actually names a host's own client half. A host's own client is evaluated by this gate like any other connection; it is placed by the same scene protocol and holds a real membership record.

What differs for a host's client is how that record is used: the result is surfaced through `NetworkSystem.HostInterestMembership` rather than acted on, which is what lets a host hide an object its client half would not have been sent without actually withholding anything from itself as the server.

Carried systems

A client that is being moved into a scene does not have to wait for the scene load to land before its own controlled object shows up. When the system is a spawned object rather than a scene object, the connection controls it, and the connection has been asked into the system's scene with that load still outstanding, the condition abstains (`InterestResult.None`) rather than withholding. This is checked after the held-scene branch and after the leaving-scene branch, so a client that already holds the scene never needs it, and a client that has been asked to leave never gets it either. Only a client on its way in, for a system it drives, gets served across the gap.

Automatic scene requests

By default a blocked spawn does nothing beyond withholding the object. Setting `SceneManager.AutomaticRequestOnBlockedSpawnEnabled` to `true` turns that block into an action: when `Evaluate` is about to withhold a spawn, it also calls `SceneManager.TryRequestSceneLoad` for that connection and scene, naming the system so the load's confirmation can arrive as a spawn on the same tick rather than waiting for the next stagger slot. The request dedupes per connection and scene, and stops firing once that client has reported it cannot load the scene.

Why stacked scenes cannot see each other

Two open instances of the same scene occupy the same world coordinates — nothing about position tells them apart. Only membership does. `SceneHandle` on a `NetworkSystem`, checked against what `Connection.IsSceneLoaded` reports for that client, is the only thing separating a player in instance A from an identical-looking object sitting in the same spot in instance B. Without this gate a distance-based condition would happily match a player in one instance against every object in every other instance at the same coordinates.

Troubleshooting

A client should be seeing an object and isn't.

- Check `NetworkSystem.SceneHandle` on the object. If it is not `NetworkSystem.UnsetSceneHandle`, the client must hold that exact scene instance.
- Check `Connection.IsSceneLoaded(sceneHandle)` for that client. If it is false, the object is being withheld correctly — the fix is getting the client into the scene (`SceneManager.TryRequestSceneLoad`), not the interest system.
- If the client was recently told to leave the scene (`RequestSceneUnload`), everything in it is withheld immediately, before the unload confirms. This is expected, not a bug.
- If the object is the client's own controlled object arriving right as it enters the scene, confirm the carry conditions hold: the system must be a spawned object rather than a scene object, the scene load must be outstanding for that connection, and the connection must control the system's group.
- On a genuinely local setup, an emulated connection or a peer sharing an in-memory Transport, `SpawnGate` abstains; this does not apply to a real host's own client, which is gated normally and reported through `NetworkSystem.HostInterestMembership` instead.
- If none of the above applies and the object still never arrives, check whether `InterestManager.RemoveDefaultConditions` was called somewhere, removing `SpawnGate` entirely.

Scenes and bundles: things that look broken

A client connects and sees nothing

`SceneManager.JoinPlacement` decides what a newly authenticated client is put into, and its default, `JoinScenePlacement.EveryOpenScene`, is what most projects want without calling `TryRequestSceneLoad` anywhere. If `JoinPlacement` was set to `JoinScenePlacement.None`, no joining client is placed automatically, and under `JoinScenePlacement.GlobalScenes` a scene opened `SceneScope.Connections` rather than `SceneScope.Global` is skipped at join. Either way, a client that never receives a `TryRequestSceneLoad` call for that scene sees nothing.

The other cause is timing rather than configuration: a `SceneScope.Connections` scene opened *after* a client already joined is not pushed to that client. Join placement happens once, when the client authenticates, and only covers the scenes open at that moment. A `SceneScope.Global` scene is the exception: opening it places every client already connected. A client connected before a `Connections` scene existed has to be asked for it explicitly.

Check `JoinPlacement`, the scope the scene was opened with, and, for a `Connections` scene, whether it existed before the client connected.

Objects appear in the wrong copy of a stacked scene

A stacked scene is one asset opened more than once, so every copy shares the same name and path — `GetSceneByName` and `GetSceneByPath` cannot tell them apart, only the scene handle can. `UnitySceneManager.TryGetScene` resolves a handle to the live Unity `Scene`, and that resolved scene is what has to be made active, or the target for `MoveGameObjectToScene`, before anything spawns into it.

The handle itself is never the cause: a handle is issued once per open scene instance and never reused across opens. If objects are landing in the wrong copy, look at the spawner instead: it instantiated while the wrong Unity scene was active, or it moved the wrong scene's copy.

A second, unrelated cause produces the same symptom for a parented object: the Unity integration only moves a root `GameObject` between scenes, because that is the only kind of move Unity itself supports. An object with a parent is refused rather than moved and stays in its parent's scene, and an error naming the parent is logged. Un-parent the object, or move its parent instead.

A load never completes

Two different timeouts produce this, and they are not the same failure:

- `SceneManager.LoadRequestTimeoutSeconds` (default 180 seconds, zero or less waits indefinitely) is how long the server waits for a client to answer a scene load request. If the client never answers, the server treats it as a failure, logs a warning naming the scene handle and the timeout, and raises `ClientSceneLoadFailed`.
- On the client, nothing completes if the loader never calls back. A custom `ISceneLoader.LoadSceneAsync` implementation has to eventually call `SceneManager.NotifySceneLoaded` (or fail through `NotifySceneLoadFailed`) — a loader that awaits something that never resolves leaves the load hanging until the server's own timeout catches it.

If the load is stuck client-side with no timeout firing at all, check that the loader in use actually reports back through `NotifySceneLoaded`.

A load comes back as Refused rather than Failed

`SceneLoadOutcome.Refused` means the client's `ISceneLoader.MaximumConcurrentScenes` was already at its limit — the loader was never consulted, because the request was rejected before it got there. `SceneLoadOutcome.Failed` means the loader was asked and the load itself did not work: the content would not fetch, the scene would not resolve, or the load threw.

The two are deliberately distinct outcomes because the remedy differs. A `Failed` load can just be asked for again with `SceneManager.TryClearSceneLoadFailure`. A `Refused` load cannot — asking again hits the same limit — the client has to be moved with `SceneReplaceMode.AllScenes` instead, so it releases what it holds before taking on the new one.

A client is kicked during scene or bundle traffic

Four violations cover unexpected scene and bundle messages, and all default to a kick:

- **`UnsolicitedSceneReportViolation`** — a client reported a scene load or release the server never asked for. Carries `SceneHandle` and `IsLoadReported` (true for a claimed load, false for a claimed release).
- **`UnexpectedSceneRequestViolation`** — a client sent a scene load request, a message that only ever travels server to client. Carries `SceneHandle` and `IsLoadRequested`.
- **`UnsolicitedBundleReportViolation`** — a client reported a bundle load or unload the server never asked for, or does not record it holding. Carries `BundleId` and `IsLoadReported`.
- **`UnexpectedBundleRequestViolation`** — a client sent the server a bundle load request, a message that only the server sends. Carries `BundleId`.

All four are a client trying to drive placement or content that only the server is allowed to drive. A well-behaved client cannot produce any of them, so seeing one means a modified or malicious client, not a legitimate race — including the unset scene handle, which is never requested and never recorded held, and which reports as unsolicited too.

Prefabs silently never spawn for one client

`BundleInterestCondition` withholds a `NetworkSystem`'s spawn from a `Connection` until that client holds the content bundle its prefab shipped in (`NetworkSystem.RequiredBundleId`), so the spawn is never delivered to a peer that cannot instantiate it — no error, the object just does not exist there yet. It abstains for a scene object (already bound rather than instantiated), for the base build (`Connection.BaseBuildBundleId`), and for a host's own local or loopback connection.

Whether the client is ever told to fetch the missing bundle depends on `BundleManager.AutomaticRequestOnBlockedSpawnEnabled`. When it is on, the withheld spawn triggers `BundleManager.RequestBundleLoad` for that connection and bundle automatically. When it is off, nothing asks the client for the bundle — the spawn stays withheld until the client is given the bundle some other way. If an object is missing for exactly one client and no error appears, check whether that client holds the required bundle (`Connection.IsBundleLoaded`) and whether automatic requesting is enabled.

A load brings a world back missing objects

Two different lookups can fail on a world load, and each is logged and skipped rather than aborting the whole load:

- A saved scene instance names a `sceneId` the manifest no longer resolves. `WorldPersistenceManager` reopens each saved scene; if that fails, `IWorldStore.OnSceneUnavailable` decides what happens: either the whole load is abandoned (logged as "the world is empty rather than half built"), or the objects in that scene are dropped and the

rest of the load continues.

- A saved object's identity no longer resolves. For a scene object, `WorldSystemIdentity.PlatformId` is looked up in the restored scene; if nothing matches, that object is dropped and logged as not found. For a constructed object, `WorldSystemIdentity.SystemTypeName` is looked up with `NetworkTypeRegistry.TryCreateSystemByName`; a type no longer registered in this build is dropped the same way.

Missing objects after a load are one of these two lookups failing, not a corrupted save — check the log for which scene id or system type it named.

In Unity: a duplicated prefab or scene object carrying a copied stamped id

Copying an object that already carries a stamped id copies the id with it, and two objects sharing one id is a correlation collision.

For scene objects, `SceneNetworkObjectStamper` runs on every scene save: it keeps the first occurrence of each `SceneObjectId` and reassigns any later duplicate to the next free value, so a duplicated scene object gets a fresh id automatically the next time the scene is saved. Until that save happens, the duplicate still carries the copied id.

For a duplicated prefab, the network id is stamped onto the prefab asset itself, so copying the prefab duplicates its id along with it and a spawn built from the copy resolves to the wrong object. Nothing re-stamps it on save, but **Nucleus > Rebuild Network Prefab Collection** does: the original keeps its id, because the collection already lists it, and the copy is given the next free id. A prefab variant that inherited its base's id is renumbered the same way, and its base keeps the id. Then rebuild both the server and the client builds so every peer carries the same ids. A player build made before the rebuild fails with the clash listed, and the registry logs an error naming both prefabs if two that share an id are ever registered together, so the problem shows up in the Console rather than as a wrong spawn.

Saving and loading a world

Using Unity? See Saving and loading a world from Unity.

WorldPersistenceManager

`CoreManager.WorldPersistenceManager` writes the server's world to a store and builds it back. Register a store first:

```
coreManager.WorldPersistenceManager.SetWorldStore(myWorldStore);
```

`SetWorldStore` replaces whatever store was previously registered; passing `null` unregisters it. `SaveAsync` and `LoadAsync` both fail immediately if no store is registered.

`IsBusy` is true for the duration of either operation. A second `SaveAsync` or `LoadAsync` called while one is already running is refused rather than allowed to interleave with it.

Saving

```
bool saved = await coreManager.WorldPersistenceManager.SaveAsync();
```

`SaveAsync` walks every open scene and every started system into the store's document, then awaits the store's own save. The store is arbitrary caller code, so the save can span any number of ticks before it resolves.

Loading, and the deadlock rule

`LoadAsync` despawns and rebuilds the routing table, which only the network loop thread may touch. Because the store's load can resolve on any thread, `LoadAsync` marshals itself back onto the loop before it starts rebuilding.

This means the thread that drives the loop must never block on it. Blocking that thread on the task (`.Result`, `.Wait()`) stops the loop from ever draining, the marshal never resolves, and the load deadlocks permanently. An ordinary `await` does not block: it hands the thread back, so the loop keeps being driven and the load finishes. The caller must keep the loop being driven, the normal way the host application drives it, for as long as the returned task is outstanding.

```
Task<bool> loadTask = coreManager.WorldPersistenceManager.LoadAsync();

bool loaded = await loadTask;
```

Await it from any thread, the loop-driving one included, and this is all it takes; the loop keeps running and the task resolves once the build finishes. What the loop-driving thread must not do is block on the task, or await it inside the code that would otherwise go on to drive the next tick.

Authority only

Both `SaveAsync` and `LoadAsync` are refused on a peer whose server is not started. A client holds a copy of the server's world, not one of its own, so there is nothing on a client worth saving and nothing a client can build without a server behind it.

What a save keeps

A save records:

- What objects existed

- What type each object was built from
- Where each object lived (its scene)
- What each object belonged to
- What each object's members held

What a save drops, and why

A save does not record observer sets, interest membership, access rosters, acknowledgment positions, or controller assignments. All of these describe peers connected to a session, and a saved world outlives the session it was taken from. There is nothing to restore them to.

Unregistered system types are skipped silently, per object

A save only writes systems whose type is registered with `NetworkTypeRegistry`. A started system of an unregistered type is left out of the save entirely, and an error is logged for that object naming its id and type. The rest of the save proceeds; there is no way to detect after the fact which objects were skipped other than by reading the logged errors.

Loading replaces, it does not merge

A load first despawns everything currently standing, then builds the saved world in its place. Connected clients see exactly that sequence: a world of despawns followed by a world of spawns.

If no `ISceneLoader` is registered on `SceneManager`, a load logs a warning and drops every object inside every scene named by the saved world, because none of those scenes can be opened.

Scenes are reopened before objects are rebuilt

A load closes all currently open scenes, then reopens each scene the saved world names before rebuilding any object into it. If a named scene cannot be opened, the store itself decides what happens next through its failure action: it can either let the load continue without that scene's objects, or fail the load outright, in which case no saved object is built. The world the load replaced does not come back either, because it was cleared before the saved scenes were read, so a failed load leaves an empty world.

Whole-world only

A save always captures the entire world, and a load always replaces the entire world. There is no incremental or partial save.

World persistence ships only in Pro

World persistence is a Pro feature.

Choosing what a save keeps

WorldStateIgnoreAttribute

`WorldStateIgnoreAttribute` (`Nucleus.Managers.Persistence`) keeps a member out of a saved world while leaving everything about how it replicates alone. Applied to a `NetworkMemberBase` field on a `NetworkComponent`, it excludes that one member. Applied to the component class itself, it excludes every member the component declares.

```
public partial class WorldStateIgnoreStateComponent : NetworkComponent
{
    public readonly NetworkMember<int> Persisted = new();

    [WorldStateIgnore]
    public readonly NetworkMember<int> Ignored = new();

    public readonly NetworkMember<int> AlsoPersisted = new();
}

[WorldStateIgnore]
public partial class WorldStateIgnoreComponentStateComponent : NetworkComponent
{
    public readonly NetworkMember<int> Counter = new();
}
```

The exclusion is compile-time. The source generator does not emit the marked member's line into `WriteWorldState` or `ReadWorldState`, so there is no runtime check paying for it on every save. A save written by an earlier build that still contains the member is simply never asked for it. After a load, the member keeps the value its component was constructed with — not a stored value, not the value from before the save, the constructed default.

Nothing about the wire changes. The member keeps its member Id, its flag bit, and its place in the component's serialized body. Marking a member costs no bandwidth and does not renumber anything around it.

WorldStateIgnore vs. ReplicationIgnore

`ReplicationIgnoreAttribute` (`Nucleus.Components`) is the inverse: it keeps a member off the wire while leaving how it is saved alone. Same two targets — a field for one member, the class for all of them.

That pairing is the whole of what separates the two decisions:

Attribute	Opts out of	Member still replicates?	Member still saved?
`WorldStateIgnoreAttribute`	the save	yes	no
`ReplicationIgnoreAttribute`	the wire	no	yes, from whichever peer wrote it

The adoption trap

A peer promoted to server by adoption never received `ReplicationIgnore`-marked values — nothing was ever sent to it for them. It holds the constructed defaults, not whatever the previous server had. Saving a world from that peer writes those defaults over whatever the store already held for those members. This is

specific to `ReplicationIgnoreAttribute`; a `WorldStateIgnore` member behaves the same on every peer regardless of role, because no peer ever saves it.

Choosing between them

- **A session counter or tick stamp** — worth replicating so observers see it live, not worth restoring after a load: `WorldStateIgnore`.
- **A visual highlight or other purely presentational flag** — often neither needs saving nor needs to leave the peer that set it, so either attribute can apply depending on whether other peers need to see it.
- **A cached local reference** (an object handle, a lookup result recomputed on start) — not meaningful to another peer and not meaningful after a load: mark it `ReplicationIgnore` if it must never leave the peer that holds it, or `WorldStateIgnore` if it should still replicate live but not survive a save.

Free vs. Pro

`WorldStateIgnoreAttribute` and its generator support are not Pro-gated. They compile and work in a Free build. The attribute simply has no observable effect until Pro's world persistence is actually saving and loading worlds — there is nothing to exclude a member from until then.

Not the same as NetworkIgnoreAttribute

`Nucleus.Serializers.NetworkIgnoreAttribute` is unrelated to both of the above. It excludes a field from a serialized type — the shape the serializer generates for a struct or class passed over the wire — not a `NetworkComponent` member's participation in replication or saving.

Writing a world store

The seam

A saved world has no file, no format and no address defined by the engine, only a shape. `IWorldStore` is that seam: implement it to persist a world somewhere other than a file, a database row, a save-service blob, cloud storage, whatever the game already uses. `JsonWorldStore` ships as the default implementation, both as a working store and as the reference to read alongside your own.

Almost every file behind this feature is Pro-only (`.Pro.cs`): `IWorldStore`, `IWorldDocument`, `WorldSystemIdentity`, `WorldLoadFailureAction`, `JsonWorldStore` and `WorldDocument`. The two exceptions are `IWorldStateWriter` and `IWorldStateReader`, which ship in both editions.

Register a store through `WorldPersistenceManager.SetWorldStore`. The engine consults exactly one store at a time.

```
CoreManager.WorldPersistenceManager.SetWorldStore(new JsonWorldStore(CoreManager, filePath
));
```

IWorldStore

```
public interface IWorldStore
{
    Task<bool> SaveAsync(IWorldDocument worldDocument);
    IWorldDocument BeginSave();
    Task<IWorldDocument> LoadAsync();
    WorldLoadFailureAction OnSceneUnavailable(ushort sceneId, uint sceneHandle);
}
```

- `BeginSave` returns an empty document for the engine to write the whole world into. It is separate from `SaveAsync` so nothing is committed until the write is complete; a failed save leaves the previous one intact.
- `SaveAsync(IWorldDocument)` keeps the document the engine just finished writing. Return `true` once it is safely stored.
- `LoadAsync` returns the document to read a saved world out of, or `null` when there is nothing to load.
- `OnSceneUnavailable(ushort sceneId, uint sceneHandle)` is asked once per scene instance the save names that this build cannot open, before any object from it has been built. Answer with a `WorldLoadFailureAction`.

The work is genuinely asynchronous. The engine does not block on it and does not care which thread it completes on; completion is marshalled back onto the network loop before anything is built.

IWorldDocument

A document is streamed, not handed over whole, so neither saving nor loading builds a second copy of the world in memory. Scenes are written and read before objects, because an object names the scene it lives in.

```
public interface IWorldDocument
{
    void WriteScene(uint sceneHandle, ushort sceneId, SceneScope sceneScope);
    IWorldStateWriter BeginSystem(uint systemId, in WorldSystemIdentity worldSystemIdentit
y);
    void EndSystem();
    bool TryReadScene(out uint sceneHandle, out ushort sceneId, out SceneScope sceneScope)
;
    bool TryReadSystem(out uint systemId, out WorldSystemIdentity worldSystemIdentity, out
 IWorldStateReader worldStateReader);
}
```

Writing direction:

- `WriteScene` records one open scene instance.
- `BeginSystem(systemId, in worldSystemIdentity)` opens one object and returns the `IWorldStateWriter` its components are written through.
- `EndSystem` closes the object opened by `BeginSystem`.

Reading direction:

- `TryReadScene` returns the next saved scene, or `false` once they are exhausted.
- `TryReadSystem` returns the next saved object, its `WorldSystemIdentity`, and the `IWorldStateReader` to read its components through. The returned reader is valid until the next `TryReadSystem` call, so apply state before asking for the next object.

IWorldStateWriter and IWorldStateReader

These write and read one object's component state, keyed by component type name and member name rather than by position. Names are what let a save survive a component gaining or losing a member.

```
public interface IWorldStateWriter
{
    void BeginComponent(string componentTypeName);
    void Write(string memberName, NetworkMemberBase networkMember);
    void EndComponent();
}

public interface IWorldStateReader
{
    bool TryBeginComponent(string componentTypeName);
    bool TryRead(string memberName, NetworkMemberBase networkMember);
    void EndComponent();
}
```

Generated code calls these, not game code: each `NetworkComponent` emits a `WriteWorldState` that names its own type and then each member in declaration order. `Write` hands a member over whole rather than as a value, so an implementation asks the member to write itself and stores the resulting bytes under its name; it never grows a case per type.

On the read side, a member the save does not carry is never assigned and keeps its constructed default. A name the save carries that the component no longer has is never asked for and is skipped. There is no version number or schema check. What this does not cover is a member whose type changed under a name that stayed the same; that reads back silently wrong. Rename the member when its meaning changes.

WorldSystemIdentity

A readonly struct recording everything a save needs beyond member values: what type to build, what it was built from, where it lives, and what it belongs to.

Field	Meaning
`SystemTypeName`	Full name of the `NetworkSystem` type to build, resolved through the type registry on load. Carried by name because the wire's numeric type identity shifts whenever a networked type is added, removed or renamed, and nothing detects that shift at runtime.
`PlatformId`	The prefab or scene-object identifier the object was started with.
`PrefabBundleId`	The content bundle the prefab was registered from, or the base build for an object that ships inside it.
`IsSceneObject`	Whether the object is a scene object; a load binds it to an object the scene already placed instead of constructing one.
`SceneHandle`	The scene instance the object lives in.
`GroupId`	The group the object belongs to.
`ParentId`	The object this one is parented to.
`ComponentTypeNames`	Full names of the `NetworkComponent` types the object carried, in order. A constructed system starts with no components, so a save needs this list to know what to add before it has values to put in them. A scene object ignores it for rebuilding, since its components were placed by the scene; it is recorded only to describe what was there.

WorldLoadFailureAction

```
public enum WorldLoadFailureAction : byte
{
    Skip,
    Fail
}
```

Returned from `OnSceneUnavailable` when a saved scene cannot be opened in this build. `Skip` drops that scene instance and every object saved inside it, then loads the rest of the world; it is the default and the safer answer when scenes are independent of each other. `Fail` abandons the load and `LoadAsync` returns `false`, but it does not bring back the world that was standing: a load despawns every object and closes every open scene before it reads the saved scenes, so by the time this is asked the old world is already gone. No saved object is built until every saved scene has been answered for, so `Fail` leaves an empty world rather than a half-built one. Saved scenes that reopened before the one that failed stay open, with nothing built in them.

JsonWorldStore as a reference

`JsonWorldStore` is the shipped `IWorldStore`. It writes a real JSON structure (scenes, objects, components and member names all read as text), with each member's packed value carried as base64 under its name, so a saved world diffs object by object and member by member.

```
public sealed class JsonWorldStore : IWorldStore
{
    public string FilePath { get; }
    public WorldLoadFailureAction MissingSceneAction { get; set; } = WorldLoadFailureActio
n.Skip;

    public JsonWorldStore(CoreManager coreManager, string filePath) { ... }
}
```

`FilePath` is the file saved worlds are written to and read from. `MissingSceneAction` is settable rather than fixed, so a game can pick `Skip` or `Fail` without implementing the interface itself; it defaults to `Skip`. `OnSceneUnavailable` just returns `MissingSceneAction`.

The file is written to a temporary path beside the target and moved over it only once the write completes, so a save interrupted partway leaves the previous world in place rather than a truncated one.

Use `WorldDocument` (the `IWorldStore`-agnostic `IWorldDocument`/`IWorldStateWriter`/`IWorldStateReader` implementation `JsonWorldStore` builds through `BeginSave`) as a second reference if your own store needs an in-memory document to write into before serializing it to your target.

Reacting to changes: OnMembersChanged

Using Unity? See Reacting to a replicated change in Unity.

`NetworkComponent` exposes one callback for every member change, rather than a separate event per member:

```
public virtual void OnMembersChanged(ulong memberFlags, MemberChangeDirection memberChange
Direction) { }
```

Override it to react to state without polling.

The two arguments

`memberFlags` is one bit per member, in member-Id order. Cast it to the component's generated flags enum to test individual members:

```
public override void OnMembersChanged(ulong memberFlags, MemberChangeDirection memberChang
eDirection)
{
    TransformComponentFlags flags = (TransformComponentFlags)memberFlags;
}
```

`memberChangeDirection` is a `MemberChangeDirection`, and names which side of the wire the raise is reporting:

- `Write` - this peer changed the members itself; the callback fires before the tick's writing runs.
- `Read` - the members were applied from the wire; the callback fires after every packet of the inbound pass has landed.

A host raises both directions in the same frame, write first, without waiting for its own packet to come back.

Write: before serialization, not after transmission

`Write` reports that the members were committed to this tick's outbound pass, not that bytes left the process. A system whose only observer is culled by interest, or withheld because the change came from that observer, still raises `Write`. Do not use it to infer that anything was actually sent.

Read: after the inbound pass, before Reconcile

`Read` fires once per tick, after every packet of the inbound pass has landed and before the Reconcile step. It is the only direction under which `NetworkMemberBase.TryGetWritingClient` can resolve a writer - that claim is scoped to the tick an inbound apply landed in. A handler that assumes a raise means a remote change arrived must check for `Read`; code calling `TryGetWritingClient` only makes sense there.

Reading the flags correctly

Three cases catch people:

- **Full apply.** A spawn, resync, reconcile, or whole-component recovery reports `ulong.MaxValue`. Treat every member as changed rather than trying to diff the flags.
- **Member-granular recovery.** A recovery serve reports only the members it actually repaired, not the full mask. Whether a component supports this is fixed per type (`IsMemberRecoverySupported`); a component that doesn't falls back to the full-apply case above.
- **Delta flags name what the wire carried, not what differs.** A member's bit being set means the wire carried that member on this pass, not that its value changed from before. A predicted member whose

prediction was confirmed still sets its bit.

Inheritance aliases member flags across layers

For a component whose inheritance chain declares members on more than one layer, each layer's flags word starts at bit zero, so the union `memberFlags` reports aliases bit positions across layers - bit 0 of one layer and bit 0 of another land on the same bit. Single-layer components, the norm, map exactly onto the generated flags enum. This is also why an inheriting component's `IsMemberRecoverySupported` is `false`: with aliased bits a recovery mask can't name a single member unambiguously, so it falls back to whole-component recovery.

Why one callback, not a per-member event

A per-member `Changed` event was a declined design. One component-wide callback raises once per component per direction instead of once per changed member, and needs no per-member subscription bookkeeping - nothing to wire up or tear down as members are added. The cost is that a handler interested in one member checks its bit in `memberFlags` rather than subscribing directly to it.

Collection member reference

Common to every collection member

All five collection members derive from the same base and share this surface:

- `Values` — the live collection.
- `PreviousValues` — a snapshot from the most recent tick before the current head on which the collection actually changed.
- `Count` — the number of items currently held.
- `Changes` — an `IReadOnlyList<TChange>` of this tick's operations, appended to as mutations land or a received operation log is applied.
- A constructor taking `int ringDepth = 2`, the number of distinct snapshots retained in the ring. Two is the minimum that exposes a previous-tick view.
- Protected virtual hooks: `GetRingDepth()`, which returns the ring depth the member was constructed with, and `GetCurrentTick()`, which returns the tick from the network loop (or an unset value when none is running). Override either only to change how a subclass sources those values.

NetworkListMember\<T0>

An ordered, index-addressable collection.

Member	Signature	Notes
`Add`	`void Add(T0 item)`	Appends to the end.
`AddRange`	`void AddRange(IReadOnlyList<T0> items)`	Appends each item in order.
`Insert`	`void Insert(int index, T0 item)`	Inserts at the given index.
`Remove`	`bool Remove(T0 item)`	Removes the first occurrence; returns whether one was found.
`RemoveAt`	`void RemoveAt(int index)`	Removes the item at the given index.
`Clear`	`void Clear()`	Empties the list.
`Contains`	`bool Contains(T0 item)`	
`IndexOf`	`int IndexOf(T0 item)`	Returns `-1` when not found.
indexer	`T0 this[int index] { get; set; }`	The setter records a `Set` operation; a write of the value the element already holds is discarded before anything is recorded.
`GetEnumerator`	`List<T0>.Enumerator GetEnumerator()`	Struct enumerator over the live list.

```
NetworkListMember<int> scores = new();
scores.Add(10);
scores.AddRange(new List<int> { 20, 30 });
scores[0] = 15;
scores.RemoveAt(1);
```

`Changes` is an `IReadOnlyList<NetworkListChange<T0>>`. Each `NetworkListChange<T0>` carries `Operation` (a `NetworkListOperation`), `Index`, and `Item`. `NetworkListOperation` is `Add`, `Insert`, `RemoveAt`, `Set`, or `Clear`; `Index` is undefined for `Clear`, and `Item` is the default of `T0` for `RemoveAt` and `Clear`.

NetworkArrayMember\<T0>

A fixed-length, index-addressable collection with per-element interpolation.

Member	Signature	Notes
`Set`	`void Set(int index, T0 item)`	Replaces one element. A write of the value already held is discarded before anything is recorded.
`Fill`	`void Fill(T0 item)`	Sets every element to the same value; length is unchanged.
`Resize`	`void Resize(int newLength)`	Grows or shrinks the array. New slots from a grow are default-filled; truncated slots are dropped. Throws `ArgumentOutOfRangeException` above 65,536, the largest length a receiver honors.
`Clear`	`void Clear()`	Sets every element to the default of `T0`; length is unchanged.
`IndexOf`	`int IndexOf(T0 item)`	Returns `-1` when not found.
`Count` / `Length`	`int Count { get; }` / `int Length { get; }`	Equivalent; `Length` is provided for the array idiom.
`Values`	`ReadOnlySpan<T0> Values { get; }`	The live contents as a span. Aliases rented backing storage and is valid only until the next mutation.

`GetInterpolated`	`T0 GetInterpolated(int index)`	The one collection member with per-element interpolation: animates the value at `index` from what it held on the previous change toward its current value, for a peer rendering between ticks. Reads as the current value when the element type has no registered interpolator, or when the index is beyond what the previous snapshot held.
`GetEnumerator`	`Enumerator GetEnumerator()`	Struct enumerator over the live contents, truncated to the logical length.

```
NetworkArrayMember<float> healthByRegion = new(ringDepth: 2);
healthByRegion.Resize(4);
healthByRegion.Set(0, 100f);
float rendered = healthByRegion.GetInterpolated(0);
```

`Changes` is an `IReadOnlyList<NetworkArrayChange<T0>>`. Each `NetworkArrayChange<T0>` carries `Operation` (a `NetworkArrayOperation`), `Index`, `Item`, and `NewLength`. `NetworkArrayOperation` is `Set`, `Fill`, `Resize`, or `Clear`. `Index` is undefined for `Fill`, `Resize`, and `Clear`; `NewLength` is undefined for every operation except `Resize`; `Item` is the default of `T0` for `Resize` and `Clear`.

NetworkDictionaryMember\<T0, T1>

A keyed collection with key type `T0` and value type `T1`.

Member	Signature	Notes
`Add`	`void Add(T0 key, T1 value)`	Throws when the key is already present.
`Remove`	`bool Remove(T0 key)`	Returns whether a matching entry was found and removed.
`Clear`	`void Clear()`	Empties the dictionary.
`ContainsKey`	`bool ContainsKey(T0 key)`	
`TryGetValue`	`bool TryGetValue(T0 key, out T1 value)`	
indexer	`T1 this[T0 key] { get; set; }`	The setter adds a new key or overwrites an existing one. A write of the value the key already holds is discarded before anything is recorded.

```
NetworkDictionaryMember<int, string> playerNames = new();
playerNames.Add(1, "Alice");
playerNames[1] = "Alicia";
playerNames.TryGetValue(1, out string name);
```

`Changes` is an `IReadOnlyList<NetworkDictionaryChange<T0, T1>>`. Each `NetworkDictionaryChange<T0, T1>` carries `Operation` (a `NetworkDictionaryOperation`), `Key`, and `Value`. `NetworkDictionaryOperation` is `Set`, `Remove`, or `Clear` — an add and an overwrite both encode as `Set` since the wire payload is identical. `Key` is the default of `T0` for `Clear`; `Value` is the default of `T1` for `Remove` and `Clear`.

NetworkHashSetMember\<T0>

An unordered collection of unique values.

Member	Signature	Notes
`Add`	`bool Add(T0 item)`	Returns whether the item was added (false when already present).
`Remove`	`bool Remove(T0 item)`	Returns whether the item was found and removed.
`Clear`	`void Clear()`	Empties the set.
`Contains`	`bool Contains(T0 item)`	

```
NetworkHashSetMember<int> activeQuestIds = new();
activeQuestIds.Add(7);
bool hasQuest = activeQuestIds.Contains(7);
activeQuestIds.Remove(7);
```

`Changes` is an `IReadOnlyList<NetworkHashSetChange<T0>>`. Each `NetworkHashSetChange<T0>` carries `Operation` (a `NetworkHashSetOperation`) and `Item`. `NetworkHashSetOperation` is `Add`, `Remove`, or `Clear`; `Item` is the default of `T0` for `Clear`.

NetworkQueueMember\<T0>

A first-in-first-out collection.

Member	Signature	Notes
`Enqueue`	`void Enqueue(T0 item)`	Appends to the back.
`Dequeue`	`T0 Dequeue()`	Removes and returns the front item.
`TryDequeue`	`bool TryDequeue(out T0 item)`	Returns `false` with the default of `T0` when the queue is empty.
`Peek`	`T0 Peek()`	Returns the front item without removing it.
`TryPeek`	`bool TryPeek(out T0 item)`	Returns `false` with the default of `T0` when the queue is empty.
`Contains`	`bool Contains(T0 item)`	

`Clear`	`void Clear()`	Empties the queue.

```
NetworkQueueMember<string> pendingActions = new();
pendingActions.Enqueue("Attack");
if (pendingActions.TryDequeue(out string action))
{
    // action == "Attack"
}
```

`Changes` is an `IReadOnlyList<NetworkQueueChange<T0>>`. Each `NetworkQueueChange<T0>` carries `Operation` (a `NetworkQueueOperation`) and `Item`. `NetworkQueueOperation` is `Enqueue`, `Dequeue`, or `Clear`. For `Enqueue` the item is the one appended; for `Dequeue` it is the one removed from the front; `Item` is the default of `T0` for `Clear`.

Sending your own data beside the members

Some things a component needs to tell a peer are not values a member can hold.

Take a cast bar. The spell being cast is a value, so it is a member. How far into the cast the caster already is, is not: it moves every tick, so holding it as a member would pay for it on every tick of every cast. What a peer actually needs is that number once, at the moment it is first told about the object at all.

A component writes that itself, by overriding a pair on `NetworkComponent`:

```
public partial class CastComponent : NetworkComponent
{
    public readonly NetworkMember<ushort> SpellIndex = new();

    private uint _castStartTick;
    private uint _elapsedCastTicks;

    protected override void WritePayload(Writer writer, PayloadReason reason, Invoker auth
or, ulong serializedMemberFlags)
    {
        // This serialization never carried the spell, so there is no cast here to describ
e.
        if ((serializedMemberFlags & (ulong)CastComponentFlags.SpellIndex) == 0)
            return;

        writer.WriteUInt32(CoreManager.NetworkLoopManager.Tick - _castStartTick);
    }

    protected override void ReadPayload(Reader reader, PayloadReason reason, Invoker autho
r, ulong serializedMemberFlags)
    {
        if ((serializedMemberFlags & (ulong)CastComponentFlags.SpellIndex) == 0)
            return;

        // SpellIndex.Value already holds what this serialization carried, so the bar can
be built from both.
        _elapsedCastTicks = reader.ReadUInt32();
    }
}
```

A client that spawns halfway through a cast now has both halves: the spell from the member, and how far in it is from the payload. So does a client that lost the tick the cast started on, because the repair the engine serves for that tick carries the payload too.

When the pair runs

It rides every serialization that carries whole values, and no ordinary delta:

- The spawn that builds the object on a peer.
- A resync, and an interest serve when an object comes back into view.
- A reconcile to a controlling client.
- A client's own write of a whole object upstream.
- The recovery serve that repairs a tick a peer lost.

An ordinary delta writes members and never the component, so a member that moves every tick costs no

payload however long it moves for. A component that overrides neither method costs nothing at all, not one bit.

It runs after the members, not before

This is the part to build on. What a payload says usually depends on what the members now hold, and a peer cannot know how to read one until it has read them. So the pair is a trailer: by the time `ReadPayload` runs, every member of that component has already landed and can be read straight off it. The example above relies on exactly that, reading `SpellIndex.Value` while it parses the elapsed time that belongs beside it.

The two arguments

`serializedMemberFlags` says which members this serialization carried, one bit per member Id, matching the `<Component>Flags` enum the generator already writes for you.

It is `NetworkComponent.EverySerializedMember` whenever the whole component is being carried, which covers a spawn, a resync, an interest serve and a reconcile. It is narrower for a recovery serve, and that is the case the flags exist for. A recovery is not a re-send of everything: it repairs only the members the lost tick actually carried. A payload describing one particular member has no business riding a repair that member was not part of, so test its bit and return when it is clear.

`reason` says why the component is being stated rather than diffed, and is one of three:

Reason	When
`Serve`	The component is stated whole so the receiver can build the object, or rebuild it from nothing it can trust. From the authority that is a spawn, a resync, or an interest serve; from a client it is a predicted spawn.
`Reconcile`	The authority is correcting the client that controls the object.
`Recovery`	The authority is repairing a tick a peer lost. This is the only reason whose member flags are narrower than the whole component.

`author` is an `Invoker` saying which side wrote the body: on the write side that is this peer, on the read side it is the sender. Direction is kept apart from the reason because the two are independent: every reason means the same act whichever side wrote it, so a `Serve` from a client is a predicted spawn and a `Serve` from the authority is a spawn or a resync.

Nothing on the wire carries either of them. The writer knows what it is doing, and the reader derives the same answers from the subpacket kind the frame already named, from whether the Connection the body arrived on is the server's, and from a discriminator every upstream full already leads with. So the two ends agree by construction and it costs no bits.

`Reconcile` is the one worth naming when deciding what to write: a controlled object reconciles to its controller constantly, so a payload written there is paid for on every tick of prediction, where the same payload on a `Serve` is paid for once. Return early for the reasons you have nothing to say for.

The one rule

Nothing frames the payload, so **every bit written must be read back**. Write four bytes and read two, and

everything behind the payload decodes against the wrong offset. That surfaces as values arriving wrong or a whole tick being discarded, usually somewhere far away from the component that caused it.

Both ends are handed the same flags and the same reason precisely so the decision to write and the decision to read are made from the same facts rather than each end guessing. Gate on those two arguments and on the member values. Never gate on local state the other end cannot see, such as a field only the server sets or something read off your own scene.

A read also happens on a body the engine is about to throw away, so applying a payload must be harmless rather than guarded against.

When a payload throws

An exception thrown inside either half is caught and logged, and the body it was part of is dropped whole rather than sent half written, because nothing frames the payload and a half-written one cannot be read past.

- A throw while writing means that body is not sent. The peers it was for are served again on a later pass, so a payload that keeps throwing keeps its object from ever reaching them.
- A throw while reading is handled like a body that could not be read. A later repair replaces what the send still owed, and a predicted spawn whose payload throws is refused.

Prefer a member where the data is a value

The pair has no change detection of its own and is not meant to. It rides the serves your members have already earned, so a component whose members never change is never served and never writes a payload.

Where the thing genuinely is a value, a member is still the better answer, and it is cheaper than it looks. A cast's *start tick* as a member changes once per cast and costs nothing on the ticks between, and every peer can subtract it from the current tick itself. Reach for a payload when the thing is not a value: a snapshot you compute rather than store, a variable-length list, or anything whose meaning is "here is where this object stands right now" rather than "here is what just changed".

The other payload

A spawn payload is the other half of this. That one is declared per object rather than per component, rides the spawn alone and never a repair, and is always read after the member states. Use it for what is true about an object at the moment it spawns. Use the pair here for what a component's own state needs to carry alongside it.

Replicated State and Change Callbacks

Pages in this category:

- NetworkMember
- How replicated state works
- Networked collections
- Reacting to changes: OnMembersChanged
- Your first networked component
- Collection member reference
- TransformComponent
- What a member can hold, and how to keep one off the wire
- How Often a Member Is Sent
- Letting clients write state
- Retention, redundancy and recovery
- Judging and arbitrating client writes

What a member can hold, and how to keep one off the wire

What T0 can be

A NetworkMember<T0> field must satisfy one of three shapes:

- A value type.
- string.
- A class implementing IPoolResettable, used together with automatic pooling.

A class-typed member must be assigned before its first full serialization. An unset member holds null, and the generated writers cannot encode that.

```
private readonly NetworkMember<float> _health = new();
private readonly NetworkMember<MyPoolableClass> _payload = new(isAutomaticPoolingEnabled:
true);
```

Automatic pooling

NetworkMember<T0> keeps a small ring of past values so it can expose PreviousValue alongside the live Value. When a class-typed, IPoolResettable value is dropped from the ring, isAutomaticPoolingEnabled: true (a constructor argument on NetworkMember<T0>) routes it back to its pool through NetworkTypePoolReturner<T0>, whose Return method invokes a pool-return delegate registered for that closed T0. This closes the loop with the generated deserializers, which rent incoming instances from ResettableObjectPool<T0> rather than allocating them.

NetworkTypePoolReturner<T0>.IsRegistered reports whether a returner exists for the type. If automatic pooling is enabled but the type either does not implement IPoolResettable or has no returner registered, that is surfaced as an error once at spawn rather than on every drop.

Automatic pooling v1 targets flat value objects. A nested class-typed field inside a pooled T0 can alias across instances, because the delta reader copies an unchanged field by reference rather than by value.

Delta support is scoped, not gapped

The generated writers and readers have delta serializers for numerics, Quaternion, List<T0> and Dictionary<T0,T1>, Nullable<T0>, and generated composite types. Types outside that set are not silently mis-encoded: several delta paths (Matrix4x4, plain arrays, NetworkSystemGroup, Channel, NetworkSystem) throw NotImplementedException instead. That is a deliberate boundary on which types are considered hot enough to delta-encode, not an oversight to work around. Register a serializer for a type that has none rather than routing it through NetworkMember<T0> as-is — see the Serialization section.

[ReplicationIgnoreAttribute]

[ReplicationIgnore] keeps a member off the wire while leaving how it is saved alone. It targets a NetworkMemberBase field or a whole NetworkComponent; on a field it excludes that member, on the component it excludes every member the component declares.

```
[ReplicationIgnore]
private readonly NetworkListMember<ItemStack> _containerContents = new();

[ReplicationIgnore]
private readonly NetworkMember<bool> _isModerationFlagged = new();
```

It is an attribute rather than a constructor argument because the exclusion is decided while the source generator emits code: a marked member's serialize and deserialize lines are never generated at all, so there is no per-peer branch testing a runtime flag. This is why a container's contents or a moderation flag can survive a restart without being handed to every observer: persistence and replication are separate decisions, and this attribute only touches the second one.

The three consequences

- **The member never enters the per-tick change examination.** It pays for no delta projection, no tolerance comparison, no send interval, and no transmission mode. A tick on which only ignored members changed produces no packet. `NetworkComponent.OnMembersChanged` still fires under `MemberChangeDirection.Write` on the peer that made the change, so server code reacts to it normally.
- **The value is the constructed default on every peer but the server.** Nothing ever arrives on the wire to populate it, so reading it on a client returns the type's default, not what the server holds.
- **A peer promoted to server by adoption holds those defaults too**, because it never received the real values. Saving from a newly adopted server writes those defaults over whatever the store already held.

See also

- Serialization — registering a delta or full serializer for a type that has none.
- Persistence — `[WorldStateIgnore]`, the inverse: keeps a member out of a saved world while leaving replication alone.

NetworkMember

NetworkMember is the scalar building block of a component

`NetworkMember<T0>` is the scalar building block a generated `NetworkComponent` declares its replicated fields with. It stores a sparse ring of past values for `T0`, exposes the current, previous, and interpolated views of that value, and owns the construction-time knobs that decide how it is compressed, paced, and projected on the wire. The non-generic behaviour (write access, `IsReplicated`) lives on its base type, `NetworkMemberBase`.

Reading the value

Member	Description
`Value`	Gets or sets the current value. A set mutates the live ring head and notifies the framework. Equality against the prior committed value is checked at serialize time inside the change examination, not on every set, so a noop assignment costs nothing beyond the write itself.
`PreviousValue`	The value held in the ring slot immediately before the live one. Returns `default(T0)` if no prior value has been committed.
`InterpolatedValue`	A smoothed value that animates from `PreviousValue` to `Value` across the member's own send interval, then holds at `Value`. See Interpolation below.

Reading the ring directly

`Value` and `PreviousValue` cover the common case. For a caller that needs a specific tick, or needs to derive a rate of change, the ring is also queryable directly:

```
public bool TryGetValueByTick(uint tick, out T0 value);
```

Returns true and outputs the value when a ring slot tagged with the supplied tick still holds it. A member is written only on the ticks it changes, so a tick between two writes has no slot of its own, and an old enough tick is no longer held. An unpredicted member keeps two slots; a predicted one keeps the prediction history (see Predicted members and prediction history).

```
public bool TryGetValueAtOrBeforeTick(uint tick, out T0 value);
```

Returns the value the member held as of the supplied tick: the newest slot at or before it. This answers for the ticks between two writes, and is false only when every slot the ring holds is newer.

```
public bool TryGetPreviousValueAndTickGap(out T0 previousValue, out uint tickGap);
```

Returns the value committed to the slot before the live one, together with the tick span between the two slots. Intended for an owner that derives a rate of change (a velocity from position deltas, for example) instead of replicating it directly. Returns false, with `tickGap` set to `0`, unless both slots hold a written value and the live slot is strictly newer.

Construction

```
public NetworkMember(
    CompressionLevel deltaCompressionLevel = CompressionLevel.Tight,
    bool isAutomaticPoolingEnabled = false,
    MemberRole role = MemberRole.State,
    float accuracy = float.NaN,
    float interpolationSnapThreshold = float.NaN,
    SendInterval sendInterval = SendInterval.Normal,
    TransmissionMode transmissionMode = TransmissionMode.Interval,
    bool isExtrapolationEnabled = true,
    byte correctionSmoothingTicks = 0,
    bool isCarriedForward = true);
```

All ten arguments are supplied by name at the field declaration, e.g. `NetworkMember<T> _x = new(role: MemberRole.Validate);`.

Argument	Construction-time only?	Meaning
`deltaCompressionLevel`	Yes	The compression level used when delta-encoding this member's value. Defaults to `CompressionLevel.Tight`.
`isAutomaticPoolingEnabled`	Yes	Returns dropped `IPoolResettable` ring occupants to their pool through `NetworkTypePoolReturner<T0>`. When true, the member owns instances assigned to `Value`: rent them from `ResettableObjectPool<T0>` and never retain or mutate an instance after assigning it.

`role`	Yes	What the member is for. `MemberRole.State` (the default) is ordinary state: inbound values apply as they arrive. `MemberRole.Predict` makes it client state, a per-tick value the simulation reads (a move direction, a jump press), written by whoever controls the system and sent upstream by a controlling client every tick; see [Client state](../control/client-state-components.md). `MemberRole.Validate` makes it a predicted result: the controlling client computes it ahead of the server and never sends it upstream, the server simulates the same tick and orders a reconcile when its result differs, and its value is quantized to its accuracy on every write. On a client that does not control the system, any role other than `State` is held at the server's values while a reconcile replays. See [Predicted members and prediction history](../control/prediction-history-and-predicte
`accuracy`	Construction default only	The wire accuracy for delta encoding, loss recovery, and lossy serialization. `float.NaN` (the default) defers to the per-type default accuracy. A per-call accuracy passed to `Initialize` still overrides it, and coarser values trade precision for bandwidth on quantities that tolerate more error than position.
`interpolationSnapThreshold`	Yes	The distance beyond which `InterpolatedValue` snaps to `Value` instead of animating from `PreviousValue`. `float.NaN` (the default) disables snapping. Usable only on numeric types with a registered tolerance comparer.

`sendInterval`	No, see `SetSendInterval`	The span between deltas while `TransmissionMode.Interval` is in force. `SendInterval.Normal` sends every tick.
`transmissionMode`	No, see `SetTransmissionMode`	The transmission strategy: `TransmissionMode.Interval` (the default) encodes ordinary deltas paced to `sendInterval`; `TransmissionMode.Divine` (Pro) projects motion instead, for values that tend to move predictably.
`isExtrapolationEnabled`	No, see `SetExtrapolationEnabled`	Whether receivers keep the value moving through the sender's silence, under either mode. `true` (the default) favors staying live when an update is lost; `false` holds still until told and favors staying accurate. Under `TransmissionMode.Interval`, `true` costs a tiny overhead on each update plus one extra update each time the value stops; `false` costs nothing extra. See [How Often a Member Is Sent](./send-pacing.md).
`correctionSmoothingTicks`	Yes	How many ticks `InterpolatedValue` takes to glide from the value shown before a reconcile to the corrected one, on a client that predicts the member, while `Value` is already correct. `0` (the default) shows the correction at once.
`isCarriedForward`	Yes	For a `MemberRole.Predict` member only: `false` keeps it at its default when `NetworkComponent.TryGetLastState(false)` carries the previous tick's client state across a tick whose client state did not arrive, for a value that must not repeat, such as a jump press. Ignored for every other role.

Interpolation

`InterpolatedValue` animates across a window sized to the member's own send interval: a member sent every tick sweeps across that single tick, and a member paced at N ticks sweeps each arrival across N. Both peers construct the member with the same interval as part of the shared component declaration, so the receiver knows the window without the sender naming it on the wire.

`interpolationSnapThreshold`, fixed at construction, is what cuts a teleport instead of sweeping it: when `PreviousValue` and `Value` are found apart by more than the threshold, `InterpolatedValue` returns `Value` directly rather than animating across the discontinuity.

Runtime overrides

The three construction-time doors above with a runtime override can be re-authored after construction, typically from a platform integration pushing a serialized value on bind:

```
public void SetSendInterval(SendInterval sendInterval);
public void SetTransmissionMode(TransmissionMode transmissionMode);
public void SetExtrapolationEnabled(bool isExtrapolationEnabled);
```

`SetReplicates(bool isReplicated)`, on `NetworkMemberBase`, sets whether the member is sent to other peers at all. It is called only by generated code, emitted beside a member's `Initialize` call and only for a member declared with `ReplicationIgnoreAttribute`; calling it by hand desynchronizes what this peer sends from what its peers expect to read.

Attributing a write

```
public bool TryGetWritingClient(out Connections.Connection connection);
```

Resolves the client that wrote this member on the current tick, defined on `NetworkMemberBase`. It is narrow by design: it is meaningful only from inside a read raise, called from `NetworkComponent.OnMembersChanged` under `MemberChangeDirection.Read`, while the tick that applied the write is still current. Called under `MemberChangeDirection.Write` it reports nothing, because that raise describes what this peer is about to send, never a client's inbound write. Asked on a later tick, or after the writing client has disconnected, it also returns false.

See also

For how `accuracy` trades precision against bandwidth, how `sendInterval` and `TransmissionMode` interact with pacing more generally, and the write-access rules `TryGetWritingClient` sits inside of, see the accuracy, pacing, and write-access reference pages.

Networked collections

The family

Nucleus replicates five collection shapes, each a `NetworkComponent` member type built on the same base:

- `NetworkListMember<T0>` — an ordered, resizable list
- `NetworkArrayMember<T0>` — a fixed-capacity array with an explicit `Length`/`Resize`
- `NetworkDictionaryMember<T0, T1>` — a key/value map
- `NetworkHashSetMember<T0>` — an unordered set of unique elements
- `NetworkQueueMember<T0>` — a FIFO queue

All five derive from `NetworkCollectionMemberBase<TChange>`, which owns the change-log and snapshot-ring bookkeeping shared across the family; each family only supplies its own storage, mutation surface, and wire shape. As with scalar members, keep the element type (and, for the dictionary, the key type) to a value type or `string`. Codegen does not enforce this, but an element held by reference can be changed without going through the public mutation API, and the collection never sees that change.

```
public class NetworkListMember<T0> : NetworkCollectionMemberBase<NetworkListChange<T0>>
public class NetworkArrayMember<T0> : NetworkCollectionMemberBase<NetworkArrayChange<T0>>
public class NetworkDictionaryMember<T0, T1> : NetworkCollectionMemberBase<NetworkDictiona
ryChange<T0, T1>>
public class NetworkHashSetMember<T0> : NetworkCollectionMemberBase<NetworkHashSetChange<T
0>>
public class NetworkQueueMember<T0> : NetworkCollectionMemberBase<NetworkQueueChange<T0>>
```

Mutating a collection

Each member exposes ordinary collection APIs on the surface — `Add`, `Insert`, `RemoveAt`, `Clear`, the indexer for a list, array or dictionary, `Enqueue`/`Dequeue` for a queue, `Add`/`Remove` for a hash set. Underneath, every mutation appends a coalesced entry to a per-tick change log rather than marking the whole member dirty:

```
public void Add(T0 item)
{
    EnsureHeadForCurrentTick();

    List<T0> values = _ringValues[HeadIndex];
    int index = values.Count;
    values.Add(item);

    AppendChange(NetworkListOperation.Add, index, item);
    NotifyChanged();
}
```

A repeated `Set`/indexer write to the same slot within a tick overwrites the pending log entry in place instead of appending a second one — a list's `AppendSetOperation` folds a `Set`, `Add` or `Insert` at the same index together rather than logging both.

What travels on the wire

The log is what the delta carries. `WriteDelta` writes the tick's coalesced operations, not a snapshot of the

collection:

```
public override void WriteDelta(Writer writer, DeltaCheckMode deltaCheckMode)
{
    if (writer.IsEveryDeltaMemberAbsolute)
    {
        Write(writer);

        return;
    }

    WriteOperations(writer);
}
```

On the receiving side, `ReadDelta` applies those operations in order, appending each one to the same `Changes` log the local mutator would have populated. An operation the local contents can no longer take — an index a lost delta would have made room for — is skipped rather than thrown on, but its bits are still consumed so the rest of the tick stays aligned; recovery repairs the divergence with a full, absolute serve.

A full `Write`/`Read` always carries the whole collection, states what it *is* rather than what moved, and leaves the change log empty.

The change gate, not the encoding

The wire format is the operation log, but whether anything is sent at all is decided separately, by a gate that runs before `WriteDelta` is ever reached. `TryFlagAsChanged` only flags a member once its log is non-empty *and* its contents differ observably from the previous ring snapshot:

```
internal override bool TryFlagAsChanged(uint tick)
{
    if (ChangeLog.Count == 0)
        return false;

    if (!HasObservableChange())
    {
        ChangeLog.Clear();

        return false;
    }

    NetworkMemberBase networkMember = this;
    NetworkComponent.OnMemberChanged(networkMember);

    return true;
}
```

This is why a mutator that ran and then refilled a collection back to its previous contents costs nothing on the wire: the log is non-empty, but `HasObservableChange` compares the live contents against the previous ring snapshot and finds no difference, so the log is cleared and nothing is flagged. A non-empty log says a mutator ran, not that the collection now holds anything different — polling external state into a collection every tick would otherwise spend framing on every tick whether or not the value actually moved. This mirrors the scalar `NetworkMember<T0>` comparison, including tolerance: elements compare through `NetworkTypeToleranceComparer<T0>`, so a floating element resting within the member's own accuracy compares equal exactly as a scalar does.

The upstream-write exception

Operations are an encoding against what the receiver already holds, so they only work where the sender knows what that receiver's previous state was. A client's upstream write breaks that assumption — nothing downstream serves recovery for an upstream delta — so a client writing a collection member, and the server relaying that write for the tick, both carry the whole collection as an absolute rather than as operations. The encoding rides as the delta's subpacket kind rather than a bit in the body, so the receiver always knows which shape to expect before decoding.

Reading the previous state

Every member exposes:

- `Values` — the live contents (`IReadOnlyList<T0>` for a list, `ReadOnlySpan<T0>` for an array, `IReadOnlyDictionary<T0, T1>` for a dictionary, `IReadOnlyCollection<T0>` for a hash set or queue)
- `PreviousValues` — the contents as of the most recent tick the collection actually changed, prior to the current head
- `Count` — the current element count (the array also exposes `Length`, identical to `Count`)

Every constructor takes a `ringDepth` argument, defaulting to 2:

```
public NetworkListMember(int ringDepth = 2) : base(ringDepth)
```

Two is the minimum that exposes a previous-tick view at all — a depth below it is raised back to two. A deeper ring covers more ticks of history, at the cost of holding that many copies of the collection; the ring is sparse, consuming a slot only on ticks where the collection actually mutated, so a shallow ring can still cover many quiescent ticks between changes.

Reacting to a change

`NetworkComponent.OnMembersChanged` names which members changed on a tick; a collection member's own `Changes` property says what happened inside it for that tick:

```
public virtual void OnMembersChanged(ulong memberFlags, MemberChangeDirection memberChange
Direction) { }

public IReadOnlyList<TChange> Changes => ChangeLog;
```

`Changes` holds the current tick's operations on both sides — the ones this peer applied locally, or the ones a delta just landed — and is cleared at the end of the tick. Because of that, code that needs to inspect it must do so synchronously inside `OnMembersChanged`: the write-direction raise runs before the tick serializes anything, and the read-direction raise runs after the inbound apply has landed. A full `Read` leaves `Changes` empty, since it states what the collection is rather than what moved.

Collections cannot be multi-writer

A collection member closes off `StateWriteAccess.AnyClient` by construction, not just by convention. `NetworkSystem.TrySetWriteAccess` refuses to widen a system's access beyond `StateWriteAccess.Controller` when that system holds any collection member:

```
if (stateWriteAccess is not StateWriteAccess.Controller && HasCollectionMember())
{
    Logger<NetworkSystem>.LogError($"NetworkSystem [{AsStringInternal()}] holds a collecti
on NetworkMember, so its write access cannot be widened beyond [{nameof(StateWriteAccess.C
ontroller)}].");

    return false;
}
```

The collection families take wire-supplied counts and read through their own paths, which never reach the scalar guard stack that keeps `AnyClient` safe for a plain `NetworkMember<T0>`. Rather than guard every one of those read paths individually, multi-writer access is refused for the whole system up front.

Retention, redundancy and recovery

The interpolation ceiling

`SystemManager.StateInterpolation` sets how many ticks the receiver buffers a state packet before applying it. `0` applies immediately; `1` holds a packet until the next tick before applying; and so on. It is capped at `SystemManager.MaximumStateInterpolation` (`7`) — a higher value is silently clamped at runtime.

This value is the ceiling for most of what is on this page. The server's redundant resends and recovery only work within the window the receiver is already buffering, so both `GlobalRedundancy` and the interpolation itself are bounded by it. A client's upstream resends are the one exception, described below.

Redundancy

`SystemManager.GlobalRedundancy` is how many past serializations of each unreliable state packet every peer resends alongside the current one. A value of `N` retransmits each tick's unreliable packets on the following `N` ticks. A value of 2 puts 3 copies of every tick on the wire (the current send plus 2 redundant resends), so the receiver can recover a tick whose earlier copies were lost.

Each redundant copy arrives one tick later than the last, so on the server's sends a value larger than `StateInterpolation` would resend copies that always arrive too late to apply. The engine clamps for this automatically.

Redundancy only matters on an unreliable channel, since a reliable one already guarantees delivery and resending is wasted bandwidth. Check `SystemManager.CanUseRedundancy()` rather than reading `GlobalRedundancy` directly: it reports true only when the server's effective redundancy is above zero *and* the transport's default channel is unreliable.

Client redundancy

`SystemManager.ClientRedundancy` (default 3) is how many extra copies a client adds, on top of `GlobalRedundancy`, to everything it sends the server: its predicted spawns and its own writes. A client with the defaults therefore sends each upstream tick four times.

It is not capped at `StateInterpolation`. A copy that arrives after the server has moved past its tick is discarded as stale and costs only its upstream bytes, while one that arrives during a burst of loss, when the ticks in between were lost too, is applied. A host's own client does not use it, since what it sends never leaves the process.

Redundancy is the first defence, not the guarantee. A predicted spawn whose copies are all lost is despawned locally, and a client's write that still goes missing is repaired by upstream loss recovery.

```
if (systemManager.CanUseRedundancy())
{
    // Safe to plan around redundant resends on this connection.
}
```

A tick that loses some of its packets applies what arrived

A tick carrying more state than one packet holds is sent as several packets. On a lossy link one of them is sometimes still missing when the tick's buffering window runs out.

With `SystemManager.PartialStateApplyEnabled` on, which is the default, the receiver applies the packets that did arrive and tells the server which ones never did. The server remembers what each packet it

sent held, so it re-serves only what the missing ones carried, on its very next tick. A tick that is lost entirely is reported the same way, and the server re-serves everything that changed since the last tick the client applied. In both cases the repair starts as soon as the client notices the loss, rather than once the server has waited long enough to suspect it. The notice the receiver sends takes the same unreliable channel the state did, because a peer already losing packets is the last one that should be made to wait on a retransmit, and it is repeated for a few ticks in case a copy is lost. If every copy is lost, the server's own recovery still catches the gap, a little later than a repair would have. With it off, the whole tick is discarded and recovered. That costs more under steady loss, because a repair spanning many packets has to arrive complete before any of it applies, and a large world can fall further behind with every attempt.

A tick is still discarded whole when a missing packet carried a change the rest of the tick relies on, such as a despawn, a controller change or a reparent. Applying around such a change could put state on the wrong object.

A client's own writes are recovered too

What a client writes to a system it controls, or one it has been allowed to write, always reaches the server eventually, however much of the upstream link is lost. Messages and remote calls are not covered: those are allowed to drop.

The server reports a gap in a client's stream exactly as a client reports one in the server's: as soon as a later tick reveals it, or when a tick arrives in part. The client answers by resending the current value of exactly the members those ticks carried, and nothing else. A tick lost with nothing sent after it is invisible to the server, so a client also resends whatever the server has not acknowledged once a patience window has passed, and again each window until it is acknowledged. A repair that is itself lost is a gap like any other and is repaired the same way.

A member the server has written since the client's lost write is skipped: the client would otherwise reassert a value older than the one the server now holds. A client that may no longer write the system resends nothing for it, and nothing older than the retention window can be resent, so a link silent for longer than that loses what it was carrying.

None of this costs anything on a link that loses nothing. `ClientRedundancy` keeps the repair rare on one that does.

The setting is the receiver's alone. The sender writes the same packets either way, so the two peers do not need to match.

```
// Discard and recover any tick that loses a packet, as every tick was handled before this
 option existed.
coreManager.SystemManager.PartialStateApplyEnabled = false;
```

Retention

`SystemManager.StateRetentionMilliseconds` governs two things at once: how much past state is kept for targeted recovery, and how long a sent state tick may go unacknowledged before recovery escalates. Raise it for high-latency audiences — every recovery threshold derives from it, so a peer whose acknowledgments simply travel slowly is never mistaken for one that lost data.

- `SystemManager.DefaultStateRetentionMilliseconds` is `1000`.
- `SystemManager.MaximumStateRetentionMilliseconds` is `2000` — higher values are silently capped.
- The history ring is allocated at the maximum up front, so changing `StateRetentionMilliseconds` at runtime never triggers a reallocation.

A peer whose acknowledged tick has aged past the full retention window can no longer be recovered per-system, and is instead re-served the world it observes.

Setting them, and setting them together

Set `StateInterpolation`, `GlobalRedundancy`, `ClientRedundancy` and `StateRetentionMilliseconds` the same on both peers. Interpolation determines how large a buffer the receiver actually has, so a mismatch means one side is budgeting for recovery capacity the other side isn't providing.

A deeper `StateInterpolation` (and the `GlobalRedundancy` it permits) buys more loss tolerance — more room for a redundant resend to land, more ticks a late packet can still be applied within. It costs latency directly: every buffered tick is a tick of delay before state is applied. A longer `StateRetentionMilliseconds` buys a wider recovery window on struggling connections, at the cost of a larger history ring (fixed at the maximum) and a longer tail before a slow-but-healthy peer would be mistaken for a lossy one.

Member-granular repair

When a component needs a targeted recovery serve, the engine checks `NetworkComponent.IsMemberRecoverySupported`. A component that overrides this to `true` also overrides:

```
public virtual void WriteRecovery(Writer writer, ulong memberMask) { }
public virtual void ReadRecovery(Reader reader) { }
```

`WriteRecovery` serializes only the flagged members named by `memberMask`; `ReadRecovery` reads that mask back and applies just those members. Whether a component type supports member recovery is the component type's own choice, and it must be identical on both peers — `IsMemberRecoverySupported` isn't negotiated over the wire.

A component that does not support member recovery (the base `IsMemberRecoverySupported` is `false`) is re-served whole through its ordinary `Write` instead of a targeted recovery.

Diagnostic reads

Four members on `SystemManager` let you inspect what's actually in the serialization history, independent of any recovery in progress:

Member	Reports
`SerializationHistoryTickCount`	How many ticks of serialization history are currently retained.
`IsSystemRecentlySerialized(uint systemId)`	Whether a given system was locally serialized as changed or spawned within the recent history window.
`GetRecentlySerializedSystemIds(HashSet<uint> collectedSystemIds)`	Adds the Id of every system locally serialized as changed or spawned within the recent history window.
`TryGetSerializedSystemIdsAfterTick(uint afterTick, HashSet<uint> collectedSystemIds)`	Adds the Id of every system serialized after a baseline tick; returns false if part of the requested range has already aged out of the history window.

```
using HashSet<uint> changedSystemIds = HashSetPool<uint>.Get();

systemManager.GetRecentlySerializedSystemIds(changedSystemIds);
```

When it isn't working

See the Diagnostics page for what a `RetentionExceededViolation` or a `RecoveryUnconfirmedViolation` means when one is raised — they're the signal that retention or recovery settings are undersized for what a connection is actually experiencing.

In the Unity inspector, `UnitySystemManager` exposes `GlobalRedundancy` (labelled Global Redundancy), `ClientRedundancy` (labelled Client Redundancy), `StateInterpolation` and `PartialStateApplyEnabled` (labelled Partial State Apply) as fields. `StateRetentionMilliseconds` has no inspector field; set it in code.

How Often a Member Is Sent

Interval or Divine, not both

A `NetworkMember<T0>` picks a `TransmissionMode`: `Interval` or `Divine`. Projection and pacing are alternative rate strategies for the same job, deciding how often a value is worth putting on the wire, so a member is either projected or interval-paced, never both. That is why `TransmissionMode` is one enum rather than two independent switches: there is no state where both a send interval and a projected path apply at once.

`Interval` encodes ordinary deltas paced to the member's `SendInterval`. `Divine` is a Pro-only alternative for values whose motion tends to be predictable (a moving transform, for example): it can go quiet for stretches where a plain interval would still be sending deltas, correcting only when needed. A member built with `Divine` ignores its send interval entirely; the field is meaningless once the mode isn't `Interval`. The mechanism itself isn't something you need to reason about to use it — pick `Divine` for a value that tends to move smoothly, `Interval` otherwise.

The three spans

`SendInterval` has exactly three values, named by the span they cover rather than by a tick count, so the meaning doesn't move when the tick rate changes:

- `Normal` — a delta on every tick the value changed. The default, and the only interval a free build actually paces to.
- `Short` — a delta at most every 250ms.
- `Long` — a delta at most once a second.

`SendIntervalTicks.Resolve(SendInterval, uint)` turns a span into a tick count for a given tick rate. It rounds up, never down, so a rate that doesn't divide the span evenly costs at most one tick of extra latency rather than quietly pacing faster than asked. `Normal` always resolves to a single tick.

Two paced tiers, not a ladder. A third tier between `Short` and `Long` would sit close enough to both to be chosen by guesswork, and every extra tier costs a population of members whose sends have to be kept off the other tiers' ticks. `Short` and `Long` are far enough apart that the real choice is between them: does this value need to look live, or only to be right before long.

Pacing never loses a value. A member that changes several times inside its interval rides its latest value on the next due tick, and a member that stops changing still rides its final delta on its next due tick. Full serializations — spawn, resync, recovery, reconcile — always carry the current value regardless of what the interval says.

```
// Interval-paced, at most one delta every 250ms.
NetworkMember<float> health = new(sendInterval: SendInterval.Short);

// Divine projection; sendInterval is ignored while this mode is set.
NetworkMember<Vector3> position = new(transmissionMode: TransmissionMode.Divine);
```

Extrapolation decides what silence means

`isExtrapolationEnabled` is a second setting you choose, and it applies under both modes. `true` (the default) keeps receivers moving the value through the sender's silence; `false` holds them still until told. The two trade off differently when an update is lost — `true` favors staying live at the risk of a brief overshoot, `false` favors staying accurate at the risk of a brief stall. Try `true` first; set it to `false` if a lost update is

visibly overshooting for your use case.

Under `Interval`, `true` lets a receiver render up to one send interval ahead of the newest value while it is moving, which covers an update that never arrives, and the sender sends one small "stopped" update when the value stops changing so the receiver settles exactly on it. That costs a tiny overhead on each update plus one extra update each time the value stops. With `false`, the receiver never renders past the newest value and there is no extra cost.

A receiver adopts the sender's mode automatically

Both `TransmissionMode` and `isExtrapolationEnabled` are construction defaults, optionally overridden per instance at runtime. A receiver doesn't need to be built with the same values: every full serialization carries the sender's mode and extrapolation setting, and the receiver adopts them automatically. Nothing here has to be toggled in lockstep ahead of time, though each value is meant to be set once rather than changed mid-stream.

Pro gating

`Divine` and every `SendInterval` above `Normal` are Pro, and both are gated the same way. They compile fine in a free build, because an enum can't be split by edition, so the values stay declared, but they change nothing there. A member set to `TransmissionMode.Divine` replicates with ordinary deltas, as `Interval` does. `SendInterval.Short` and `SendInterval.Long` pace nothing: free resolves every span to a single tick and replicates every changed tick, which is what `Normal` already means. That's what lets one prefab load in either edition without a `#if` around every member declaration.

Interval and interpolation

A paced member's interpolation window is its resolved send interval, not a fixed constant. A member sent every tick animates across that single tick; a member paced to `Short` or `Long` animates its arrival across exactly the span it was sent on. The receiver never has to be told the window separately, because both peers construct the member with the same interval — it's part of the component's field declaration — so the span is already known on both ends.

No distance-driven tier

`SendInterval` declares only `Normal`, `Short`, and `Long`. There is no tier that scales with distance to an observer or with anything else; the three spans are the whole set, chosen once per member and resolved the same way for every observer.

How replicated state works

See Start Here / How replication works for the pipeline overview. This page covers only the member-and-component contract every other page in this section assumes.

Containment

A `NetworkSystem` holds `NetworkComponents`, up to 64 per system. A `NetworkComponent` holds `NetworkMembers`, at most 64, one for each bit of the `ulong memberFlags` that `OnMembersChanged` reports. The generator does not check that limit on a component, so one that declares more still builds but does not replicate correctly. `Constants.MaximumNetworkMemberCount` (63) is a separate limit, on the structs and classes a member's value is made of: the generator reports `SERIALIZERS000` once such a type has 63 serialized fields and properties, so 62 is the most one can hold.

Data lives in components, not in the `NetworkSystem` subclass itself. `NetworkSystem` tracks identity, lifecycle, and observers; the values that actually replicate are declared on the `NetworkComponents` attached to it. A component exposes its own `Write`, `WriteDelta`, `Read`, and `ReadDelta`, and it is these that generated code fills in per member.

Full versus delta

A member simply carries the value as of the last write. There is no separate late-joiner payload to author.

When a new observer starts watching a system, that system serves a full snapshot of every component to that observer; every serialization after that is a delta of what changed. This applies uniformly: a full, a recovery serve, and an interest serve all ride the same absolute encoding, while an ordinary tick's serialize rides a delta. A member's author never writes two versions of anything - the full path and the delta path both read the same current value, just at different granularity.

One write pass, one read pass, per tick

Each tick raises `NetworkComponent.OnMembersChanged(ulong memberFlags, MemberChangeDirection memberChangeDirection)` up to twice per component: once for what this peer is sending, once for what it received.

- `MemberChangeDirection.Write` fires before any of the tick's writing runs, on the peer that made the change. It reports that the members were committed to the outbound pass, not that bytes left the process - a system with no observers, or one withheld from an observer, still raises this.
- `MemberChangeDirection.Read` fires after every packet of the inbound pass has landed, before reconciliation. A full apply (spawn, resync, reconcile, whole-component recovery) reports every member changed; a member-granular recovery reports only the members it repaired.

A host raises both directions in the same frame, write first, without waiting for its own packet to come back. Code that only cares about a remote change - anything calling `NetworkMemberBase.TryGetWritingClient` - must check for `MemberChangeDirection.Read`; that claim only resolves under a read.

Who may write

By default, `StateWriteAccess.Controller` (the enum's `0` value): the server, or the single controlling client, may write a system's state. Letting other observers write as well (`StateWriteAccess.AnyClient`) is a Pro opt-in, covered on its own page. A free build has no `StateWriteAccess` type, so there only the controller writes.

What a change costs

Four independent axes decide the price of a member changing, each covered on its own page:

- **Accuracy** - how finely a value's change is compared and encoded.
- **Compression level** - how tightly the wire encoding packs the value.
- **Transmission mode** - how the value's changes travel across ticks (e.g. absolute versus projected).
- **Send interval** - how often a changed member is actually allowed to send.

Vocabulary

Nucleus has its own terms; do not substitute names from another engine's netcode.

- **System**, not object: `NetworkSystem` is the networked entity.
- **Component**, not behaviour: `NetworkComponent` is a container of members within a system.
- **Member**, not variable: `NetworkMemberBase` is the individual replicated value.
- **Controller**, not owner: the client permitted to write a system's state by default.

Letting clients write state

What write access controls

Every `NetworkSystem` has a `StateWriteAccess`, layered beside control rather than replacing it. The controller may always write state; access widens who else can, beyond the controller alone.

```
public enum StateWriteAccess : byte
{
    Controller = 0,
    AnyClient = 2,
}
```

`Controller` is the default and the behavior every system had before write access existed: the server, or the single controlling client. `AnyClient` opens writes to the server and to any client that currently observes the system.

Ordinal 1 is deliberately vacant. `StateWriteAccess` is serialized into authored scenes, so renumbering `AnyClient` would silently reinterpret every scene that already carries a value of 1 as something else.

Setting and reading access

The server sets access with `TrySetWriteAccess`:

```
public bool TrySetWriteAccess(StateWriteAccess stateWriteAccess)
```

It's server-only; a client call is rejected. Widening a system's own permission state through a client call would make the permission client-settable, so the guard exists at the entry point rather than downstream. It also refuses to widen a system that holds a collection member (see below).

The current setting is readable from `WriteAccess`. On the client side, the read that matters is `CanLocalClientWriteState`: whether the server most recently told this client it may write. It's a capability hint the server pushes down, not something the client derives locally, and it's always `false` on the server.

Access widens permission, it never replaces observation

`AnyClient` is scoped to observers, not to every client in the world. The eligibility test is always "does this connection observe the system" - asked under every access, controller included. A peer that has stopped observing has, by definition, stopped being told what the value is, so there's nothing it could honestly be writing against.

The consequence worth repeating: losing interest in a system silently removes write permission with it. `AnyClient` means you may write what you can see. Interest changes and write permission changes are the same event.

What access does not change

`ControllerConnectionId` stays exclusive and server-assigned regardless of write access. Client state, prediction, and reconcile remain controller-only whatever the access says - widening write access adds a path for state members, it doesn't touch control.

Nothing travels as a grant

The server enforces every write from its own copy of `WriteAccess`. A client is told only which access is in force and whether it personally may write (`CanLocalClientWriteState`); it is never handed anything

resembling a permission token. A forged or replayed "I may write" claim from a client is inert, because the server never trusts the client's own read of its permission - it re-checks against `WriteAccess` on every incoming write.

The echo-withholding window

A client that writes state upstream would otherwise receive its own write back as a correction and stutter on it. To prevent that, a losing or contested write is shown locally for a window before the server's value is accepted:

```
public uint StateWriteConvergenceDeadlineTicks = UnsetConvergenceDeadlineTicks;

public const uint UnsetConvergenceDeadlineTicks = uint.MaxValue;
```

Left at `UnsetConvergenceDeadlineTicks`, the deadline is derived from the connection's measured round-trip time rather than fixed. Set it explicitly per system when the default feel is wrong - a shared UI value and a contested physics object don't want the same answer. `StateWriteConvergenceDeadlineTicks` itself is declared alongside the widening logic, so this window exists only in a Pro build; a free build has no echo-withholding at all.

Write access beyond the controller ships only in Pro

Write access beyond `Controller`, and the echo-withholding window that goes with it, are Pro. The gate is structural: both live in `NetworkSystem.WriteAccess.Pro.cs`, split from the free-side `NetworkSystem.WriteAccess.cs`. `StateWriteAccess` itself, `NetworkSystem.WriteAccess` and `NetworkSystem.TrySetWriteAccess` are Pro-only too, so a free build has none of them and code naming them does not compile. A free build's eligibility check admits the controller alone.

Two limits apply even with Pro:

- A system holding a collection member cannot be widened past `Controller`. `TrySetWriteAccess` rejects the call, because a client write to a collection arrives as an absolute with no operation log to relay to other observers.
- A client permitted to write on a system with more than one possible writer sends its members upstream as absolutes, not deltas. A member holds a single decode baseline, so two writers' deltas can't both reconstruct against it safely - only one sender per baseline can use the cheaper delta stream, and a multi-writer system gives that up.

TransformComponent

Using Unity? See NetworkTransform.

`TransformComponent` is the engine-neutral replicated pose of an object: position, rotation, and scale. It declares no space of its own — whichever adapter binds it decides whether the values mean world or parent-local, and that choice is expressed by which subclass a system carries, not by a flag on the component.

Members

Member	Type	Compression	Transmission mode
`Position`	`NetworkMember<Vector3>`	`CompressionLevel.Aggressive`	`TransmissionModeDefaults.Motion`
`Rotation`	`NetworkMember<Quaternion>`	default	`TransmissionModeDefaults.Motion`
`Scale`	`NetworkMember<Vector3>`	default	`TransmissionModeDefaults.Motion`

`Vector3` and `Quaternion` are `System.Numerics` types. `TransformComponent` never touches an engine's own vector or rotation type; an adapter converts at the boundary.

```
public readonly NetworkMember<Vector3> Position = new(CompressionLevel.Aggressive, transmi
ssionMode: TransmissionModeDefaults.Motion);
public readonly NetworkMember<Quaternion> Rotation = new(transmissionMode: TransmissionMod
eDefaults.Motion);
public readonly NetworkMember<Vector3> Scale = new(transmissionMode: TransmissionModeDefau
lts.Motion);
```

Setting all three at once

`TransformComponent` adds three methods that push a single choice across `Position`, `Rotation`, and `Scale` together, so the whole pose replicates as one unit rather than three independently configured members:

- `SetSendInterval(SendInterval sendInterval)`
- `SetTransmissionMode(TransmissionMode transmissionMode)`
- `SetExtrapolationEnabled(bool isExtrapolationEnabled)`

Apply the same value on every peer. A platform integration typically pushes a serialized interval, mode, or extrapolation setting on bind so this stays consistent without each peer configuring it by hand.

Subclassing

`TransformComponent` is a `partial class`, meant to be subclassed by an engine adapter that binds `Position`, `Rotation`, and `Scale` to a real transform and decides what space they're measured in. A subclass's members are part of its owning system's component list, so both peers must agree on which subclass — and therefore which space — the system carries.

TransmissionModeDefaults.Motion

All three members are built with `TransmissionModeDefaults.Motion` rather than a literal `TransmissionMode`. `Motion` resolves to `TransmissionMode.Interval` by default and is only raised to a projecting mode when Pro's projector registration runs at assembly load; a free build never raises it, so `Motion` resolves to `Interval` there.

Framework components use `TransmissionModeDefaults.Motion` instead of naming a mode directly so that one source reads correctly in both editions: divine projection where it's available, ordinary interval-paced deltas where it isn't. Naming `TransmissionMode.Divine` directly also compiles in a free build, where the member replicates with ordinary deltas as `Interval` does, so game code can name a `TransmissionMode` directly in either edition.

Judging and arbitrating client writes

Declining a value

Whether a client may write a `NetworkMember<T0>` at all is settled by control (and, in Pro, by `StateWriteAccess`) before the payload is even read. What the value should be is a separate question, and it can only be asked once the value is decoded. To answer it, implement `IStateWriteValidator<T0>` on the `NetworkComponent` that owns the member and return a `StateWriteAction` from `ValidateStateWrite`:

```
public partial class SwitchBankComponent : NetworkComponent, IStateWriteValidator<uint>
{
    private const uint SwitchCount = 3;

    public readonly NetworkMember<uint> ActiveSwitchIndex = new();

    public StateWriteAction ValidateStateWrite(NetworkMember<uint> networkMember, uint pro
posedValue, Connection writingClientConnection) =>
        proposedValue < SwitchCount ? StateWriteAction.Accept : StateWriteAction.Decline;
}
```

`StateWriteAction.Accept` lands the value exactly as it would with no validator present. `StateWriteAction.Decline` leaves the member on the value it already holds. `Accept` is the zero value, so a validator that falls through without expressing an opinion never accidentally declines a write.

The member's current value is still readable through `networkMember.Value` inside the call — the proposed value hasn't replaced it yet, which is what makes comparing the two possible here and nowhere else.

Why it's generic, and implemented once per type

`IStateWriteValidator<T0>` is generic over the member's value type, and a component implements it once for each type it wants judged. A component holding a `NetworkMember<uint>` and a `NetworkMember<bool>` implements the interface twice if it wants to judge both.

That's what keeps the value typed and off the heap. `NetworkComponent` itself is not generic, so a single non-generic hook would have to hand every value over as `object` and box each one on the inbound apply path. Splitting the hook per type avoids that entirely.

Telling two members of the same type apart

A component can hold more than one `NetworkMember<uint>`, and both route through the same `ValidateStateWrite(NetworkMember<uint>, uint, Connection)` implementation. The member is passed as a parameter for exactly this reason — the type alone doesn't identify which one is being written. Compare the passed member against your own fields instead of keying on anything the framework supplies:

```
if (ReferenceEquals(networkMember, ActiveSwitchIndex))
{
    // ...
}
```

Authority-side only

`ValidateStateWrite` is never consulted on a client, and never for the server's own writes. A client running it would be vetoing the value it's being told to hold; a server judging its own write would be asking

permission from the code that just made the decision.

A component that doesn't implement `IStateWriteValidator<T0>` for a given type isn't consulted at all — it pays a single type test on the apply path, nothing more.

What Decline does on the wire

Declining doesn't unwind anything. The bits for the value were already consumed off the stream before the validator ran, so the stream stays aligned regardless of the outcome. Nothing about a declined value is relayed onward — the other observers are never told about a value the server didn't take.

On the client whose write was declined, the member keeps showing that client's own value for the duration of the convergence window, then reverts to the server's value once the window closes. That reversion costs no traffic: the server's value never moved, so there's nothing new to send. It's picked up from the read baseline the decode already advanced.

A declined write is not a violation

Declining a value never raises anything and never counts against the client. The client proposed something the game didn't want, which is an ordinary outcome — not evidence of misbehavior. Reach for the violation pipeline only when a client sent something it should never have been able to send at all, not for a value that was merely rejected.

Contention: first write wins

When two clients write the same member on the same tick, arbitration is the default behavior and it applies whether or not a validator is present: the first write applied wins. Because the inbound walk visits connections in ascending `Connection.Id` order, "first write applied wins" and "lowest permitted connection Id wins" are the same statement. No preference is expressed beyond that ordering.

`Connection.Id` is arbitrary but stable for the life of a connection — it isn't chosen for fairness, and it isn't a security boundary. A client that disconnects and reconnects gets a fresh `Id`, so arbitration on `Id` can be farmed by reconnecting into a more favorable position. Anything that genuinely needs to matter — a claim that should survive a reconnect, or resist being gamed by one — has to be arbitrated on `Connection.Identity`, the value that names a client across reconnects, instead of on `Connection.Id`.

Attributing a landed write

To find out which client's write actually landed on a member this tick, call `TryGetWritingClient(out Connection connection)` on the `NetworkMember<T0>`. It's meant to be called from inside `NetworkComponent.OnMembersChanged` while under a read raise (`MemberChangeDirection.Read`), which runs while the tick that applied the write is still current — asking on a later tick reports `false`.

A server write, and a local write on either peer, never stamp a writing client, so asking about your own write honestly reports nothing. Only a value that genuinely arrived from a client over the wire is attributed, and a writer that has since disconnected also reports `false` rather than handing back a stale `Connection`.

Your first networked component

Using Unity? See Replicating a value from a Unity script.

Declare the component

A networked component is a `partial class` inheriting `NetworkComponent`, with one or more `readonly NetworkMember<T0>` fields:

```
public partial class SimpleStateComponent : NetworkComponent
{
    public readonly NetworkMember<int> Counter = new();
}
```

The `partial` keyword is not optional. A source generator scans the class for its `NetworkMember<T0>` fields and emits the other half of the type: `Write`, `WriteDelta`, `Read`, `ReadDelta`, and a per-component flags enum with one bit per member, in member-Id order. Without `partial` there is nowhere for that generated code to attach, and the component never serializes anything.

Add more fields for more state; each gets its own bit in the generated flags enum and its own Id within the component.

Rent a system that carries it

A `NetworkComponent` doesn't exist on its own; it lives on a `NetworkSystem`. `NetworkSystemPool.TryRent` takes the system type and one type argument per component it should carry, and hands back a started system through its `out` argument. It returns `false` when no `CoreManager` could be found or the start was refused:

```
if (!NetworkSystemPool.TryRent<NetworkSystem, SimpleStateComponent>(coreManager, out Netwo
rkSystem? system, canStartSystem: true))
    return;

system.TryGetComponent(out SimpleStateComponent component);
component.Counter.Value = 31337;
```

`canStartSystem` defaults to `true`. Renting starts the system immediately unless you pass `false`, in which case you start it yourself later. A system rented this way carries no engine object at all - it's pure C# on both sides of the wire.

Find it again on a peer

A system rented purely from code reaches the other peer as a bare wire spawn: its `PlatformId` is `NetworkSystem.UnsetPlatformId`, so it never reaches a spawn handler (`INetworkSystemSpawnHandler.OnDynamicSystemSpawned` is not called for it). There's no engine object to have hung a reference off in advance, so nothing on the receiving peer already knows this system exists.

`NetworkSystemPool.Watch` is how you find it anyway. It registers a standing subscription for every system of a given composition - a system type plus a set of component types - and hands each matching system to your handler as it starts locally or arrives over the wire:

```
NetworkSystemWatch watch = NetworkSystemPool.Watch<NetworkSystem, SimpleStateComponent>(
    coreManager,
    acquiredSystem =>
    {
        acquiredSystem.TryGetComponent(out SimpleStateComponent component);
        int value = component.Counter.Value;
    })!;
```

The match is on composition, not on type identity: any `NetworkSystem` (or subclass) carrying a `SimpleStateComponent` satisfies this watch, regardless of what else it carries. That's why the type argument list matters - a watch for a plain `NetworkSystem` with no component arguments would match every system in the world.

Call `NetworkSystemPool.Unwatch(watch)` when you no longer need it. An optional released handler on `Watch` fires when an acquired system stops, which for a wire-constructed system means it's about to return to its pool - drop any reference you took before that call returns.

Read the value back

`NetworkSystem.TryGetComponent<T0>` is how you get a component off a system, on either peer:

```
if (system.TryGetComponent(out SimpleStateComponent component))
{
    component.Counter.Value = 42;   // write - only meaningful on the system's writer
    int current = component.Counter.Value;  // read - always safe
}
```

Setting `Value` marks the member changed for the current tick and queues it for the next delta. Reading `Value` returns whatever the ring's live head currently holds, which on a non-writing peer is the last value applied from the wire.

The overrides you don't call

`NetworkComponent` declares `Write`, `WriteDelta`, `Read`, `ReadDelta`, `Initialize`, `Deinitialize`, and `OnReturn` as `public virtual`. The source generator overrides `Write`/`WriteDelta`/`Read`/`ReadDelta` for you from your `NetworkMember<T0>` fields, and the framework calls `Initialize`/`Deinitialize`/`OnReturn` at the points in a component's lifecycle where they apply - rent and system-start, system-stop, and pool-return, respectively.

None of these are yours to call. They exist for the generator to fill in and for the framework to invoke on its own schedule. If you need to react to a value changing, override `OnMembersChanged` instead - it's called once per tick per direction whenever any member of the component changed, and it's the supported hook for reacting without polling.

Systems, Objects and Spawning

Pages in this category:

- NetworkSystem reference
- What a system is
- Renting and finding systems in code
- Starting and stopping systems
- Parents, children and hierarchy
- System groups
- Binding spawns to your own engine
- References between systems
- Spawn compensation
- Predicted spawn and despawn requests

NetworkSystem reference

Identity

A `NetworkSystem` carries two kinds of identity: the network Id every peer agrees on, and the platform Id that ties it to an engine object.

Member	Type	Meaning
`Id`	`uint`	The Id unique to the network for this instance. `UnsetId` (`0`) until the system is started.
`PlatformId`	`uint`	The Id of the object for the running platform this system belongs to: a scene object's identifier within its scene, or a dynamic prefab's local identifier within its shard. `UnsetPlatformId` (`0`) until assigned.
`PrefabBundleId`	`ushort`	The content bundle owning this system's prefab, pairing with `PlatformId`. Meaningful only for a dynamically spawned prefab; zero is the base build's default shard.
`RequiredBundleId`	`ushort`	The bundle a client must hold before this system may spawn for it. Falls back to `PrefabBundleId` when no override is set.
`SetRequiredBundleId(ushort)`	method	Overrides `RequiredBundleId` without touching `PrefabBundleId`, so the wire identity of the prefab is unaffected.
`UnsetRequiredBundleId`	`const ushort`	`ushort.MaxValue`, **not** `0` like the other Unset constants — zero is a valid requirement, naming the base build.
`IsSceneObject`	`bool`	True when `PlatformId` identifies a scene object rather than a dynamically spawned prefab.

`SceneHandle`	`uint`	The live scene instance this system belongs to. `UnsetSceneHandle` (`0`) when it belongs to no server-opened scene.
`GroupId`	`uint`	The Id of the `NetworkSystemGroup` this system belongs to. `UnsetGroupId` (`0`) when ungrouped.
`ParentId`	`uint`	The `Id` of the system this one is parented to. `UnsetParentId` (an alias of `UnsetId`) when parented to nothing.

```
if (system.RequiredBundleId != NetworkSystem.UnsetRequiredBundleId)
{
    // An explicit override is in effect, distinct from PrefabBundleId.
}
```

`RequiredBundleId` is server-side only: it feeds interest resolution and is never replicated, so reading or setting it costs no bandwidth.

Lifecycle

`State` is a `NetworkSystemState`:

Value	Meaning
`Starting`	Accepted but not yet live. Normally transient; a start too late for the current tick's state flush holds here until the next tick increments.
`Started`	Live and networked.
`Stopping`	A stop has been accepted; the actual stop happens at the end of the tick.
`Stopped`	Not networked.

`IsStarted` is `State is NetworkSystemState.Started`.

What each identity read is worth during a transition:

- **Starting**: `IsStarted` is false. If the start was deferred past the tick's state flush, `Id` is still `UnsetId` and the rest of the identity (`PlatformId`, `GroupId`, `SceneHandle`) is not yet assigned — all of it is stamped together when the system reaches `Started`.
- **Stopping**: `IsStarted` is false, but `Id`, `PlatformId`, `GroupId`, `SceneHandle` and `ParentId` still hold their last live values. Nothing clears them until the despawn completes at end of tick and the system returns to its pool.

Role reads

Member	Meaning

`IsServerStarted`	Whether the local peer runs the server.
`IsClientStarted`	Whether the local peer runs an authenticated client.
`IsHostStarted`	Whether the local peer runs as a host (a started server alongside an authenticated local client).
`CoreManager`	The `CoreManager` managing this system. Every other manager (transport, interest, RPC, systems) is reached from here.

In host mode, `IsServerStarted` and `IsClientStarted` are both true at once.

```
if (system.IsServerStarted)
{
    // Authority-only logic.
}
```

Composition

Member	Meaning
`ComponentCount`	The number of `NetworkComponent`s attached to this system.
`TryGetComponent<T0>(out T0 component)`	Retrieves a contained `NetworkComponent` of type `T0`, such as a transform component. Returns false when none is attached. When the system carries the type more than once, this is the first.
`TryGetComponent<T0>(byte componentId, out T0 component)`	Retrieves the component at a component Id when it is of type `T0`. The Id is the order the component was added in, the same on every peer, so this reaches every instance of a repeated type.
`MaximumComponentCount`	The most components one system can hold: 31.
`GroupMemberCount`	The number of members this system's group currently holds. Zero for an ungrouped system. On the server this is the live group count; on a receiver it is the count the full header carried.

```
if (system.TryGetComponent(out MyTransformComponent transform))
{
    // Read or write members on the component the system rented.
}
```

Diagnostics

`NetworkSystemExtensions.AsString(this NetworkSystem)` builds a human-readable one-liner for logging: state, platform id, group id, and id. It returns `"IsNull"` when called on a null reference, so it is safe to log without a null check.

```
Logger<MyType>.LogInformation($"System: [{system.AsString()}]");
```

Where the rest lives

This page indexes identity, lifecycle, role and composition. The rest of `NetworkSystem`'s surface is documented in its own category:

Category	Covers
Control	`ControllerConnectionId` and who drives the system.
Write access	`StateWriteAccess` and per-member write permissions.
Interest	Observer registration and what a connection is served.
RPCs	`IRpcScope`, `RpcSendAccess`, and remote procedure calls scoped to a system.
Client state	`MemberRole.Predict` members, sent upstream by the controlling client every tick.
Reconcile	`ReconcileRequired`, `ResultMismatched`, the simulation callbacks and correcting predicted state.

References between systems

Interest means a peer can lose sight of a system and regain it later, on ticks it does not control. A plain identifier survives that — the number does not change — but a plain object reference does not: the instance a peer resolved last tick may not exist on this peer at all next tick, and the one it resolves to next week is not guaranteed to be the same managed object. `NetworkSystemMember` and `NetworkConnectionMember` exist to hold a reference that keeps working across that gap: replicated as an identifier, resolved to a local instance whenever this peer can currently see the target, and re-resolved on its own whenever interest changes.

NetworkSystemMember

`NetworkSystemMember` points at another `NetworkSystem`.

```
public NetworkSystem Value { get; set; }
public uint TargetId { get; }

public void Initialize(NetworkComponent networkComponent, byte id);
```

`Value` is the local instance the reference currently resolves to, or `null` when it references nothing or references something this peer cannot currently see. `TargetId` is the raw identifier and is readable either way — it holds the target's Id whether or not `Value` has resolved it, because the Id is the only thing that can resolve the reference later.

Declare one as a field on a `NetworkComponent`, the same way a `NetworkMember<T0>` is declared:

```
public readonly NetworkSystemMember Target = new();
```

`Initialize` wires the member to its owning component and gives it its member Id within that component; the generator calls it, the same as it does for every other member family.

NetworkConnectionMember

`NetworkConnectionMember` is the same shape, pointed at a peer instead of a system:

```
public Connection Value { get; set; }
public uint TargetId { get; }

public void Initialize(NetworkComponent networkComponent, byte id);
```

`Value` and `TargetId` mean the same thing here as they do on `NetworkSystemMember`. The one place they diverge is what happens when the referenced peer leaves. Connection identifiers are recycled — the next peer to connect can be handed the same Id a departed peer used — so a reference cannot simply keep resolving to whatever now holds that Id. When the peer a `NetworkConnectionMember` names departs, `Value` returns to `null` rather than silently aliasing whoever arrives next. The Id itself is left alone, so the reference resolves again if that same peer genuinely reconnects with the same identifier.

Resolution is a subscription, not a one-time lookup

Assigning `Value` does not do a one-shot lookup that is later forgotten. Both member types subscribe their target Id to a registry for as long as the reference exists, and that subscription is what makes the member keep working as interest changes: when the target leaves this peer's routing table, the member drops back to `Value == null`; when the target (or, for a connection, that same Id) enters the routing table again, the member resolves it again automatically, with no code re-checking or re-fetching anything.

This is why a reference is written and read even while `Value` is `null` — `TargetId` is what rides the wire, and it is what lets the member pick the resolution back up later without anyone re-sending the assignment.

An unset reference costs one bit, a set one costs about a byte more

An unset reference costs one bit on the wire. A set reference costs that bit plus the Id, and an Id in the low range costs about a byte. A game that holds no references between systems pays essentially nothing for the feature existing.

Why not a value member

`NetworkSystemMember` and `NetworkConnectionMember` do not derive from the same base a scalar member does, and they don't behave like one. An identity is exact or it's wrong — there's nothing to interpolate between two identities, no snap threshold for a value that either matches or doesn't, no accuracy tolerance to compare against, and no delta projection to run on something with no continuous path. All of that machinery exists for values that move smoothly through a numeric space; a reference just names a thing, so none of it applies.

Reacting to resolution

A reference resolving or losing its target does not fire an event of its own — it's reported the same way any other member change is, through the component's `OnMembersChanged` callback. See Reacting to changes: OnMembersChanged for how to read that callback and tell a reference's resolution apart from a regular value write.

Predicted spawn and despawn requests

> **Using Unity?** See Client-predicted spawning in Unity.

Predicted spawning lets a client create or remove an object immediately, ahead of the server agreeing, instead of waiting a round trip to find out whether it may. It is a Pro feature.

Deny by default

Nothing is predictable until two things are both true:

- The object's type declares a `PredictedSpawnPolicy` other than `None` — `Spawn`, `Despawn`, `SpawnAndDespawn`, or `AllowUndeclaredComposition` layered on top of one of those. The policy lives on `NetworkSystem.PredictedSpawnPolicy` and defaults to `None`, so an object nobody has authored an opinion for predicts nothing.
- The server has registered a `SystemManager.PredictedSpawnValidator` (for spawns) or `SystemManager.PredictedDespawnValidator` (for despawns). Every predicted spawn is refused while `PredictedSpawnValidator` is `null`, and every predicted despawn is refused while `PredictedDespawnValidator` is `null`.

Setting `PredictedSpawnValidator` is also what turns on lease serving: a session that never registers one can never admit a predicted spawn, so there's no point leasing identifiers to its clients. Registering a validator serves the lease to every client already authenticated and to everyone who authenticates afterwards.

`PredictedSpawnPolicy` is not replicated. Each peer reads its own copy — for a prefab, that's authored on the prefab and so the same everywhere; for a pure-code object, it's whatever that peer's own code set. The server judging a predicted despawn reads its own object's policy rather than trusting the asking client's account of it.

`AllowUndeclaredComposition` is a separate concern from `Spawn`/`Despawn`: it permits a client to describe an object whose component composition no call site in this build declares. Without it, an undeclared composition is refused outright, because it's the shape of a client fabricating systems and components of its own.

Asking

On a client, `SystemManager` exposes two methods:

```
public bool EnsureStartPredictedSystem(NetworkSystem networkSystem, uint platformId, ushor
t prefabBundleId, uint sceneHandle, out PredictedSpawnRefusal predictedSpawnRefusal);

public bool EnsureStopPredictedSystem(NetworkSystem networkSystem, out PredictedSpawnRefus
al predictedSpawnRefusal);
```

`EnsureStartPredictedSystem` creates the object locally right away and sends the server a request describing it. Set the object's opening state before calling it, not after — the request carries a snapshot of the object's components taken during the call, and anything written afterward only reaches the server if the object is also replicating it. `platformId` is the prefab's identifier within its shard (or `NetworkSystem.UnsetPlatformId` for an object with no engine prefab behind it), `prefabBundleId` is the content shard the prefab was registered from, and `sceneHandle` is the live scene instance the object belongs to (or `NetworkSystem.UnsetSceneHandle`).

`EnsureStopPredictedSystem` doesn't remove anything. It marks the object as going away —

`NetworkSystem.IsPredictedDespawnPending` becomes true and `PredictedDespawnPending` fires — and sends the server a request. The object stays started, routed and replicating in the meantime; inbound state still applies to it. If the server agrees, its ordinary despawn arrives and removes the object through the same path any despawn uses. If it refuses, the pending flag clears and `PredictedDespawnRejected` fires on the same instance, which never went anywhere.

Both methods return `false` on a local refusal and report why through the `out PredictedSpawnRefusal` parameter, which is also raised as `SystemManager.PredictedSpawnRefused`. This is refusal by the client's own framework, before anything was created, removed, or sent — a host predicting nothing, an empty lease, a policy that doesn't admit what was asked, and so on. It's distinct from the server refusing a request that actually reached it.

Judging

On the server, a spawn request is judged by a delegate:

```
public delegate PredictedSpawnDecision PredictedSpawnValidatorHandler(in PredictedSpawnReq
uestContext predictedSpawnRequestContext);
```

`PredictedSpawnRequestContext` carries the whole of what the client is asking for, without anything having been constructed to produce it: `Connection`, `SystemType` (resolved against this build's own registry, not trusted from the wire), `Components` (a `PredictedSpawnComponents` view over the declared composition), `SystemId`, `GroupId`, `PlatformId`, `SceneHandle`, `PrefabBundleId`, `SystemBundleId`, `SystemLocalId`, `IsDeclaredComposition`, and `HasPrefab`. When `PredictedSpawnStateInspectionEnabled` is on, `PredictedSystem` is the actual object built with its opening state already applied, so the validator can judge real values instead of only shape.

The validator returns a `PredictedSpawnDecision`:

```
public static PredictedSpawnDecision Permit(bool isRequesterController = false);
public static PredictedSpawnDecision Refuse();
```

`Permit` optionally hands the requesting client control of the object once it's built; the default is off, since the common predicted spawn is a projectile the server simulates and nobody controls. Refusal is the default, including when no validator is registered at all.

A despawn request is judged the same shape, with less to check because the server already holds the object:

```
public delegate PredictedDespawnDecision PredictedDespawnValidatorHandler(in PredictedDesp
awnRequestContext predictedDespawnRequestContext);
```

`PredictedDespawnRequestContext` carries `Connection` and `System` — the server's own live copy of the object, readable directly rather than described. The validator returns a `PredictedDespawnDecision` via `Permit()` or `Refuse()`, refusing by default.

The lease

A client can't spend an identifier it doesn't already have, so the server leases it a batch ahead of time. Four settings on `SystemManager` control the lease, and two report its current state.

Member	What it controls
`PredictedSpawnLeaseSize`	How many spawns a client is kept holding — the burst it can predict in one round trip before waiting on the server. Default 8.

`PredictedSpawnSystemIdsPerSpawn`	How many `NetworkSystem` identifiers each leased spawn carries — the most systems one predicted object can be built from. Default 8. An object needing more is refused as `PredictedSpawnRefusal.LeaseExhausted`.
`PredictedSpawnSystemLeaseSize`	`PredictedSpawnLeaseSize * PredictedSpawnSystemIdsPerSpawn` — the total system identifiers a client is kept holding. Read-only; it's the product, not a separate knob.
`PredictedSpawnsPerSecond`	How fast a client earns replacement identifiers, in spawns per second, or zero to replace them as fast as they're spent. Default 30.
`PredictedSpawnLeaseCount`	On a client, how many more objects it can predict right now. Zero on the server.
`PredictedSpawnSystemLeaseCount`	On a client, how many more `NetworkSystem` identifiers it's holding. Zero on the server.

`PredictedSpawnLeaseSize` and `PredictedSpawnSystemIdsPerSpawn` count different things and it's easy to conflate them: the lease size is a quota on *objects* (one grouped prefab of five systems still draws one, the same as a single ungrouped system), while the per-spawn identifier count is the *supply* each of those objects draws from. Raise the lease size for a weapon that fires in real bursts; raise the per-spawn count for a prefab built from more systems.

`PredictedSpawnsPerSecond` is what stops the lease size alone from setting the real ceiling. Without pacing, a replacement served the instant a spawn is admitted makes the effective limit one lease per round trip — which hands the best-connected client the highest ceiling, since a 20ms link earns replacements far faster than a 200ms one for the same lease. The rate is what actually bounds a client, with the lease size only as the burst on top of it. It's counted per identifier rather than per object: a grouped prefab of three systems draws three against the rate, whatever the game calls one spawn.

A lease that's run dry answers `PredictedSpawnRefusal.LeaseExhausted` locally, without a request ever reaching the server.

Outcomes and events

On the predicting client, `NetworkSystem` raises:

- `PredictedSpawnConfirmed` — the server's snapshot arrived, meaning it agrees the object exists. Raised before the snapshot's own state is applied.
- `PredictedSpawnRejected(PredictedSpawnRejectReason)` — the server refused the object, raised immediately before the local copy is torn down.
- `PredictedDespawnPending` — the client has asked the server to remove this object and is waiting.
- `PredictedDespawnRejected(PredictedDespawnRejectReason)` — the server refused the removal. The object was never removed, so it just carries on.

`PredictedSpawnRejectReason` has eight values: `NotPermitted` (the ordinary refusal — the validator said no), `UnleasedId`, `UnknownType`, `SceneNotEntered`, `GroupMismatch`, `Malformed`, `NotPermittedByPolicy`, and `UndeclaredComposition`. `PredictedDespawnRejectReason` has four: `NotPermitted`, `NotPermittedByPolicy`, `UnknownSystem`, `NotObserved`.

On the client's `SystemManager`, `PredictedSpawnRefused` fires for a refusal the local framework made on its own, before anything reached the server — see Asking above.

What happens to a predicted object's simulated state once the server confirms it is controlled by `NetworkSystem.PredictedSpawnConfirmMode`:

- `ApplyAuthoritativeState` (default) — the confirming snapshot applies in full, so the object lands exactly where the server built it. Any divergence between what the client predicted and what the server built is shown, not hidden.
- `KeepPredictedState` — the snapshot is still read, but its member values are discarded; the object keeps the path the client already simulated, adopting only identity, controller and placement from the confirmation. Use this for an object whose local path must not visibly jump, typically a projectile the player is already watching.

When the server never answers

The request is sent once, and the client's upstream redundancy (`ClientRedundancy`) resends it on the ticks after, so a single lost packet does not lose it. If no answer arrives at all, the client gives up and despawns its local copy, raising `PredictedSpawnRejected` with `NotPermitted` first, as for an ordinary refusal. It waits two round trips, but never less than half a second and never more than three seconds, so on a bad link an object nobody agreed to does not stay on screen for long.

An answer can still arrive after the client gave up: the server admitted the spawn, but its confirmation was lost until after the timeout. The client then receives the server's object as an ordinary spawn, with the server's values, so both peers still end up holding the same object.

Inspecting state before deciding

`PredictedSpawnStateInspectionEnabled` builds the predicted object and reads its opening state before the validator runs, so the validator can judge actual values — a projectile's speed, a spawn position — rather than only the object's declared shape and type. It's off by default, because most validators don't need it: a composition and a prefab identity answer most questions.

The cost lands only on a refused request. With inspection on, the instance built for the look is the same one started if the validator permits the spawn, so the approved path pays nothing extra; only a refusal pays for a construction it didn't end up using.

Renting and finding systems in code

Using Unity? See NucleusBehaviour: the script base class.

Renting a system

`NetworkSystemPool.TryRent<TSystem, TComponent0...>(coreManager, out networkSystem, canStartSystem)` gets you a `NetworkSystem` with no engine object behind it at all. The generated overloads run from zero components up to thirty-one (`TComponent0` through `TComponent30`), and each one rents `TSystem`, attaches every named component, and starts the system unless you pass `canStartSystem: false`. It returns `true` with the system in `networkSystem`, typed as `TSystem`. It returns `false` when no `CoreManager` could be found or the start was refused, and it logs the reason.

```
if (!NetworkSystemPool.TryRent<NetworkSystem, MatchClockComponent>(coreManager, out Networ
kSystem networkSystem, canStartSystem: true))
    return; // no CoreManager instance could be resolved, or the start was refused

if (networkSystem.TryGetComponent(out MatchClockComponent matchClockComponent))
{
    matchClockComponent.Round.Value = 1;
    matchClockComponent.SecondsRemaining.Value = 10f;
}
```

`TSystem` is nearly always the plain `NetworkSystem` type itself; what distinguishes one rent from another is the component list, not the system type.

Renting a count decided at runtime

The generated overloads fix the component list when your code compiles. When the number comes from data instead (one component per seat in a vehicle the level designer configured, say), rent one component type a runtime number of times:

```
public partial class SeatComponent : NetworkComponent
{
    public readonly NetworkMember<bool> IsOccupied = new();
}

byte seatCount = 4;

if (!NetworkSystemPool.TryRent<NetworkSystem, SeatComponent>(coreManager, seatCount, out N
etworkSystem vehicleSystem))
    return; // no CoreManager, a count outside 1 to NetworkSystem.MaximumComponentCount, o
r the start was refused

for (byte componentId = 0; componentId < seatCount; componentId++)
{
    if (vehicleSystem.TryGetComponent(componentId, out SeatComponent seatComponent))
        seatComponent.IsOccupied.Value = false;
}
```

The components take the Ids `0` to `seatCount - 1` in the order they were added, which is also the order a receiver rebuilds them in, so component Id 2 is the same seat on every peer. Nothing extra rides the wire for this: a spawn already carries each component's type. A count of zero, or more than `NetworkSystem.MaximumComponentCount` (31), is refused and logged.

Returning one

A rented system is pooled, not new'd, so give it back rather than letting it fall out of scope:

- `NetworkSystemPool.Return<TSystem>(networkSystem)` returns it to `ResettableObjectPool<TSystem>`.
- `NetworkSystemPool.ReturnAndNullifyReference<TSystem>(ref networkSystem)` does the same and clears your reference to null.

A started system does not go through either of these directly — stop it through `SystemManager.EnsureStopSystem(networkSystem, isPoolReturnRequestedOnDespawn: true)`, and the pool return happens as the despawn completes. A receiver's copy, built by the framework from the wire, returns to its own pool automatically once its despawn is applied; you never call `Return` on that one.

The problem: nothing to find it by

A system rented this way carries `NetworkSystem.UnsetPlatformId`. `INetworkSystemSpawnHandler` is only consulted for a spawn whose `PlatformId` is set, so a pure-code system never reaches it — there is no `TryBindSceneObject` call and no `OnDynamicSystemSpawned` call to hang a reference off. On the peer that rented it you already have the reference the rent call returned. On every other peer, the system just starts; nothing tells that peer's code it exists, and there is no identifier it could have known in advance to look one up by.

The answer: watching for a composition

`NetworkSystemPool.Watch<TSystem, TComponent0...>(coreManager, acquired, released)` registers a standing subscription for every system of a given type and component composition, on whichever peer registers it — the one that rents the system, the one that receives it over the wire, or both. It returns a `NetworkSystemWatch`; hand that back to `NetworkSystemPool.Unwatch(watch)` when you're done.

```
private NetworkSystemWatch _matchWatch;

private void OnMatchAcquired(NetworkSystem networkSystem) => _matchSystem = networkSystem;

private void OnMatchReleased(NetworkSystem networkSystem)
{
    if (_matchSystem == networkSystem)
        _matchSystem = null;
}

public void Start()
{
    _matchWatch = NetworkSystemPool.Watch<NetworkSystem, MatchClockComponent>(coreManager,
 OnMatchAcquired, OnMatchReleased);
}

public void Stop()
{
    NetworkSystemPool.Unwatch(_matchWatch);
    _matchWatch = null;
}
```

`NetworkSystemAcquiredHandler` fires for every system that satisfies the watch, whether it started here or arrived off the wire. `NetworkSystemReleasedHandler` fires as an acquired system stops; drop whatever you took in `acquired` there; a wire-constructed system returns to its pool immediately after that

call, so a reference kept past it can end up naming a different spawn entirely.

Matching is by **composition, not type**. Nearly every rent asks for a plain `NetworkSystem` and differs only in the components attached, so naming the right components in the watch is what tells it apart from every other system in the world. Registering a watch also replays it against every system already started, so it works whether it is set up before or after the system exists.

A worked sample

The server rents and starts a match clock with no engine object:

```
public class MatchClock : INetworkLoopStepCallback
{
    public NetworkSystem NetworkSystem { get; private set; }

    private readonly CoreManager _coreManager;
    private float _secondsPerTick;

    public MatchClock(CoreManager coreManager) => _coreManager = coreManager;

    public NetworkLoopSteps GetNetworkLoopSteps() => NetworkLoopSteps.EarlyFixedUpdate;

    public bool TryStart()
    {
        if (!NetworkSystemPool.TryRent<NetworkSystem, MatchClockComponent>(_coreManager, o
ut NetworkSystem rentedNetworkSystem, canStartSystem: true))
            return false;

        NetworkSystem = rentedNetworkSystem;

        _secondsPerTick = 1f / _coreManager.NetworkLoopManager.TickRate;

        if (NetworkSystem.TryGetComponent(out MatchClockComponent matchClockComponent))
        {
            matchClockComponent.Round.Value = 1;
            matchClockComponent.SecondsRemaining.Value = 10f;
        }

        _coreManager.NetworkLoopManager.RegisterNetworkLoopStepCallbacks(this);

        return true;
    }

    public void OnNetworkLoopStep(NetworkLoopSteps networkLoopStep, StepDelta stepDelta)
    {
        if (NetworkSystem is null || !NetworkSystem.TryGetComponent(out MatchClockComponen
t matchClockComponent))
            return;

        matchClockComponent.SecondsRemaining.Value -= _secondsPerTick;
    }
}
```

The receiver watches for the same composition and is handed that exact system, whether it registered before the rent happened (via the replay) or the spawn arrives afterward:

```
public class MatchReadout
{
    public NetworkSystem MatchSystem { get; private set; }

    private NetworkSystemWatch _matchWatch;

    public void Start(CoreManager coreManager)
        => _matchWatch = NetworkSystemPool.Watch<NetworkSystem, MatchClockComponent>(coreM
anager, OnMatchAcquired, OnMatchReleased);

    private void OnMatchAcquired(NetworkSystem networkSystem) => MatchSystem = networkSyst
em;

    private void OnMatchReleased(NetworkSystem networkSystem)
    {
        if (MatchSystem == networkSystem)
            MatchSystem = null;
    }
}
```

Both sides drive the same real loop: the server's `MatchClock` ticks `SecondsRemaining` down every `EarlyFixedUpdate`, and `MatchReadout.MatchSystem` reads the replicated value off the very system the watch handed it, on whichever peer it runs on. Neither side special-cases which peer it is.

When to use SystemManager.SystemStarted instead

`SystemManager.SystemStarted` and `SystemManager.SystemStopped` are the broad half of the discovery surface: they raise for *every* system that starts or stops on this peer, regardless of composition. Reach for them instead of a watch when you genuinely want everything and will sort it out yourself — logging, bookkeeping across arbitrary system types, or a single subscriber that dispatches by type. A subscriber that wants only systems of a known composition should register a `NetworkSystemWatch` instead and be handed just those.

Who May Send a Call, and How Many

What send access controls

Every `NetworkSystem` has an `RpcSendAccess`, layered beside control exactly as `StateWriteAccess` is layered beside write control.

```
public enum RpcSendAccess : byte
{
    Controller = 0,
    AnyClient = 2,
}
```

`Controller` is the default: only the server, or the single controlling client, may send the system a call. `AnyClient` opens sending to the server and to any client that currently observes the system.

Ordinal 1 is deliberately vacant. The value is serialized into authored Unity scenes, so renumbering `AnyClient` would silently reinterpret every scene that already carries a value of 1 as something else.

Setting and reading access

The server sets access with `TrySetRpcAccess`:

```
public bool TrySetRpcAccess(RpcSendAccess rpcSendAccess)
```

It's server-only; a client call is rejected, since letting a client widen its own send access would make the permission client-settable.

The current setting is readable from `RpcAccess`. On the client side, the read that matters is `CanLocalClientSendRpc`: a one-bit capability hint the server most recently declared, telling the local client whether it personally may send. It exists so a well-behaved client never sends a call the server is going to refuse, and it's always `false` on the server.

Nothing travels as a grant

The server enforces every call from its own copy of `RpcAccess`. A client is told only which access is in force and whether it personally may send (`CanLocalClientSendRpc`); it is never handed anything resembling a permission token. A forged "I may send" claim from a client is inert, because the server re-checks every incoming call against `RpcAccess` and its own observer roster.

Access widens permission, it never replaces observation

`AnyClient` is scoped to observers, not to every client in the world. Losing interest in a system silently removes the right to call it along with the interest itself - `AnyClient` means you may call what you can see.

Kept separate from state write access

`RpcSendAccess` and `StateWriteAccess` are deliberately different knobs. "You may ask this object to do something" and "you may overwrite this object's state" are different grants - folding them into one would mean a game that wants observers to fire a call at an object has to open its state stream to them as well.

RpcSendPermissionViolation

A call sent to a system without permission raises `RpcSendPermissionViolation`:

```
public struct RpcSendPermissionViolation : IViolation
{
    public uint SystemId;
}
```

`SystemId` is the Id of the system the sender addressed without permission. It's raised at most once per drain, however many refused calls that drain carried.

The violation reaches a well-behaved client too, on a revocation race: permission is withdrawn while calls already sent are still in flight. That's why it's a violation the game decides on rather than a discard the framework settles alone. Subscribe through `ViolationManager.RpcSendPermissionViolationDetected`.

Capping calls per drain

`RpcManager` bounds how many inbound calls one connection may deliver in a single drain:

```
public uint MaximumInboundRpcsPerConnectionPerDrain = DefaultMaximumInboundRpcsPerConnecti
onPerDrain;

public const uint DefaultMaximumInboundRpcsPerConnectionPerDrain = 64;

public const uint UnsetMaximumInboundRpcsPerConnectionPerDrain = 0;
```

The default is 64. Setting the field to `UnsetMaximumInboundRpcsPerConnectionPerDrain` removes the bound entirely. Without a cap, a peer packing calls into a datagram costs the receiver a decode and a dispatch per call while costing the sender almost nothing.

Calls past the budget are dropped rather than dispatched, and the excess raises `RpcFloodViolation` once per drain:

```
public struct RpcFloodViolation : IViolation
{
    public uint ReceivedRpcCount;
    public uint AllowedRpcCount;
}
```

`ReceivedRpcCount` is how many calls the sender delivered in the drain; `AllowedRpcCount` is how many the receiver admits per drain. Subscribe through `ViolationManager.RpcFloodViolationDetected`.

Spawn compensation

A spawned object starts a round trip behind on an observer

An object spawned by the server does not exist on an observer until its spawn packet crosses the network and clears that peer's interpolation buffer. By the time the observer starts simulating it, the object is already a round trip behind where it should be. A projectile spawned moving is the clearest case: it appears at its origin point instead of somewhere along the path it has already travelled, and it stays visibly behind until something closes the gap.

Spawn compensation closes that gap by giving the object a burst of extra simulation time right after it starts, front-loaded so most of the catch-up happens in the opening ticks and the rest eases out smoothly. It is a Pro feature.

Enable compensation once the object has started

Call `NetworkSystem.EnableSpawnCompensation` once the object has started, typically from whatever the game runs when the object comes into being:

```
EnableSpawnCompensation(SpawnCompensationScope.All);
```

`aggression` controls how strongly the budget is front-loaded and defaults to `SpawnCompensationCurve.DefaultAggression` (2). Passing a value below `SpawnCompensationCurve.MinimumAggression` (1) clamps up to it; at 1 the budget is spent at a constant rate across the schedule instead of easing.

```
EnableSpawnCompensation(SpawnCompensationScope.All, aggression: 3f);
```

Calling it a second time restarts the schedule rather than adding to it. `CancelSpawnCompensation` abandons a catch-up still owed time, leaving the object wherever the schedule had reached:

```
CancelSpawnCompensation();
```

Each peer resolves its own budget from its own role and its own measured link, so the schedule itself never goes on the wire. The one figure that does is how much catch-up the server already spent, carried on the object's spawn so a remote client can add it to its own (see Bounds below). An object that never compensates sends no figure.

Choosing a scope

`SpawnCompensationScope` is a flags enum that says which peers run the catch-up:

- `None` — nobody compensates; the object starts its life at the tick it was built on.
- `Authority` — the server catches the object up through the latency a client's spawn *request* spent reaching it. Only a client-predicted spawn has lost any time getting to the server; the server's own spawns are on time by definition and this scope compensates nothing for them.
- `RemoteClients` — every client other than the one that predicted the spawn catches the object up through its own downstream latency.
- `All`: both hops compensate. They stack: a remote client is owed what the server already spent plus its own delivery delay, and the two together are held to the one cap.

Whichever scope is declared, the peer that predicted the spawn never compensates: it has been simulating the object since the tick it spawned it, so it is already exactly as far ahead as the budget would carry it.

The choice is really about who simulates the object. An object the server simulates and replicates the pose of wants `Authority` alone — a receiving client that fast-forwarded it would be dragged straight back by the next delta from the server, so the catch-up would only show as a stutter. An object every peer simulates for itself from the parameters it spawned with, the ordinary projectile, wants `All` — no correction is coming to fight it, and the receiving client's own downstream latency is real time the object owes.

Two members report the catch-up while it runs

`IsSpawnCompensating` is true while a schedule is still owed time on the object. `SpawnCompensationSeconds` is the extra time the current tick is spending on the catch-up, in seconds, and reads zero when nothing is compensating — read it beside the tick's own delta when the game simulates on a step of its own rather than subscribing to the event.

The more direct hook is `SpawnCompensationStepped`, raised once per tick while the schedule is running, immediately before that tick's early fixed-update callbacks:

```
SpawnCompensationStepped += OnSpawnCompensationStepped;

void OnSpawnCompensationStepped(in SpawnCompensationDelta delta)
{
    transform.position += velocity * delta.TotalDelta;
}
```

`SpawnCompensationDelta` carries the timing for that step: `FixedDelta` is the tick's ordinary fixed delta, `CompensationDelta` is the extra time this step is spending, and `TotalDelta` is the two added together — the figure to advance motion by so the handler needs no branch for whether a catch-up is running. `BudgetSeconds` and `RemainingSeconds` report the whole schedule and what is still owed after this step; `StepIndex` and `StepCount` (counting from one) report progress through it.

How much time a peer may be asked to make up is capped

How much a peer can be asked to compensate is capped, in milliseconds:

- `LagCompensationManager.DefaultMaximumSpawnCompensationMilliseconds` is 150, the default cap.
- `LagCompensationManager.MaximumSpawnCompensationCeilingMilliseconds` is 500, the highest the live setting can be raised to.
- `LagCompensationManager.MaximumSpawnCompensationMilliseconds` is the live setting; its setter clamps to the ceiling above.

```
CoreManager.LagCompensationManager.MaximumSpawnCompensationMilliseconds = 300;
```

The setting lives beside lag compensation's because the two are halves of one problem: lag compensation judges a shot against the past, and spawn compensation brings a freshly spawned object up to the present. In Unity it is **Max Spawn Compensation (ms)** on the Unity Lag Compensation Manager.

The cap bounds the whole of one peer's catch-up, counted across every hop the object travelled. Under `SpawnCompensationScope.All`, a remote client adds what the server already spent to its own delay and holds the sum to the cap, so an object that crossed two links is displaced no further than one that crossed a single slow link. Because a schedule is never longer than its budget, the cap also bounds how long a catch-up runs.

`NetworkSystem.MaximumTransmittedSpawnCompensationTicks` is 63 — the most ticks of already-spent server-side compensation the wire can carry when it tells a remote client how much of the object's age the server already accounted for. A budget larger than that is clamped to 63 ticks for

transmission, independent of the millisecond cap.

`aggression` shapes the curve rather than the budget: it never changes how much time is owed, only how quickly the schedule spends it.

Spawn compensation is not lag compensation

Spawn compensation does not rewind the world. It only advances a newly spawned object's own simulation faster for a short window after it starts; nothing else on the peer is touched, and no other object's state is altered or replayed. It is also not lag compensation, which judges a shot against where targets were on the shooter's screen when it fired, from a recorded hitbox history rather than by moving anything. The two are halves of one problem: lag compensation judges a shot against the past, and spawn compensation brings one freshly spawned object up to the present. Both are Pro, and the cap on spawn compensation lives on `LagCompensationManager` for that reason.

Binding spawns to your own engine

The seam

`SystemManager.SetSpawnHandler(INetworkSystemSpawnHandler)` registers the one object that correlates a spawned `NetworkSystem` with an object in your engine. Only systems that carry a `PlatformId` reach it: a full-serialize spawn whose `PlatformId` is set is routed to the handler, while a code-only system (`PlatformId` of `NetworkSystem.UnsetPlatformId`) never does and keeps the framework's default construction path. If you need to react to a code-only system starting, watch for it instead of expecting the spawn handler to fire.

Register the handler once, at bootstrap:

```
systemManager.SetSpawnHandler(mySpawnHandler);
```

Two object categories, two directions of control

`INetworkSystemSpawnHandler` has two members that mirror the two kinds of networked object, and they invert who builds the `NetworkSystem`:

- **Scene objects** already exist locally before any spawn arrives. The receiver builds the system itself, and the wire only has to bind onto it. `TryBindSceneObject` hands back that already-built system:

```
bool TryBindSceneObject(in SceneObjectKey sceneObjectKey, out NetworkSystem networkSystem)
;
```

A `false` result is an error at the call site: the scene was not loaded, or the scenes diverge. The returned system has been rented and initialized but not started; the framework assigns its network identifiers and applies the incoming state.

- **Dynamic prefabs** are the opposite: the framework builds and starts the system from its wire type identity first, and only then tells the handler, so the handler can instantiate the matching prefab and link it:

```
void OnDynamicSystemSpawned(NetworkSystem networkSystem, ushort prefabBundleId,
    uint prefabLocalId, uint groupId, uint sceneHandle);
```

`prefabBundleId` is the content shard the prefab ships in (zero is the base build's default shard); `prefabLocalId` is the prefab's local identifier within that shard, and is also the spawn's `PlatformId`. By the time this is called the system is fully started with its state already applied, so the handler is free to instantiate and link immediately.

SceneObjectKey

A scene object's identifier is only unique inside the scene file it was stamped in. Two live instances of the same scene, or two different scenes, can each hold an object stamped with the same id. `SceneObjectKey` pairs the two halves so a receiver holding several scenes at once binds a spawn to the right instance rather than just the right id:

```
public readonly struct SceneObjectKey : IEquatable<SceneObjectKey>
{
    public readonly uint SceneHandle;
    public readonly uint SceneObjectId;
}
```

`SceneHandle` is the live scene instance the object belongs to (or `NetworkSystem.UnsetSceneHandle`

when the peer is in whatever scene it booted into rather than one the server opened). `SceneObjectId` is the id stamped on the object, unique only inside its one scene. Both halves are required to look a scene object up correctly.

Group handling

`OnDynamicSystemSpawned` takes a `groupId`. Systems that share a non-`NetworkSystem.UnsetGroupId` group id are meant to link onto **one** engine instance rather than each getting their own: the first member to spawn instantiates the object, and every later member with the same `groupId` attaches to that same instance instead of instantiating a second one. Track the live instance per group id so a later member can find it.

The rest of the contract

- `OnSystemSceneChanged(NetworkSystem networkSystem, uint newSceneHandle)` — the system moved to a different server-opened scene; relocate its linked object into that scene rather than rebuilding it. Must be idempotent, since a redundant re-delivery can invoke it for a move already applied.
- `OnSystemParentChanged(NetworkSystem childNetworkSystem, NetworkSystem parentNetworkSystem)` — the system's parent changed; re-seat the linked object under the new parent's linked object, or back to the scene root when `parentNetworkSystem` is null. What the re-seat must preserve depends on the space the object's transform replicates in, which the handler reads from the object itself. Also idempotent.
- `OnSystemDespawned(NetworkSystem networkSystem, bool isObjectKept)` — the system stopped, on the server once its despawn has serialized to observers, or on a receiver when the server's despawn applies. `isObjectKept` is true when an interest condition stopped the system for this peer alone and the server wants the engine object retained rather than reclaimed: release the link, but keep the object and register it against `networkSystem`'s `Id` so a later re-entry spawn links onto it instead of building a second one. `false` is every ordinary despawn, where the handler releases every link it holds and destroys the object (unless the object's own destruction is what initiated the stop).
- `OnRetainedObjectReleased(uint systemId)` — the counterpart to a kept object: the server has now culled the system outright, or despawned it for everyone, so the object parked under `systemId` should be reclaimed exactly as an ordinary despawn would. Called for every despawn naming an id this peer no longer routes, so it must be a no-op when nothing is retained under that id.

Failure modes the handler owns

Two things the handler is responsible for getting right that the framework cannot catch for you:

- **A spawn for an unregistered bundle.** `OnDynamicSystemSpawned` can be called with a `prefabBundleId`/`prefabLocalId` pair your content registry has not loaded yet. Content must be loaded before spawns for it can arrive; treat an unresolved pair as an error rather than silently dropping the system.
- **Placing the instance in the named scene.** The handler must put the freshly built or moved object in the scene `sceneHandle` names, not in whatever scene the engine would place a new object into by default. A world holding several open scenes otherwise collapses every dynamic spawn into one scene.

Reference implementation

`Nucleus.Integrations.Unity.Managers.Systems.UnitySystemSpawnHandler` is the shipped Unity implementation of this interface and is the best worked example to read end to end. It keeps a dictionary of pending scene-object systems keyed by `SceneObjectKey`, a dictionary of live instances

keyed by group id, and a dictionary of retained instances keyed by system id; its `PlaceInScene` method is the concrete answer to the scene-placement failure mode above, and its unregistered-bundle check in `OnDynamicSystemSpawned` is the concrete answer to the other one. A non-Unity engine integration implements the same five methods against its own object model the same way.

Starting and stopping systems

Using Unity? See Spawning and despawning objects.

Starting a system

`SystemManager.EnsureStartSystem` has two overloads. The plain one is for a system with no engine object behind it — pure code:

```
bool started = coreManager.SystemManager.EnsureStartSystem(networkSystem);
```

The system must already be initialized. It carries `NetworkSystem.UnsetPlatformId`, so its header pays nothing for prefab, scene or parent fields it has no use for, and a receiver rebuilds it from its wire type identity alone. Find it on the receiving side through `SystemManager.SystemStarted` or a `NetworkSystemWatch` — there's no engine object for a script to have a reference to.

The fuller overload is for a system an integration is binding to an object it already owns:

```
bool started = coreManager.SystemManager.EnsureStartSystem(
    networkSystem,
    platformId,
    isSceneObject: false,
    prefabBundleId: 0,
    sceneHandle: NetworkSystem.UnsetSceneHandle);
```

- `platformId` correlates the system with an engine object for replication; a scene object's identifier within its scene, or a dynamic prefab's local identifier within its shard.
- `isSceneObject` is true when `platformId` identifies a scene object rather than a dynamically spawned prefab.
- `prefabBundleId` is the content shard's bundle identifier for a dynamically spawned prefab; zero is the base build's default shard, and scene objects carry none.
- `sceneHandle` is the live scene instance the system belongs to, so a receiver binds a scene object to the right instance and instantiates a dynamic prefab into the right scene; it defaults to unset, which is every world that opens no scenes.

Stopping a system

```
bool stopped = coreManager.SystemManager.EnsureStopSystem(networkSystem, isPoolReturnReque
stedOnDespawn: true);
```

An accepted stop transitions the system to `Stopping` immediately, serializes a despawn to observers, then completes: the system transitions to `Stopped`, leaves routing, releases its Id, and the registered spawn handler is notified. A wire-constructed system always returns to its pool once that happens; a locally rented system stays owned by the code that rented it unless `isPoolReturnRequestedOnDespawn` asks the framework to recycle it too.

Only the server stops a replicated system. A pure client's stop request is refused. A peer with no started transport (offline usage) can stop its own systems freely.

Shorthands

`NetworkSystemExtensions` gives both calls as instance methods that resolve the `CoreManager` from the system itself:

```
networkSystem.EnsureStart();
networkSystem.EnsureStop(isPoolReturnRequestedOnDespawn: true);
```

Both return false and log instead of throwing if the system is null or its `CoreManager` reference isn't set.

Deferred start and stop

A start or stop that lands mid-tick doesn't always ride that tick's stream.

A start called too late for the tick's state flush is deferred: the system stays fully usable to the caller (rented, initialized, components in place), but its Id stays unset and it goes live only after the next tick increments, when its spawn can ride that tick's serialization.

A stop called after the tick's state flush is held the same way: it can't ride the current tick's stream, so it's held to lead the *next* tick's packets, ahead of anything else in that tick, so receivers remove the system before applying anything new.

If a system is started again while a stop is still pending on it, the restart cancels the pending stop instead of both traveling — the system stays `Started` with its identity intact, and the queued despawn is withdrawn. If the despawn hadn't serialized yet, the cancel is wire-invisible; if it had already gone out this tick, every observing client is demoted to a full serialize, so the next tick's full re-spawns the system on the receivers that just removed it. This is what lets a spawn-then-despawn-then-spawn sequence in the same tick resolve to the latest intent rather than sending both a despawn and a respawn.

Reliable despawn

A despawn rides the unreliable state stream along with everything else. A peer that loses that one packet acknowledges straight past it and is left holding a ghost.

The server keeps a ledger of completed despawns (tick and system Id). When a peer's acknowledged tick reaches or passes a ledgered despawn it was never sent, the server re-delivers that despawn as a reliable `DespawnCleanup` message carrying the system's Id. The peer applies it immediately — cleanup isn't state-aligned, the despawn's tick has long passed, so the stop completes on receipt rather than waiting for interpolation. An Id the peer doesn't currently route is skipped silently, since the state-path despawn may have arrived after all.

The send is deliberately gated on the peer's acknowledgment reaching the despawn, never ahead of it: a cleanup sent before the acknowledgment catches up can overtake state for that system still in flight, and a peer that still held entries for a system a cleanup had already removed would read a divergence it didn't have. A joining peer is the one exception — it's swept through the whole ledger, since everything in it was despawned before that peer's stream began.

A stop that's deliberately silent (every peer performs it for itself, such as a scene release) is ledgered the same way but owes no `DespawnCleanup` — nothing is said on the wire for it.

Observing the roster from your own code

```
coreManager.SystemManager.SystemStarted += networkSystem =>
{
    // Raised once a system has completed its start on this peer, whether this
    // peer created it or a wire spawn built it.
};

coreManager.SystemManager.SystemStopped += networkSystem =>
{
    // Raised as a started system stops, immediately before a wire-constructed
    // system returns to its pool. Read what you need during the call.
};

int startedCount = coreManager.SystemManager.StartedSystemCount;

if (coreManager.SystemManager.TryGetSystemReference(id, out NetworkSystem networkSystem))
{
    // True either because the Id was found, or because id was the intentional
    // Unset sentinel — in which case networkSystem comes back null.
}

coreManager.SystemManager.UnknownSystemDeltaReceived += (connection, systemId) =>
{
    // A received delta referenced a system this peer doesn't have, so the rest
    // of that delta subpacket was abandoned. Means a missed spawn (the
    // unreliable channel can drop it) or a more serious desync.
};
```

`SystemStarted` and `SystemStopped` are the broad half of the discovery surface — a subscriber that wants every system and will sort them out itself. A subscriber that knows the exact composition it wants should register a system watch instead and be handed only those.

Moving a started system between scene instances

```
coreManager.SystemManager.EnsureMoveSystemToScene(networkSystem, destinationSceneHandle);
```

This is the one path that moves a spawned object between scenes — nothing in interest, physics, or scene loading calls it on its own. It's server-only, and only for a started, dynamically spawned system (not a scene object) that carries a platform identity. The move restamps the system's scene handle in place, preserving its Id, controller, access and group, then re-evaluates interest so a peer no longer in the destination scene is despawned and a peer newly in it is served a full. See the scenes category for how scene instances and handles work.

A spawn can carry one-time data

A spawn payload is a block of data attached to one object's spawn: a random seed, a loot table or a launch speed that the object needs when it arrives but that no member should keep replicating. Derive from `NetworkSpawnPayload`, write and read your own values, and declare an instance on the system before it spawns:

```
public class CrateSpawnPayload : NetworkSpawnPayload
{
    public int LootTableIndex;

    public override void WriteSpawnPayload(Writer writer, NetworkSystem networkSystem) =>
writer.WriteInt32(LootTableIndex);

    public override void ReadSpawnPayload(Reader reader, NetworkSystem networkSystem) => L
ootTableIndex = reader.ReadInt32();
}

if (NetworkSystemPool.TryRent<NetworkSystem, CrateComponent>(coreManager, out NetworkSyste
m networkSystem))
    networkSystem.SetSpawnPayload(new CrateSpawnPayload { LootTableIndex = 3 });
```

- The payload travels with every complete snapshot of the object, so a late joiner and a peer that regains interest receive it too, and it is never sent in the updates in between. Declare it in the same call stack as the rent, exactly as you write a component's opening values.
- A receiver reads it into an instance of its own, available as `networkSystem.SpawnPayload`. It is read after the object's member values, so `ReadSpawnPayload` can already see them and use them to decide what else to read. `ReadSpawnPayload` must read exactly what `WriteSpawnPayload` wrote, and it can run more than once for one object, so whatever it applies has to be harmless to repeat.
- Declare it on the peer that creates the object. That is the server, or, for a predicted spawn, the client predicting it: that client can create the object, set its payload and spawn it, and the server passes the payload on to everyone else. A client cannot change a payload once the object exists.
- The source generator registers every concrete payload type for you. A concrete payload type without a public parameterless constructor is a build error, because a receiver has to construct it.
- A system can carry a payload and nothing else. Rent one with no components through `NetworkSystemPool.TryRent<NetworkSystem>(coreManager, out NetworkSystem networkSystem)` and declare the payload on it.

A system can reach a peer a tick or two after it starts

A system may not start replicating to a given peer the instant it starts. Spawn admission is paced, so a peer can see the system a tick or two later than the tick it actually started on (see Interest Management).

System groups

Using Unity? See Several systems on one object.

Why groups exist

A networked object is often more than one `NetworkSystem`: a body, an inventory, a health pool. Each is spawned, replicated, and despawned independently unless something ties them together. A group does that tying: it marks several systems as belonging to the same object so a receiver waits for the whole set before treating the object as arrived, and the group's `Despawn()` stops every member together.

Membership is carried on the wire as a shared `GroupId`, read from `NetworkSystem.GroupId`. A system with no group carries `NetworkSystem.UnsetGroupId`. Nothing about a group is visible on an ungrouped system beyond that unset id.

NetworkSystemGroup

`NetworkSystemGroup` is the object you build a group with. It is not itself replicated; it is host-side bookkeeping that stamps a shared `GroupId` onto its members and spawns or despawns them together.

Initialize it before adding members:

```
NetworkSystemGroup group = ...;
group.Initialize(coreManager);
```

`Initialize` rents a fresh `GroupId` from the `SystemManager`. A client predicting a spawn under a group id it did not rent itself — because the server already leased one and told the client what it is — uses `InitializeAdopted` instead, which stores the given id rather than renting a new one:

```
group.InitializeAdopted(coreManager, groupId);
```

Add and remove members with `AddSystem` and `RemoveSystem`:

```
group.AddSystem(bodySystem);
group.AddSystem(healthSystem);
group.RemoveSystem(healthSystem);
```

Adding a system stamps it with the group's `GroupId` immediately, whether or not the group has spawned yet.

Spawning and despawning

`Spawn()` starts every member that is not already started. Call it again after adding more members and only the new ones start — this is how a group spawns some systems up front and others later onto the same networked instance:

```
group.Spawn();

// Later, once the player unlocks it:
group.AddSystem(inventorySystem);
group.Spawn();
```

`Spawn(networkSystem)` is the single-member shortcut: it adds the system to the group and starts it in one call.

```
group.Spawn(inventorySystem);
```

`Despawn()` stops every member the group holds. A group despawn always cascades to the whole set;

there is no per-system despawn through the group. Stopping one member directly with `SystemManager.EnsureStopSystem` stops only that member, and the rest of the group stays spawned.

```
group.Despawn();
```

Reading membership

`NetworkSystem.GroupMemberCount` reports how many systems the group holds. On the server it reads the group's live member count, so it grows as members are added. On a receiver, which builds no `NetworkSystemGroup` of its own, it reports the count carried on the full header the first member arrived with.

That received count is what lets a receiver wait for the whole set instead of acting the moment the first member shows up: it knows how many systems to expect for a given `GroupId` before it treats the object as fully arrived.

```
uint expected = someSystem.GroupMemberCount;
```

`GroupMemberCount` is zero for an ungrouped system.

Controller retention

`NetworkSystemGroup.ControllerRetentionPolicyOverride` sets what happens to every member's controlling identity when the controller leaves, for the whole group at once. It is resolved less specifically than a system's own `ControllerRetentionPolicyOverride`: a member's own override wins if set, then the group's, then the engine default. Setting it on the group is how the systems making up one object are kept from disagreeing with each other about retention.

```
group.ControllerRetentionPolicyOverride = ControllerRetentionPolicy.Retain;
```

What `Release` and `Retain` actually do is covered where controller retention itself is documented.

Group-wide behavior elsewhere

Two other parts of the engine treat a group as a single unit rather than repeating this page's material:

- Spawn pacing admits a group whole rather than splitting it across ticks, even past the per-tick spawn ceiling.
- Interest resolution folds each member's despawn and stop votes across the whole group, so one member out of range or one member still wanted is enough to decide the fate of every member.

INetworkSystemGroup

`INetworkSystemGroup` exists as a marker interface. As of this writing it declares no members — it names the concept of "a group" without yet defining a contract for it.

Parents, children and hierarchy

Using Unity? See Carrying and re-parenting objects.

A NetworkSystem can be attached to another so the two move together and replicate as a unit. The link is server-only, the child follows a strict eligibility rule, and a receiver sees the identifier before it can see the object it names.

Setting a parent

`SystemManager.EnsureSetSystemParent` is the only way a started system's parent changes:

```
bool isParentChanged = CoreManager.SystemManager.EnsureSetSystemParent(crateSystem, carrie
rSystem);
```

Passing `null` for the parent detaches the system and returns it to the root of its scene:

```
CoreManager.SystemManager.EnsureSetSystemParent(crateSystem, parentNetworkSystem: null);
```

The call only succeeds on the server. A client calling it is refused and logged, and nothing else in the engine reparents a started system.

Eligibility

Only a dynamically spawned `NetworkSystem` can be a child. A scene object's placement is authored and half its bind key, and a system with no platform identity has no engine object to seat anywhere, so both are rejected. Being a parent has no such restriction: any started system can hold children.

A proposed parent is also rejected when:

- it is the same system as the child,
- it is not started and registered,
- it belongs to a different `CoreManager`,
- it is in a different scene than the child, or
- attaching it would make the hierarchy circular or nest deeper than `SystemManager.MaximumParentDepth` (8).

The depth and cycle check walks up from the proposed parent toward the root, bounded by `MaximumParentDepth`, so it terminates even against a hierarchy an earlier bug already made circular.

How the link travels

An object spawned already carrying a parent has that parent's Id in its spawn header, so a new observer learns the relationship as part of the spawn itself. A later change made with `EnsureSetSystemParent` travels separately: it serializes as a reparent entry to every peer that already holds the object, carrying the system's Id and its new `ParentId`. A peer that already has the object re-seats it in place rather than treating the change as a new spawn.

The receiving side

On a receiving peer, `ParentId` can name a system that has not arrived yet, or that was written to the same tick after its child. Two members exist for that gap:

```
public uint ParentId { get; }
public bool IsParentResolved { get; }
public event ParentChangedHandler ParentChanged;
```

`IsParentResolved` is true once the system holds a live reference to the parent `ParentId` names; it is false both for an unparented system and for one whose parent identifier has arrived but the parent itself has not been resolved yet. `ParentChanged` fires on both ends, the server as it reparents and a client as it applies the change, and carries the Ids rather than the systems:

```
void OnParentChanged(uint previousParentId, uint currentParentId)
{
    // ...
}
```

Despawn and interest

Stopping a parented system cascades to its children, deepest first, so a receiver is never handed a despawn for a parent while a child still names it. Interest respects the same link: a system withholds its own subtree from a peer it has been culled for, so a child is never revealed to a peer that has not been given its parent.

Not supported yet

An object cannot be spawned directly into a parent; call `EnsureSetSystemParent` after the spawn instead. Hierarchies cannot cross scenes, a proposed parent in a different scene than the child is refused outright, so do not design around either.

What a system is

The replicated unit

A `NetworkSystem` is the thing Nucleus replicates. It is not a data container itself; the data lives in the `NetworkComponent` instances attached to it. A system can hold up to 31 components (`NetworkSystem.MaximumComponentCount`), and you read them back with `TryGetComponent<T0>` or check how many are attached with `ComponentCount`.

```
if (system.TryGetComponent<TransformComponent>(out TransformComponent transform))
{
    // transform is one of the components attached to this system.
}

int attached = system.ComponentCount;
```

Think of the system as the identity and lifecycle; the components are the fields that get serialized onto the wire.

Each component also has a component Id: the order it was added in, starting at zero. Every peer builds the system's components in the same order, so an Id names the same component everywhere. `TryGetComponent<T0>(out T0)` returns the first component of a type, which is all you need while each type appears once. A system can carry one type several times, though (a `NetworkTransform` replicating several parts does exactly that), and then you reach each instance by its Id:

```
for (byte componentId = 0; componentId < system.ComponentCount; componentId++)
{
    if (system.TryGetComponent(componentId, out TransformComponent part))
    {
        // part is the component added at position componentId, the same one on every peer
.
    }
}
```

Lifecycle

Every system moves through the same four states, exposed as `NetworkSystemState`:

- `Starting` — accepted but not yet live.
- `Started` — networked and usable.
- `Stopping` — a stop has been accepted but the tick has not ended yet.
- `Stopped` — not networked.

Read the current state through `NetworkSystem.State`, or use the shortcut `IsStarted`, which is true only while `State` is `Started`.

```
if (system.IsStarted)
{
    // safe to read replicated component data here.
}
```

A start accepted after the tick's state flush stays at `Starting` until the next tick begins; during that window `Id` is still unset.

Identifiers

A system carries several identifiers, and each answers a different question:

- **`Id`** — the network identity: which system this is, unique across the network. `UnsetId` is 0, and a system reads this as unset until it actually starts.
- **`PlatformId`** — which prefab, or which scene object, this system is linked to on the running platform. `UnsetPlatformId` is 0, which is also what a system with no engine object carries — a plain system never sets this.
- **`PrefabBundleId`** — the content bundle that owns the prefab named by `PlatformId`. Only meaningful for a dynamically spawned prefab; a scene object doesn't use it.
- **`GroupId`** — the `NetworkSystemGroup` this system belongs to, or unset if it isn't grouped.
- **`ParentId`** — the `Id` of the system this one is parented to (a crate riding a vehicle, a weapon in a hand), or `UnsetParentId` if it sits at the root of its scene.
- **`SceneHandle`** — the live scene instance the system belongs to, or `UnsetSceneHandle` if it belongs to whatever scene the peer booted into rather than one the server opened.
- **`IsSceneObject`** — true when `PlatformId` identifies a scene object that already exists on every peer, rather than a prefab a receiver must instantiate.

Role reads

Every system carries three reads for what role the local peer is playing: `IsServerStarted`, `IsClientStarted`, and `IsHostStarted`. These answer "is the server/client/host running here", not "do I control this object" — for control questions, use `IsController(ControllerType)` instead.

Systems and engine objects

A `NetworkSystem` does not need an engine object at all — a purely logical system (matchmaking state, a game rule tracker) can start, replicate, and stop without ever touching a GameObject or any other engine construct.

In an engine integration, one engine object can link to several systems at once, each with its own `Id` and lifecycle, all driven by the same underlying object.

Vocabulary

A `NetworkSystem` is a system, not a "networked GameObject." In the Unity integration, the marker component that links an engine object to its system is a separate thing — a small bridge, not the system itself.

Configuring a transport from launch arguments

Applying launch arguments

`Transport.ApplyCommandLineArguments()` reads the current process's parsed launch arguments and applies the flags a transport recognizes onto its `Configuration`. Nothing calls it for you; a build that wants its endpoint to come from the command line calls it itself, after adding the transport and before connecting.

```
Transport transport = await coreManager.TransportManager.AddTransportAsync<Synapse>();
transport.ApplyCommandLineArguments();

await transport.ConnectAsync(invoker);
```

An overload takes a `CommandLineArguments` instance directly, for arguments that did not come from this process's own command line, such as a launcher's own configuration:

```
CommandLineArguments launcherArguments = new(launcherProvidedArgs);
transport.ApplyCommandLineArguments(launcherArguments);
```

A flag that was not supplied leaves its setting at whatever the build configured. Applying arguments is safe to call even when the launcher passes none.

Apply arguments before connecting

Apply arguments before `ConnectAsync`. The socket reads `Configuration` (and, for Synapse, `RemoteHost`) as it builds; changing those values after the connection has started has no effect on that connection.

Flags that ship

Flag	Honored by	Sets
`port` (`Transport.PortArgument`)	Every transport	`Configuration.Port`
`address` (`Synapse.AddressArgument`)	`Synapse`	`RemoteHost`

`port` is read as a `ushort`. Every transport honors it because the base `Transport.OnApplyCommandLineArguments` applies it.

`address` is Synapse-specific. The server side of a Synapse transport binds every local address, so it reads only the port; the client side connects out to `RemoteHost`, so `address` sets that. Supplying an `address` value that does not parse as an IP address leaves `RemoteHost` unchanged and logs an error.

```
transport.ApplyCommandLineArguments(new(["Server.exe", "-port", "7777", "-address", "10.0.
0.5"]));
```

Adding your own flags

Override `OnApplyCommandLineArguments` on a transport to read additional flags. Call the base implementation first so the shared `port` flag still applies:

```
protected override void OnApplyCommandLineArguments(CommandLineArguments commandLineArgume
nts)
{
    base.OnApplyCommandLineArguments(commandLineArguments);

    if (commandLineArguments.TryGetValue(MyCustomArgument, out string value))
    {
        // apply value
    }
}
```

The parser

`CommandLineArguments` parses a `string[]` of launch arguments into flag/value pairs. `CommandLineArguments.Process` parses `Environment.GetCommandLineArgs()` on first use and caches the result for the process lifetime.

Supported forms:

- Separated: `-flag value` or `--flag value`
- Inline: `-flag=value` or `--flag=value`
- Single or double dashes, either form
- Flags match case-insensitively (`-PORT` and `-port` are the same flag)
- A dash immediately followed by anything other than a letter is read as a value, not a flag, so a negative number (`-5`) supplied as a value survives instead of being mistaken for a flag
- A token with nothing before it that a flag would claim (the executable path in argument zero) is ignored
- A flag with nothing after it (end of arguments, or the next token is itself a flag) is still recorded, with an empty value, so it reads as a switch

```
CommandLineArguments args = new(["Server.exe", "-port", "7777", "--address=10.0.0.5", "-he
adless", "-offset", "-5"]);
```

Reading values

`TryGetValue`, `TryGetUInt16`, and `TryGetUInt32` report failure rather than throwing, since launch arguments are user input:

```
if (commandLineArguments.TryGetUInt16("port", out ushort port))
    Configuration.Port = port;
```

`TryGetValue` returns false for a flag that was never supplied, and also for a flag supplied with no value (an empty string does not count as a value). `TryGetUInt16` and `TryGetUInt32` additionally return false when the supplied value does not parse as that integer type.

Use `Contains(flag)` to read a valueless switch, where the flag's presence is the information:

```
if (commandLineArguments.Contains("headless"))
{
    // run without a display
}
```

Connecting across networks

The three hops

Get a session working in this order. Each hop changes exactly one thing, so when it breaks you know which layer to blame.

Loopback, one process. Both peers run in the same process on `127.0.0.1`. This is the default: `RemoteHost` starts at `IPAddress.Loopback` in C#, and the Unity `SynapseTransport` component defaults its Remote Host field to `127.0.0.1`. Nothing about ports or addresses can be wrong here, so if this fails the bug is in your game logic, not the network.

LAN, two machines. Point the client's Remote Host at the server machine's LAN IP (e.g. `192.168.x.x`) instead of loopback. The server's own configuration does not change. This hop tests that the port is actually open on the server machine and that nothing between the two machines (a firewall) is dropping the traffic.

Open internet, peer outside your network. The client needs the server's public IP or DNS name. This hop adds a router in the path, which introduces port forwarding as a new failure point on top of the previous two.

What each side needs

The client needs the server's reachable address and the port it is listening on. The server needs that port open to it.

- Unity: set **Remote Host** on the `SynapseTransport` component to the server's address, and **Port** to match on both sides.
- C#: set `Synapse.RemoteHost` (an `IPAddress`) before connecting the client, and `Configuration.Port` the same on both peers.

See the Unity and C# transport pages for the full field lists.

Why the server ignores the address

`Synapse`'s server side always binds every local address on the machine; it never looks at `RemoteHost`, and it reads only `Configuration.Port`. That is why the server-side address field doesn't exist at all — there's nothing to set. A server listening on port 7777 answers on its loopback address, its LAN address, and its public address simultaneously, all on that same port. What has to be right is which of those addresses the client is told to dial, and whether traffic to that address and port actually reaches the machine.

The failure ladder

Work through these in order; each one only matters once the ones above it are ruled out.

Wrong port. The client's port doesn't match the server's `Configuration.Port`. Cheapest thing to check, check it first.

Firewall. The OS or network firewall on the server machine is dropping inbound packets to that port before the socket ever sees them.

Router with no forward. On a home/office network, the router won't route an inbound packet from the internet to the server machine's LAN address unless a port forward (or UPnP) sends it there. LAN-to-LAN traffic never hits this; it only shows up on the third hop.

Carrier-grade NAT. Some ISPs (and most mobile carriers) put the server behind a NAT you don't

control and can't forward through, no matter how correctly the router is configured. If forwarding is set up correctly and still doesn't work, this is usually why — and no amount of retrying fixes it from your end.

Reading which layer failed

`LocalConnectionState` alone can't tell you where it stopped. A `Synapse` client reports `Connected` as soon as it has sent its handshake, before the server has answered, and a link that fails for any reason ends at `Disconnected`; the shipped transports never set `Error` or `TimedOut`. The sign that the server actually answered and accepted the client is `ClientManager.LocalClientAuthenticated`, so read the two together:

- **Drops to `Disconnected` without `LocalClientAuthenticated` ever firing.** The client's packets aren't reaching the server at all. The client waits out `Configuration.ConnectedTimeoutSeconds` (15 seconds by default) and then drops. This is a routing/reachability problem: wrong address, wrong port, firewall, or missing port forward. A server that refuses the client drops it straight away instead, and `ClientManager.LastAuthenticationDenialReason` holds the reason it gave.
- **`LocalClientAuthenticated` fired, then the link dropped to `Disconnected`.** The initial handshake worked, so the path was open at least once, but traffic stopped flowing after that. This points at something intermittent: a NAT mapping that expired, a firewall rule that only blocks certain packet patterns, or a real network interruption. It's a different problem than never connecting at all, so don't go back to checking the port or the forward.

When to stop fighting NAT

If you've confirmed the port is right, the firewall is open, and the router has a working forward, and it still won't connect over the open internet, you're likely behind (or connecting to a peer behind) carrier-grade NAT or a similarly locked-down network you have no access to. At that point, stop trying to forward a port that can't be forwarded and use `RelayTransport` instead: one peer hosts a room and gets back a `RoomCode`, and every other peer joins with just that code — no address, no port, no forwarding required on either side.

No discovery, no punchthrough

Nucleus does not scan the LAN for servers and does not attempt NAT punchthrough on its own. An address always has to come from somewhere you provide — typed in, read from a config file, fetched from your own matchmaking service, or (for the relay path) a room code exchanged out of band.

Link states and connection events

Invoker: which role acted

Most connection operations happen for a specific role, and one process can run both roles at once on a host. `Invoker` names which one:

```
public enum Invoker : byte
{
    Client = 0,
    Server = 1,
}
```

Every state-change event and every connect/disconnect result carries an `Invoker` so a handler on a host build can tell which half of the process the change is about.

LocalConnectionState and RemoteConnectionState

`LocalConnectionState` describes a socket this peer owns - its own client connection, or (on a server) one of its client sockets:

```
public enum LocalConnectionState : byte
{
    Disconnected = 0,
    Connecting = 1,
    Connected = 2,
    Disconnecting = 3,
    Error = 4,
    TimedOut = 5,
}
```

`Disconnected` is the zero value, so a `Connection` nothing has touched yet reads as down rather than as some other state. The values are not written to the wire and nothing orders them numerically; the walk from `Disconnecting` to `Disconnected` doesn't mean less than `Connecting` to `Connected`, it just names the state passed through.

The shipped transports never set `Error` or `TimedOut`. A connect that fails and a link that drops later both end at `Disconnected`.

`RemoteConnectionState` describes what this peer believes about the other side of a link, and only has two values:

```
public enum RemoteConnectionState : byte
{
    Disconnected,
    Connected,
}
```

A `Connection`'s current values are read from its `LocalState` and `RemoteState` properties.

The state-change events

`TransportManager` raises two events, both carrying a `ConnectionStateChange`:

```
public event ConnectionStateChangedHandler? ConnectionLocalStateChanged;
public event ConnectionStateChangedHandler? ConnectionRemoteStateChanged;

public delegate void ConnectionStateChangedHandler(ConnectionStateChange connectionStateCh
ange);

public readonly struct ConnectionStateChange
{
    public readonly Invoker Invoker;
    public readonly Connection Connection;
    public readonly bool IsRemoteStateChange;
}
```

`ConnectionLocalStateChanged` fires when a `Connection.LocalState` changes - a socket this peer owns starting, connecting, or dropping. Check `Invoker.Client` alongside `LocalConnectionState.Connected` to see your own client socket come up, but that is not the same as the server accepting it: a `Synapse` client reaches `Connected` as soon as it has sent its handshake, before the server has answered. To know your own client has actually joined, use `ClientManager.LocalClientAuthenticated`, which fires once the server has accepted it and assigned its `Id`, or read `TransportManager.IsClientStarted`, which is only true once the client is both connected and authenticated.

`ConnectionRemoteStateChanged` fires when a `Connection.RemoteState` changes - this peer's belief about the far side of a link. On a server this is the event that answers "did a client arrive or leave". `ConnectionStateChange.IsRemoteStateChange` tells a shared handler which of the two events it's looking at without needing separate methods for each.

ConnectionStateChangeResult

Every connect and disconnect call returns one of these instead of throwing:

```
public enum ConnectionStateChangeResult : byte
{
    Success,
    AlreadyInState,
    InvalidConnection,
    InvalidSocketState,
    UnspecifiedError,
}
```

`Transport` exposes the calls that return it:

```
public abstract Task<ConnectionStateChangeResult> ConnectAsync(Invoker invoker);
public abstract Task<ConnectionStateChangeResult> DisconnectLocalConnectionAsync(Invoker i
nvoker);
public abstract Task<ConnectionStateChangeResult> DisconnectRemoteClientAsync(Connection c
onnection);
```

`AlreadyInState` and `InvalidConnection`/`InvalidSocketState` are not failures worth alarming over - they mean the call was redundant or arrived after the link was already gone. Treat `UnspecifiedError` as the one worth logging.

PeerConnectionDiscovered and PeerConnectionDropped

`TransportManager` also raises two peer-naming events, deliberately separate from the state-change events:

```
public event PeerConnectionChangedHandler? PeerConnectionDiscovered;
public event PeerConnectionChangedHandler? PeerConnectionDropped;

public delegate void PeerConnectionChangedHandler(Connection connection);
```

`ConnectionRemoteStateChanged` is subscribed by the observer and interest managers with no server gate, so raising it for every peer arrival would have a client walk its whole system set removing an observer and an interest entry that were never there. `PeerConnectionDiscovered`/`PeerConnectionDropped` exist for game code that wants to know about peers specifically, including on a host: a client is told about a peer by the server reporting it, and the `Connection` handed over is a stand-in - an identity with no link behind it (`Connection.IsPeerStandIn`). A host is told through the same events rather than through its own roster copy, so code watching this seam hears about peers the same way whichever kind of peer it runs on. `PeerConnectionDropped` fires while the entry is still held, so a handler gets one last chance to clear anything referencing it before the `Connection` returns to its pool.

Reading a link without holding it

Holding a `Connection` reference across time is unsafe: connections are pooled, and an entry returned to the pool is handed to the next peer that arrives. Read `LocalState`, `RemoteState`, and `Transport` while you have a live reference, but don't keep the reference itself around.

To remember a peer for longer than its link lasts, take its `ConnectionHandle` (`Connection.Handle`) instead of the `Connection`. A handle pairs the connection's `Id` with a `Generation`, so a handle taken from an earlier occupant of a recycled `Id` fails to resolve rather than silently naming whoever replaced it:

```
public bool TryGetConnection(ConnectionHandle connectionHandle, out Connection connection)
```

Resolve it fresh each time you need it - don't cache the `Connection` the resolve returns either.

`LocalConnectionStateExtensions` (declared as `ConnectionStateExtensions`) adds two checks on `LocalConnectionState`:

```
public static bool IsConnectedOrDisconnected(this LocalConnectionState localConnectionStat
e);
public static bool IsAnyDisconnected(this LocalConnectionState localConnectionState);
```

`IsConnectedOrDisconnected` is true only for `Connected` or `Disconnected` - the two resting states. `IsAnyDisconnected` is true for anything but `Connecting` and `Connected`, which covers `Disconnecting`, `Disconnected`, `Error`, and `TimedOut` in one check.

Running a dedicated .NET server

> **Using Unity?** See Building a headless Unity server.

There is no FishNet page for this, because every FishNet entry point begins in a scene. Nucleus does not require one: a dedicated server is a plain console application that references `Nucleus`, and nothing else.

The whole program

```
using System.Diagnostics;
using Nucleus.Connections;
using Nucleus.Managers.Core;
using Nucleus.Managers.NetworkLoop;
using Nucleus.Managers.Transports;
using Nucleus.Transports.Synapse;

// Name your own provider here so the default thread-pool timer never starts.
MainThreadStepProvider loop = new();
CoreManager coreManager = new(networkLoopStepProvider: loop);
TransportManager transportManager = coreManager.TransportManager;

// Nothing steps the loop until this thread does, so this bring-up cannot race it.
Synapse synapse = (Synapse)await transportManager.AddTransportAsync<Synapse>();
synapse.Configuration.Port = 7777;

await synapse.ConnectAsync(Invoker.Server);

// Ctrl+C and a container stop (SIGTERM raises ProcessExit) both end the loop below.
CancellationTokenSource stopSource = new();
ManualResetEventSlim loopExited = new();
Console.CancelKeyPress += (_, eventArgs) =>
{
    eventArgs.Cancel = true;
    stopSource.Cancel();
};
AppDomain.CurrentDomain.ProcessExit += (_, _) =>
{
    stopSource.Cancel();
    loopExited.Wait(TimeSpan.FromSeconds(5));
};

// Step the network loop on this thread until a stop is requested.
Stopwatch frameClock = Stopwatch.StartNew();
while (!stopSource.IsCancellationRequested)
{
    float frameDeltaMilliseconds = (float)frameClock.Elapsed.TotalMilliseconds;
    frameClock.Restart();

    loop.Step(frameDeltaMilliseconds);
    Thread.Sleep(1);
}

await synapse.ShutdownAsync();
coreManager.Deinitialize();
loopExited.Set();

/// <summary>Steps the network loop only when called, on the calling thread.</summary>
sealed class MainThreadStepProvider : INetworkLoopStepProvider
{
    private readonly NetworkLoopStepDriver _driver = new();

    public bool IsStarted { get; private set; }

    public bool IsTickFrame => _driver.IsTickFrame;

    public void Initialize(NetworkLoopManager networkLoopManager) => _driver.Initialize(ne
```

```
tworkLoopManager, 1000f / networkLoopManager.TickRate);

    public void Start() => IsStarted = true;

    public void Stop() => IsStarted = false;

    public void Return() { }

    public void Step(float frameDeltaMilliseconds)
    {
        if (!IsStarted)
            return;

        _driver.AdvanceEarly(frameDeltaMilliseconds, isVariableUpdateAllowed: true);
        _driver.AdvanceLate(frameDeltaMilliseconds, isVariableUpdateAllowed: true);
    }
}
```

`CoreManager`'s constructor builds every subsystem manager, then installs the provider you pass it as its last line. This provider steps the loop only when `Main` calls `Step`, so nothing runs against the managers while `Main` adds the transport and connects. `AddTransportAsync<T0>` instantiates the transport, registers it with the `TransportManager`, and initializes it. Setting `Configuration.Port` before `ConnectAsync` is what the server socket binds to; `ConnectAsync(Invoker.Server)` is what actually starts listening.

After that, `Main` is the loop. There is no `Update` method to hook and no scene to load: the `while` loop hands the provider the time since its previous pass, and the `NetworkLoopStepDriver` inside the provider decides when a tick runs and invokes the steps in order. See Driving the loop yourself for more on the driver.

Pass no provider and `new CoreManager()` starts a default one that steps the loop from a thread-pool timer before the constructor returns. Everything `Main` does after that, adding the transport and connecting included, then races a loop that is already running. See Threading and lifetime.

Referencing the engine

There is no NuGet package for Nucleus. Reference the three runtime DLLs, `Nucleus.dll`, `CodeBoost.dll`, and `SynapseSocket.dll`, as a set from one release, not independent picks. Any project that declares networked types also needs `Nucleus.CodeAnalysis.SourceGenerators.dll` from the same release as an `Analyzer` item, so the generated serializers your types need actually get produced. Adding Nucleus to a .NET project shows the project file.

`Nucleus.dll` targets `netstandard2.1`. That is the engine's own floor, not a suggestion for your host: target a modern framework such as `net8.0` or `net9.0` for your own executable rather than copying `netstandard2.1` onto it.

Routing engine logs

By default, Nucleus logs through `CodeBoost.Logging.LoggingService`, which starts out wired to a `ConsoleLogger`. In a container, `Console.WriteLine` output is usually fine as-is, but if you want logs routed anywhere else — a file, a structured sink, wherever your host reads them — implement `CodeBoost.Logging.ILogger` and call:

```
LoggingService.UseLogger(myLogger);
```

Call this before `new CoreManager()` so every manager's startup logging already goes through it.

Shutting down cleanly

Tear down transports before tearing down the manager:

```
await synapse.ShutdownAsync();
coreManager.Deinitialize();
```

`Transport.ShutdownAsync()` closes the socket; `CoreManager.Deinitialize()` stops the network loop and releases every manager it owns, in the reverse of the order it built them. `Deinitialize()` is safe to call more than once.

In a container, wire this to `SIGTERM` rather than relying on process exit, so an orchestrator's stop signal gives connected clients a clean disconnect instead of a dropped socket. The program above already does: .NET raises `ProcessExit` on `SIGTERM`, the handler only cancels `stopSource` and waits, and `Main` leaves its loop and runs the teardown on the thread that was stepping the loop. Don't call `ShutdownAsync` or `Deinitialize` from the handler itself, because that thread would tear the session down while `Main` is still stepping it.

The threading consequence

Every callback the engine raises (connection state changes, replication callbacks, anything you hook) runs on the thread that steps the loop. With the provider above that is `Main`'s thread, inside `loop.Step`, so code in the `while` loop can share state with those callbacks without locks. Only the two stop handlers run elsewhere, which is why they touch nothing but `stopSource` and `loopExited`.

`CoreManager`'s default network loop step provider is different: it drives the loop from a thread-pool timer, not from the thread that called `Main`, and every callback runs on that pool thread. A console host that accepts the default and touches shared state from both `Main` and an event handler needs to treat that state as cross-thread, the same as any other background-timer callback; there is no single "main thread" guarantee there the way a game engine's update loop gives you one.

Taking the port from the command line

Hard-coding a port works for local testing, but a real deployment wants it and the bind address configurable at launch. See Configuring a transport from launch arguments for how `Transport.ApplyCommandLineArguments()` reads flags like `port` into `Configuration.Port`.

Choosing a hosting topology

The three shapes

Nucleus does not pick a topology for you. A `CoreManager` can run its server half, its client half, or both, and that one choice is what separates a dedicated server, a player-hosted session, and a relayed room.

Dedicated server. You run a process with only the server half started. It has no gameplay of its own; every player connects as a client. You pay for the machine, and it keeps running whether or not any particular player is online.

Host. One player's process starts both halves. That player is also a client of their own server. Nobody else pays for a machine or keeps a process alive on your behalf; the session lives exactly as long as that player's process does.

Relayed room. A host, plus a `RelayTransport` in front of it. Neither the host nor any joining peer listens for inbound connections. Both sides are clients of the relay server, and the relay hands them to each other under a room code.

What each costs

Shape	Who runs a machine	Reachability	Session lifetime
Dedicated server	You	Server needs an open, reachable port	Independent of any one player
Host	The hosting player	Both server and client halves need an open, reachable port	Dies when the host's process does
Relayed room	The relay operator	Neither peer needs to be reachable	Dies when the host's process does, unless paired with a directory handover

A host is not a special mode

There is no host-only code path in the engine. A host is a `CoreManager` with a server started and a client started in the same process, talking to each other. `HostPairing` describes exactly this: two independent sockets, one process, each computed from the other's address. Under `HostPairing.Endpoint` the two halves exchange real datagrams over loopback like any other server/client pair; under `HostPairing.Local` they're paired in process without ever putting a packet on the wire. Either way, it's the same server and the same client code every other topology runs.

That has one direct consequence for trust: the host's `CoreManager` is the server. It runs the same server logic a dedicated server would, with the same say over what's true. Hosting doesn't relax that — it just means the server is sitting in a player's process instead of yours.

Every topology has its own reachability problem

A dedicated server needs an open, forwarded port people can reach. A player host needs the same thing on the player's own connection — behind carrier-grade NAT or a restrictive firewall, that often doesn't work.

A relayed room needs neither. `RelayTransport` carries the session through a Blitz Relay server: "nothing here listens. Both sides are clients of the relay, one holding the host role in a room and the rest joining it."

The host connects out to the relay, the relay names a room, and every joining peer connects out to the same relay and gives it the room code. No peer accepts inbound connections, so there's no port to forward and no NAT to fight.

The relay doesn't change who's in charge

Relaying usually gets blamed for handing control to whoever the relay trusts, or for adding a man-in-the-middle that could tamper with state. That doesn't happen here, because the relay only replaces the transport's addressing — it's still one peer's `CoreManager` acting as the server. The host authenticates and simulates exactly as it would over a direct connection; the relay just forwards its packets under a room code instead of a socket address. Nothing on the wire becomes client-authoritative by being relayed, and no other peer gains any capability it wouldn't have had directly against that same host.

Session lifetime

A plain host's session lives and dies with that player's process. Close the game, and the server half — and everyone's connection to it — goes with it.

What this repo ships for surviving that is a relay-plus-directory pairing: `ISessionHost` is the seam a directory implementation uses to publish where a session is hosted (`HostedCredential`) and to join wherever the directory currently points (`JoinAsync`). `NewfarmHostMigration` drives that against the directory on a poll loop, so a session can hand its hosting role from one peer to another without every client needing to be told a new address by hand. World-state retention across that handover (keeping the game state alive rather than starting the new host cold) is a Pro feature, and so is the pairing itself, because `NewfarmHostMigration` adopts the retained world when it promotes a peer. A broader, engine-level host migration feature is a separate, larger undertaking documented elsewhere and not yet built; this pairing is not that.

Trade-off axes

Weigh a topology on these, not on habit:

- **Cost.** A dedicated server is a machine you run and pay for continuously. A host costs you nothing directly — the player's machine and connection carry it. A relay costs whoever runs the relay server, continuously, whether or not it's you.
- **Trust.** All three have exactly one `CoreManager`, and it is the server. What differs is whose process it runs in, and whether you trust that process's operator (yourself, or a player) to run honest server logic.
- **Latency.** A dedicated server sits somewhere fixed, generally central. A host's latency is whatever every other peer's connection to that one player looks like — good for the host, variable for everyone else. A relay adds the extra hop through the relay server on top of the host's own connection.
- **Persistence.** A dedicated server's world persists independent of players. A plain host's world dies with the host's process. A relayed room with the directory pairing can hand the session to a new host, and Pro can carry the world state across that handover.
- **Who keeps a machine running.** A dedicated server asks you to. A host asks a player to, for as long as the session needs to exist. A relay asks whoever operates the relay to keep a lightweight forwarding service up, plus (for a survivable session) a directory.

What Nucleus does not provide

Nucleus does not run matchmaking, does not run a lobby service, and does not operate a hosted relay for you. `RelayTransport` and the directory seam (`ISessionHost`) are building blocks you point at infrastructure you run or a service you choose — the engine doesn't supply that infrastructure itself.

All three transports Nucleus ships — Synapse, Yak, and the Blitz Relay integration — are UDP. Pick a topology for its reachability and trust trade-offs, not for a transport-level constraint; none of the three shapes changes which transport you're allowed to use.

How transports work

What a transport does

A `Transport` moves bytes between two peers, and nothing else. It connects, disconnects, sends and receives datagrams, and tells the engine whether a given connection is its own client talking to its own server (`IsOwnClientConnection`). It does not know about identity, interest, reliability semantics, acknowledgment, or recovery — those all live in the engine above it, in `Connection` and the managers that use it.

That division is what lets you swap a transport without touching anything else. A game built against Synapse, Yak, or Blitz Relay replicates, spawns, and reconciles the same way regardless of which one is moving the bytes, because none of that logic ever reaches down into the transport.

Channels

Every send picks a `Channel`:

- `Channel.Reliable` — ordered and reliable.
- `Channel.Unreliable` — unreliable.

Nucleus is unreliable-first: state, RPCs, and messages default to `Channel.Unreliable` and only move to `Channel.Reliable` when something requires it.

What happens past the transmission unit differs by what is being sent:

- A reliable message or remote call that will not fit is handed to the transport whole, and the transport splits it.
- An unreliable message or remote call that will not fit is split by the engine into pieces that each fit one datagram, so it stays unreliable on every transport, the relay included. It arrives whole or not at all.
- A state write that will not fit is sent across several packets on whichever channel it was already using.

Maximum Transmission Unit

`Configuration.MaximumTransmissionUnit` is 1200 bytes by default — the ceiling on what the engine packs into one packet before it fragments or promotes.

The engine never gets the full 1200 bytes of payload to work with. Each transport reports `DatagramOverheadBytes`, the header it prepends to every datagram, and the engine reserves that (plus its own packet header) out of the configured unit before writing anything. So the number you set is a limit on the datagram, not a guarantee of payload space, and how much space you actually lose depends on the transport:

- Synapse reports its socket library's own header size as overhead.
- Blitz Relay reports more: the relay's own framing on top of the same underlying socket header, because the relay is a hop the packet passes through, not the endpoint reading it. A relayed session always has less usable payload per packet than a direct Synapse connection at the same MTU.

The three shipped transports

Transport	What it is	When to use it

Synapse	Real UDP sockets	Direct connections, dedicated servers, anywhere both peers (or the server) have a reachable address and port.
Yak	Offline, in-process, nothing bound	Local single-player or same-process testing where no socket should exist at all.
Blitz Relay	A third machine carries the traffic	Peers that can't reach each other directly — no port forwarding, both behind NAT — at the cost of relay overhead and a dependency on that third machine.

Adding transports

A `CoreManager` can hold several transports at once — add each with `TransportManager.AddTransportAsync` or `TryAddTransportAsync`. A transport is added and initialized before it is connected; `ConnectAsync` is a separate, later call.

Where to go next

Driving this from the Unity integration: see the transport component pages. Driving it from plain C#: see the API pages for `Transport`, `Configuration`, and `TransportManager`.

Transports, Connectivity and Hosting

Pages in this category:

- The Synapse transport
- TransportManager
- How transports work
- Running a dedicated .NET server
- Link states and connection events
- The Yak offline transport
- Choosing a hosting topology
- Configuring a transport from launch arguments
- Connecting across networks
- Packet transforms
- Simulating latency, jitter and loss
- The relay transport
- The Transport contract

Packet transforms

A transform rewrites every packet on its way in and out

A packet transform rewrites the bytes of every packet on its way to and from a Transport. Use it to layer encryption, compression, obfuscation, or a custom integrity check underneath the engine, without touching any send or receive call site.

This is a Nucleus-only, Pro feature. The interface, the shipped example, and the `TransportManager` hooks that apply it all live in `.Pro.cs` files and are absent from a Free build.

Implementing IPacketTransform

```
public interface IPacketTransform
{
    uint ReservedBytes { get; }

    bool TryTransform(PacketTransformDirection packetTransformDirection, Connection connec
tion, Channel channel, ReadOnlySpan<byte> source, Span<byte> destination, out int writtenL
ength);
}
```

`TryTransform` runs once per packet, in each direction:

- `PacketTransformDirection.Outbound` — the packet is leaving the engine, about to be handed to the Transport.
- `PacketTransformDirection.Inbound` — the packet has just arrived from the Transport and has not yet been read.

Read `source`, write the rewritten payload into `destination`, and report how many bytes were written through `writtenLength`. `destination` is always at least `source.Length` + `ReservedBytes` long, which is enough room for whatever the transform adds.

Return `true` when the rewrite succeeded and `writtenLength` is valid. Return `false` to discard the packet — this is how an inbound transform reports a failed integrity or authentication check. A discarded inbound payload raises `PacketTransformRejectedViolation`, with the `Channel` it arrived on and how many bytes it carried before the transform ran.

`TryTransform` is the hottest user-supplied callback in the engine. Keep it allocation-free.

Installing it

```
coreManager.TransportManager.PacketTransform = new MyPacketTransform();
```

Set `TransportManager.PacketTransform` before starting the Transport. Both peers must be running an equivalent transform, and both must be running it before either connects — a packet transformed by one side and read raw (or by a mismatched transform) on the other is exactly the case `PacketTransformRejectedViolation` exists for.

What ReservedBytes costs

`ReservedBytes` is the largest number of bytes the transform may add to an outbound payload. The engine deducts it from the Transport's `MaximumTransmissionUnit` to get the effective transmission unit it packs packets against, so a transformed packet still fits on the wire.
`TransportManager.GetMaximumTransmissionUnit` reports this reduced figure, not the Transport's

raw configured unit.

What a transform sees

A transform sees the whole of what the engine hands the Transport, including the engine's own packet header. It sees nothing the Transport adds around that: a handshake or a keep-alive the Transport sends on its own behalf is below this seam and keeps flowing untouched.

Key a transform on something both peers already hold

Key a transform on something both peers already hold — a build key, or something the Transport itself established — never on anything Nucleus negotiates. Nucleus authenticates over ordinary packets, and those cross this seam like any other, so a key that only exists once a session is authenticated has no way to exempt the traffic that authenticates it.

ExampleXorPacketTransform

`ExampleXorPacketTransform` is the shipped illustration of the shape a transform takes. It masks each payload with a four-byte XOR value, writing that mask ahead of the masked bytes so a fresh mask per packet keeps identical payloads from producing identical packets on the wire.

It provides no security of any kind and must never be used to protect real traffic — the mask ships beside the data it masks, so anyone reading the packet can undo it with no key. Treat it as a template for the length accounting and control flow a real transform needs (an authenticated cipher such as AES-GCM), not as something to ship.

In Unity

There is no Unity component for this. Set it the same way, through the `TransportManager` reached off `UnityCoreManager.CoreManager`, before starting:

```
unityCoreManager.CoreManager.TransportManager.PacketTransform = new MyPacketTransform();
```

That line is the whole of the Unity story.

The relay transport

Using Unity? See Blitz Relay Transport component.

RelayTransport

`RelayTransport` is a `SocketPairTransport<RelayServerSocket, RelayClientSocket>`: like every socket-pair transport it has a server socket and a client socket, but neither one listens. Both peers are outbound clients of a Blitz Relay server. One of them opens a room and holds it; the rest dial in with the room's name. That is what removes the port-forwarding problem — nobody needs an inbound port, a public IP, or a NAT hole, because nothing on either peer ever accepts a connection.

Member	Purpose
`RelayEndPoint`	The relay to carry this session. Defaults to loopback on `DefaultRelayPort` (7770).
`ConnectionKey`	The key the relay admits peers with. This is the relay operator's key, not a player's.
`RoomCode`	The room to join, and once the server connects, the room the relay named for it. Set it before connecting a client (a joining peer reads it); read it after connecting a server (the server's side writes it once the relay assigns a room).
`HostedRoomCode`	The room this peer is itself hosting, or empty when its own hosting side isn't up. Distinct from `RoomCode`, which for a peer that only joined answers with the room it was told to join, not one it hosts.
`MaximumClients`	How many clients the room this peer creates will hold. Defaults to `DefaultMaximumClients` (16). Used only when `ServerConfiguration.MaximumConnections` hasn't been set — a relayed room has to be given a real size when it's made, unlike a listening transport where "unset" can mean no limit.
`RelayHandshakeTimeoutMilliseconds`	How long to wait for the relay to answer a handshake before giving up. Defaults to 10000.

DatagramOverheadBytes

`RelayTransport.DatagramOverheadBytes` is the relay's own header on a host data frame (message type, virtual client id, channel) plus the overhead the socket library beneath the relay adds to every datagram. The engine reserves this before building a packet, so it comes straight off your usable transmission unit compared to a transport that talks to the wire directly — a relayed session pays for the relay's framing in addition to the socket layer's.

When the relay closes the room

If the relay closes the hosting peer's link, for example because the relay shut down or dropped the room, that peer's server stops on its own. On the next poll every client is disconnected and the server's local state walks down to stopped, the same as if you'd stopped it yourself, so the usual server-stopped and client-disconnected callbacks fire. You don't need to watch for a dead room. To keep playing, stand up a new room, or let `NewfarmHostMigration` do it.

Why the room needs a session-host seam

A relayed room belongs to whichever peer made it, and it dies with that peer. Every other peer's link to the room dies with it too. That means a handover isn't a reconnect — it's a brand-new room, under a name nobody could have known in advance, with no channel left between the survivors to learn it. `RelayTransport` alone has no way to solve that; it only knows how to open, join, and leave a room it's told about.

ISessionHost

`ISessionHost` is the seam a session-carrying service implements so `NewfarmHostMigration` can drive it without knowing what the service actually is:

Member	Role
`AdapterTag`	Names the service, filed with the directory alongside every credential this publishes, so a peer receiving a credential knows whether it can use it.
`HostedCredential`	What another peer needs to reach the session this peer is hosting, or empty when this peer isn't hosting one.
`IsHostingLinkUp`	Whether this peer's own hosting side is up.
`IsJoinedLinkUp`	Whether this peer's link to whoever is hosting the session is up. Independent of `IsHostingLinkUp` by design — a freshly promoted peer has one up and the other down.
`Poll()`	Drives whatever the service needs driving, once per `NewfarmHostMigration.Poll`.
`StartHostingAsync(uint timeoutMilliseconds)`	Stands the session up on this peer, returning a `SessionHostResult`.
`JoinAsync(string credential, uint timeoutMilliseconds)`	Joins the session at the given credential.
`StopHostingAsync()`	Stops hosting, leaving `HostedCredential` empty and `IsHostingLinkUp` false.
`LeaveAsync()`	Leaves a session this peer had joined, leaving `IsJoinedLinkUp` false.

`RelaySessionHost` is `RelayTransport`'s implementation of this: its `AdapterTag` is `"blitzrelay"`, `HostedCredential` reads `RelayTransport.HostedRoomCode`, and `StartHostingAsync`/`JoinAsync`/`StopHostingAsync`/`LeaveAsync` connect and disconnect the transport's server and client sockets. `SessionHostResult` is a two-way distinction, not a bool: `Success`, `CredentialUnreachable` (the service answered and the room isn't there or won't have this peer — the signal a directory acts on), and

`LocalFailure` (this peer's own side couldn't try, which is never reported to the directory).

NewfarmHostMigration

`NewfarmHostMigration` is the coordinator that keeps a session alive across a lost host: it registers the session with a newfarm directory, notices the host going, and either takes the session over itself or rejoins wherever the directory says it moved. It's constructed with a `CoreManager`, an `ISessionHost`, and a directory endpoint.

```
NewfarmHostMigration hostMigration = new(coreManager, sessionHost, directoryIpEndPoint);
```

Driving it:

- `StartHostingAsync(uint timeoutMilliseconds = DefaultTimeoutMilliseconds)` — opens a session with the directory and stands it up on this peer.
- `JoinAsync(ulong sessionId, uint timeoutMilliseconds = DefaultTimeoutMilliseconds)` — finds a session by id and joins wherever it currently lives. A join can end with this peer elected to host it, which still reports success.
- `TrySurrenderHosting()` gives the session up to whoever the directory elects next, while this peer stays in the session as an ordinary client afterward.
- `CloseSession()` — ends the session for everybody, rather than handing it on.
- `Poll()` — call once a frame or once a tick. Drives the directory link, the `ISessionHost`, and watches for the current host going quiet.

State and identity:

- `State` — a `NewfarmMigrationState` describing what this peer is currently doing about the session.
- `SessionId` — the id the directory issued for this session; what a host distributes to its clients so they can find their way back.

Events:

- `Promoting` — raised on the peer told to take the session over, before it does. The session is not hosted again until this returns.
- `Promoted(string credential)` — raised once this peer is hosting the session, carrying the credential the service named.
- `Rejoined(string credential)` — raised on a surviving peer once it has rejoined the session where it moved.
- `Abandoned(string reason)` — raised when the session could not be carried on, whether the directory refused it or no peer would host it.

The half that makes a handover mean anything, carrying the game world across it rather than starting the new host empty, is Pro. It lives in `ClientManager.Adoption.Pro.cs` and `SceneManager.Adoption.Pro.cs`: a client kept with `DisconnectResetMode.RetainReceivedWorld` retains the world it had when its link dropped, and the newly promoted server adopts that retained world as its own to serve. `NewfarmHostMigration` itself needs Pro as well: when it promotes a peer it calls `ServerManager.TryAdoptRetainedWorld`, which a free build does not have. The `Nucleus.Integrations.Newfarm` assembly therefore does not compile against a free `Nucleus.dll`, and neither does `Nucleus.Integrations.BlitzRelay`, which references it.

Assembly boundary

`RelayTransport` and `RelaySessionHost` live in their own assembly, `Nucleus.Integrations.BlitzRelay`, reached through the one `InternalsVisibleTo` grant the engine makes:

```
[assembly: InternalsVisibleTo("Nucleus.Integrations.BlitzRelay")]
```

A `Transport` is built on internal members of the engine's socket layer, so nothing outside the engine can write one without this grant. It's scoped to exactly this assembly — nothing else can reach those internals, and nothing in the engine itself names a relay, references one, or knows one exists. The relay server and the newfarm directory server are separate services; `RelayTransport` talks to the relay, `NewfarmHostMigration` talks to the directory, and neither runs as part of the other.

This handover is the relay's own, not an engine-wide feature

This is the relay-plus-directory handover this transport ships: a room that moves and a coordinator that finds the new one. It is not a general engine-level host migration feature — no such feature exists yet outside this relay/directory pairing.

Simulating latency, jitter and loss

Using Unity? See Simulating a bad connection in the editor.

A loopback connection is instant and lossless, which means anything that only matters on a bad link never gets exercised until players hit it. A transport can make loopback behave like a real connection, so you can drive that behavior on demand.

Describe the connection with NetworkConditions

`NetworkConditions` is a struct in `Nucleus.Transports`. Each field describes one way a real connection goes wrong, and the default value simulates nothing.

- `LatencyMilliseconds` is a one-way delay added to every packet sent.
- `JitterMilliseconds` is extra delay on top of the latency. Each packet gets a fresh random amount below this figure, so packets genuinely arrive at different times instead of all shifting by the same amount. A steady 100 ms link and one swinging between 20 ms and 180 ms average the same and feel nothing alike, and only jitter produces the second.
- `PacketLossChance` is the chance, from 0 to 1, that a packet is dropped. `0.2f` drops one packet in five.
- `OutOfOrderChance` is the chance, from 0 to 1, that a packet is held back long enough to arrive after packets sent later.
- `DuplicateChance` is the chance, from 0 to 1, that a packet is also sent a second time.
- `Seed` fixes the random choices behind the other fields, so a test can be run again with the same choices for the same packets. `0`, the default, leaves them different on every run. The same seed repeats the choices, but a networked run can still differ, because how many packets are sent and when depends on timing. To replay a failing run, pick a seed yourself, log it, and set it.

```
using Nucleus.Transports;

NetworkConditions networkConditions = new()
{
    LatencyMilliseconds = 100,
    JitterMilliseconds = 40,
    PacketLossChance = 0.1f,
};
```

Pass the conditions to the TransportManager

The `TransportManager` sets the conditions on every transport at once, or on every transport of one type.

```
// Every transport.
coreManager.TransportManager.SetNetworkConditions(networkConditions);

// Only Synapse transports.
coreManager.TransportManager.SetNetworkConditions<Synapse>(networkConditions);

// Pass null to go back to a clean connection.
coreManager.TransportManager.SetNetworkConditions(null);
```

If no transport of the type you name has been added, the call logs a message and does nothing.

To read the conditions back, ask for one type or for every transport. Asking for every transport gives you a

pooled list with one entry per transport, in the same order as `TransportManager.Transports`, so return it when you are done.

```
NetworkConditions? synapseConditions = coreManager.TransportManager.GetNetworkConditions<S
ynapse>();

List<NetworkConditions?> everyTransportsConditions = coreManager.TransportManager.GetNetwo
rkConditions();
// Read the entries here.
ListPool<NetworkConditions?>.Return(everyTransportsConditions);
```

A transport can also be set directly with `transport.SetNetworkConditions(networkConditions)`, and read with `transport.GetNetworkConditions()`.

Not every transport supports every condition

A transport that cannot simulate a bad connection logs a warning when you set conditions on it, and reads back null. A transport that can may still support only some of the fields.

`Synapse` supports everything except `DuplicateChance`, which it ignores. A packet it puts out of order is held back by up to 100 ms.

`Synapse` sends its connection setup packets, the handshake and the NAT traversal probes, without any simulated delay or loss, so a connection always opens however bad the conditions are. A connect that succeeds under simulated conditions does not show that the handshake copes with that link.

Set the conditions before Synapse connects

`Synapse` decides whether to simulate anything at all when it connects. Set the conditions after adding the transport and before calling `ConnectAsync`, and the connection starts out simulating them.

Once connected, a `Synapse` that started with conditions takes new values straight away. One that connected with no conditions ignores them until its next connection. If you want to change the conditions mid-session, start with a small one, such as a latency of 1 ms, so the simulation is running from the start.

Each CoreManager has its own conditions

Conditions belong to a transport, so a test process that runs a server `CoreManager` and a client `CoreManager` side by side sets each one on its own. Setting the server's conditions changes nothing about how the client sends.

Loss is per direction

Loss is applied by each peer to the packets it sends. A request and its reply each roll independently, so a round trip survives only when both directions do. With `PacketLossChance` = `0.2f` on both peers, a round trip's chance of surviving is lower than 80%, because either leg can be the one that is dropped.

Watch for the repair, not the breakage

The engine has its own answers to a lossy link: redundancy, targeted recovery, and a retention window. The point of testing under loss is not to watch things break but to confirm the repair actually runs. Watch for these two things:

- Check convergence: the peer should eventually hold exactly what the server holds, even with packets dropped along the way.
- Check recovery activity: the server should actually serve a repair. A test that asserts convergence without also asserting a recovery happened can pass on a clean run and prove nothing about the repair

path.

Read the result back from the Connection

`Connection` reports what actually happened on the link:

- `RoundTripTimeMilliseconds` is the smoothed round trip time.
- `RoundTripTimeDeviationMilliseconds` is how far samples typically stray from that average, which is the measured jitter.
- `PacketLossPercentage` is the connection's measured loss.

These are measured by the engine's own probes rather than echoed from the conditions, so they are a good check that the conditions are having the effect you expect before you trust the rest of the test.

The Synapse transport

Using Unity? See Synapse Transport component.

Synapse is Nucleus's production UDP transport, backed by SynapseSocket. It has no editor dependency; every setting is a plain C# property or field set before connecting.

Adding and connecting

Add the transport through `TransportManager.AddTransportAsync<T0>()`, then configure it before calling `ConnectAsync`:

```
Synapse synapse = (Synapse)await transportManager.AddTransportAsync<Synapse>();
synapse.RemoteHost = IPAddress.Loopback;

await synapse.ConnectAsync(Invoker.Server);
await synapse.ConnectAsync(Invoker.Client);
```

`RemoteHost` is the address the client half dials, and it defaults to `IPAddress.Loopback` so a single-process session works without setting anything. Override it before `ConnectAsync(Invoker.Client)` to reach a remote host; the server side always binds every local address, so `RemoteHost` has no effect on `Invoker.Server`.

A host process calls `ConnectAsync` with both invokers on the same `Synapse` instance, once for `Invoker.Server` and once for `Invoker.Client`.

Configuration

`Synapse.Configuration` is the shared `Configuration` struct:

Field	Default	Notes
`Port`	`0`	Server listen port and the port the client connects to. The engine's own default is `0`; the Unity component sets it to `7777`, but that default lives on the component, not here.
`MaximumTransmissionUnit`	`1200`	Packets beyond this length are split into multiple packets.
`ConnectedTimeoutSeconds`	`15`	How long a connected socket waits without remote data before the connection is considered timed out. One second is the floor, whatever value is set. Synapse also holds the heartbeat interval to at most a third of this value, so a link carrying nothing else is still heard from twice over before its peer gives up on it.

`ConnectingTimeoutSeconds`	`10`	How long a client waits for the server to answer its connect before giving up and returning to `Disconnected`. Zero falls back to `ConnectedTimeoutSeconds`. Only the client applies it, since a server answers every handshake as it arrives.

Cap how many clients the server accepts with `synapse.SetMaximumConnections(uint)`. The value lands in `ServerConfiguration.MaximumConnections`, which your code can read but not write, and `ServerConfiguration.UnsetMaximumConnections` (`0`, the default) accepts any number. Synapse applies the cap the next time the server starts. A client beyond it is refused during its handshake without an answer, so it gives up as it would on a server that never answered.

Finding out why a connection dropped

A connection reported `Disconnected` carries no reason. To learn why, subscribe to `ViolationObserved` on the `Synapse` transport. It is raised for every violation the socket layer reports, before the socket layer acts on it, so a timeout or a reliable message that could not be delivered arrives just before the connection is reported `Disconnected`.

```
using SynapseSocket.Core.Events;

Synapse synapse = (Synapse)await coreManager.TransportManager.AddTransportAsync<Synapse>()
;
synapse.ViolationObserved += OnViolationObserved;

void OnViolationObserved(Connection? connection, in ViolationEventArgs violationEventArgs)
{
    if (violationEventArgs.Reason is ViolationReason.Timeout or ViolationReason.ReliableEx
hausted)
    {
        // Record why this connection is about to drop.
    }
}
```

- `Reason` names the cause: `Timeout` when nothing was heard within `ConnectedTimeoutSeconds`, `ReliableExhausted` when a reliable message ran out of resends, `PeerDisconnect` when the other side said goodbye, and the security reasons when a peer sends something malformed, oversized or too fast.
- `Action` is the response the socket layer has already settled on. The event only reports it and cannot change it.
- On a server, `connection` is the remote client, still registered. On a client it is the server. It is null for a sender that never connected.

The event is raised on the network loop thread and allocates nothing. A handler may read what it is given, but must not block, stop the transport or disconnect anyone from inside it.

Starting and stopping

- `ConnectAsync(Invoker.Server)` starts listening for remote clients.
- `ConnectAsync(Invoker.Client)` connects the local client to `RemoteHost`.

- `DisconnectLocalConnectionAsync(Invoker)` disconnects a local connection (server or client) and closes its sockets.
- `DisconnectRemoteClientAsync(Connection)` disconnects a specific remote client from the server.
- `ShutdownAsync()` stops all local connections.

HostPairing

`HostPairing` controls how a `Synapse` transport recognizes its own client half when one process runs both server and client. It's read when the client socket is asked to connect, so it must be set before that.

- **`HostPairing.Endpoint`** (the engine default) — the two halves talk over the socket like any other pair. The server recognizes its own client by the source port of its datagrams, matched against the port the client socket bound. A host proves the same thing under this value that a remote client would prove: a real datagram, a real handshake, a real acknowledgment.
- **`HostPairing.Local`** — the two halves are paired in process and never exchange a datagram. Engaged only when the client is asked to dial a loopback address with its own server already up. Nothing about a host's own traffic crosses a socket, but a host also no longer proves anything about the transport with its own half.

`IsOwnClientConnection(Connection)` is how the transport answers "is this connection my own client half": under `Local` it's a reference check against the server socket's paired client connection, and under `Endpoint` it's the port match described above.

DatagramOverheadBytes

`Synapse.DatagramOverheadBytes` reports the widest header the socket layer puts on a datagram. The engine subtracts it when sizing its own segments, so each one stays inside a single datagram instead of being chopped again by the socket layer.

Other members

`SetPort(ushort)` sets both `Configuration.Port`, the port the server binds, and `RemotePort`, the port the client dials.

`IsLocalTransport()` returns `false`. Synapse always uses a real UDP socket, even under `HostPairing.Local`, where the traffic just never crosses one.

The Transport contract

What a Transport is

`Transport` is the abstract base every connection method implements against: a socket-backed transport like Synapse, an offline one like Yak, or anything else that can move bytes between peers. `TransportManager` drives it once per frame and never touches a socket directly.

The required members

A concrete `Transport` must implement all of these:

- `InitializeAsync(CoreManager coreManager, uint transportId)` - sets up the transport. The base assigns `CoreManager` and `Id`; an override calls base and does its own setup.
- `ConnectAsync(Invoker invoker)` - begins a connection using the current configuration. A client invoker connects the local client to the server; a server invoker starts listening.
- `OnSendPacket(OutgoingPacket outgoingPacket)` - the abstract method that actually transmits a packet. `SendPacket(OutgoingPacket)` is concrete: it runs the outbound packet transform, counts the transmitted bytes, and then calls `OnSendPacket`. Never override `SendPacket` itself; implement `OnSendPacket`.
- `PollForIncomingPackets(Invoker invoker)` - required by the base class, but the engine never calls it. Do any polling your transport needs inside `ReceivePackets`, or from a method of your own.
- `ReceivePackets(Invoker invoker, ref List<IncomingPacket> receivedPackets)` - the one receive call the engine makes, once per frame for each half of the transport that is running: the server while it listens, and the client while it connects or is connected. Add every packet that has arrived since the last call to the caller's list.
- `TryGetConnection(Invoker invoker, out Connection connection)` - returns the `Connection` for the socket belonging to an `Invoker`.
- `IsOwnClientConnection(Connection serverSideClientConnection)` - answers whether a `Connection` the server holds for a peer is in fact this transport's own client half. This one has no default and no warning: a transport that cannot tell must still return `false` rather than guess. Answering `false` costs a host the in-process delivery of its own traffic; a wrong `true` hands a stranger the server's own half.
- `DisconnectLocalConnectionAsync(Invoker invoker)` - disconnects a local connection and closes any related open sockets.
- `DisconnectRemoteClientAsync(Connection connection)` - disconnects a remote client from the server.
- `ShutdownAsync()` - stops all local connections and cleans up.

The virtuals

Two virtuals have a silent, working default:

- `DatagramOverheadBytes` returns `0`. Override it when the transport prepends its own header to every datagram, so the engine fills each segment up to the transmission unit less that overhead instead of chopping the result a second time.
- `OnApplyCommandLineArguments(CommandLineArguments commandLineArguments)` applies

`PortArgument` to `Configuration.Port`. A transport with endpoint settings of its own overrides it, calls base, and reads its own flags.

Two virtuals warn when left at their default, because there is no honest fallback behavior for them:

- `IsLocalTransport()` - whether the transport only runs locally, offline (several security checks are disabled while true). Logs a warning and returns `false` until overridden.
- `SetPort(ushort port)` - sets which port to use. Logs a warning until overridden.

SocketPairTransport: the shortcut for a server socket plus a client socket

Most transports have the same shape: one socket acting as the server, one acting as the client, each active per `Invoker`. `SocketPairTransport<TServerSocket, TClientSocket>` implements the whole `Transport` contract in terms of that shape, so a concrete transport supplies only its two socket types (both constrained to `CommonSocket`) and whatever behavior is genuinely specific to it.

It rents both sockets in `InitializeAsync`, calls `Initialize` on each, then calls the overridable `OnSocketsInitialized()` once both are ready - the hook for a transport that has to prepare its pair together. Every other contract member routes to whichever socket matches the calling `Invoker`, and `IsOwnClientConnection` is answered by reference identity against the transport's own client socket's `Connection`.

CommonSocket's helpers

`CommonSocket` is a utility base for the sockets a `SocketPairTransport` pairs up - useful, not required. Two of its helpers do the payload copying a socket implementation needs:

- `CopyIntoOwnedPayload(ArraySegment<byte> receivedPayload, out byte[] rentedArray)` copies a payload the underlying engine only lends for the duration of its receive callback into a buffer rented from `ArrayPool<byte>.Shared`, so the copy outlives the callback. The caller owns the rented array from there on.
- `TryCopyIntoIncomingPacket(OutgoingPacket outgoingPacket, out IncomingPacket incomingPacket)` copies an `OutgoingPacket`'s payload into a new `IncomingPacket`, renting its own array. It is the offline-transport pattern: see `Yak`.

The manager side a transport calls back into

`TransportManager` exposes the calls a transport implementation is expected to make:

- `RentInitializedConnection(CommonSocket commonSocket)` - rents and initializes a `Connection` for a socket that needs one.
- `ChangeLocalConnectionState(Invoker invoker, LocalConnectionState nextLocalConnectionState, Connection? connection)` - changes a connection's local state and invokes callbacks, or confirms it is already there.
- `SetRemoteConnectionStateAsConnected(uint idBySocket, Connection connection)` - server-only; marks a connection's remote state as connected and invokes callbacks.
- `SetRemoteConnectionStateAsDisconnected(Connection connection)` - server-only; marks a connection's remote state as disconnected and invokes callbacks.

The payload-ownership rule

A received payload is copied into a pooled buffer exactly once, and that buffer is returned by exactly one

owner. `CopyIntoOwnedPayload` and `TryCopyIntoIncomingPacket` are the two places that copying happens; whichever code queues the resulting `IncomingPacket` becomes the array's owner from that point on, and whatever path does not queue it must return the array to the pool itself. Two owners, or none, both corrupt the pool.

You cannot write a transport outside the Nucleus assembly today

You cannot write a `Transport` outside the Nucleus assembly today. `CommonSocket`'s `ConnectAsync`, `SendPacket`, and `ReceivePackets` are declared `internal`, and on `SocketPairTransport<TServerSocket, TClientSocket>` the `ServerSocket` and `ClientSocket` properties themselves are `internal` too - there is no public surface left to build a socket pair from outside the engine. `AssemblyInfo.cs` grants exactly one exception, `[assembly: InternalsVisibleTo("Nucleus.Integrations.BlitzRelay")]`, so that relay and rendezvous code can stay out of the engine rather than being folded into it to reach this machinery.

If you want to understand the shape of a real transport, read the source: `Yak` for the minimal offline case, `Synapse` for a full socket-backed one.

TransportManager

Using Unity? See Transport Manager component.

`TransportManager` is the `CoreManager` member that owns the added `Transport` instances, tracks every `Connection`, and answers role and peer-resolution queries. Reach it through `CoreManager.TransportManager`.

Adding a transport

`AddTransportAsync<T0>()` constructs a `T0 : Transport` with `Activator.CreateInstance<T0>()`, adds it, and initializes it, returning the new instance:

```
Transport transport = await coreManager.TransportManager.AddTransportAsync<MyTransport>();
```

`TryAddTransportAsync(Transport transport)` adds and initializes a transport instance you built yourself, returning `false` instead of adding it again if the same reference is already in `Transports`:

```
MyTransport transport = new(customConfig);
bool added = await coreManager.TransportManager.TryAddTransportAsync(transport);
```

Prefer `AddTransportAsync<T0>()` when a default-constructed transport is enough. Reach for `TryAddTransportAsync` only when the transport needs constructor arguments or other setup before it is added.

Removing a transport

`TryRemoveTransport(Transport transport)` removes and shuts down one transport reference, returning `false` if it was not found. `RemoveTransports<T0>()` removes and shuts down every added transport of type `T0`, returning the removed instances as a `List<Transport>`.

`Transports` is the read-only view (`IReadOnlyList<Transport>`) of everything currently added.

Role queries

- `IsServerStarted` — `true` when this peer's local server connection is `LocalConnectionState.Connected`.
- `IsClientStarted` — `true` when this peer's local client connection is `LocalConnectionState.Connected` **and** `IsAuthenticated`. A connected-but-unauthenticated client reads `false`.
- `IsHostStarted` — `IsServerStarted && IsClientStarted`.

Resolving peers

`TryGetConnection(uint id, out Connection connection)` resolves this peer's own `Connection` for an id. It checks the local client connection first (so a host's own client connection resolves the same way a pure client's does), then the roster of `Connections`. It returns `false` for `Connection.UnsetId` and for an id this peer was never told about.

`TryGetConnection(ConnectionHandle connectionHandle, out Connection connection)` resolves a `ConnectionHandle` instead of a bare id, and only succeeds while `connection.Generation` still matches `connectionHandle.Generation`. Ids and pooled `Connection` instances are both recycled, so a bare id can alias a different peer than the one you meant; a handle is what to hold when a peer needs to be remembered longer than its link lasts.

```
if (transportManager.TryGetConnection(storedHandle, out Connection connection))
{
    // connection is still the peer storedHandle was taken from.
}
```

`TryGetServerConnection(out Connection connection)` returns this peer's own server-socket connection, and `TryGetLocalClientConnection(out Connection connection)` returns this peer's own client connection — a pure server has none and returns `false`.

`ActiveConnections` is the read-only collection of every `Connection` currently known to this peer. What it holds depends on who is asking: on the server it is the connected remote clients; on a client it is peer stand-ins the server reported (never the local client itself, which `TryGetLocalClientConnection` answers for).

Channel and transmission unit

`DefaultChannel` is the `Channel` (`Channel.Unreliable` by default) the framework prefers when nothing more specific is asked for.

`GetMaximumTransmissionUnit(Transport transport)` returns how many bytes of that transport's configured transmission unit the engine may fill. It reports the transport's own configured unit less whatever a configured packet transform reserves, so code that packs against a transport's budget always asks here rather than reading the transport's configuration directly.

HostLoopbackDelivery

`HostLoopbackDelivery` governs when a host's server half delivers a message or call it addressed to its own client half (and vice versa), instead of that copy making a real round trip. It is only ever `HostLoopbackDelivery.Immediate` from game code — handlers run at the send site, before the send call returns. Setting it to anything else is refused, and logged, rather than silently ignored. `HostLoopbackDelivery.Deferred` exists so the engine's own tests can measure the alternative; it is not reachable from game code.

The transport-implementer side

A few members exist for a `Transport` implementation to drive the manager, not for game code to call directly: `RentInitializedConnection`, `ChangeLocalConnectionState`, `SetRemoteConnectionStateAsConnected`, and `SetRemoteConnectionStateAsDisconnected`. These are covered on the Transport contract page.

The Yak offline transport

`Yak` is a `SocketPairTransport<ServerSocket, ClientSocket>` whose two sockets are emulated rather than bound to anything. A client and a host exist in the same process, on the same `CoreManager`, with no socket opened and nothing sent over a wire. `Yak.IsLocalTransport()` returns `true`.

Adding and connecting it

```
Yak yak = new();

if (!await transportManager.TryAddTransportAsync(yak))
{
    // handle failure
}

_ = yak.ConnectAsync(Invoker.Server);
_ = yak.ConnectAsync(Invoker.Client);
```

Both `Invoker.Server` and `Invoker.Client` are connected on the same `Yak` instance. Nothing else changes about how the rest of the engine is used — systems start, spawn, and replicate exactly as they would on a real transport.

IsLocalTransport

`IsLocalTransport()` is read by exactly one place in the engine: the internal check that decides whether a Connection is a local peer. A Connection counts as one when it is emulated (`Connection.IsEmulated`), or when its Transport reports itself local and that Transport's own client half is connected, which is always the case on Yak.

That check in turn is read by both Pro and Free code. On Pro it gates spawn pacing (`InterestManager.SpawnPacing.Pro.cs`), so a peer running on Yak is never paced as if it were a remote client waiting on bandwidth. On Free it's read by `SceneInterestCondition.cs` and `SceneManager.cs`, where a local peer is exempted from the interest and scene-completion bookkeeping that only makes sense for a peer receiving state over a real link.

Swapping Yak for Synapse under a running CoreManager

A `CoreManager` that started offline on Yak can drop Yak and bring up Synapse in its place without restarting:

```
await coreManager.TransportManager.RemoveTransports<Yak>();

Synapse synapse = (Synapse)await coreManager.TransportManager.AddTransportAsync<Synapse>()
;
synapse.Configuration.Port = port;
synapse.RemoteHost = IPAddress.Loopback;

await synapse.ConnectAsync(Invoker.Server);
await synapse.ConnectAsync(Invoker.Client);
```

The world the peer was already running survives the swap: started systems keep their state and their ids, because the swap only replaces the transport underneath a server that was never anything but its own server. A second peer can then join over the real socket and receive that same retained world, as if the host had been on Synapse from the start.

When to use Yak

Reach for Yak for single-player, or for any test or tool that wants a real client-host pair — real replication, real spawning, real state flow — without opening a socket. It's also the natural way to start a session offline and only bring a real transport up once a player chooses to host.

Don't use Yak to reason about anything a socket actually does. It proves nothing about serialization timing, packet loss, MTU splitting, or latency over a real connection — those only show up once packets cross a real socket, which is what Synapse is for.

No Unity component

No `NetworkTransport` component ships for Yak. To use it from a Unity project, add it to the `CoreManager` from a script rather than from the Inspector.

Compression, Accuracy and Per-Type Registrations

Compression levels

`CompressionLevel` controls how tight a value's wire encoding is. Four values:

- **Aggressive** — the tightest windows. Tuned for small, well-predicted changes, such as Divine projection corrections. Costs efficiency on mid-range values.
- **Tight** — balanced windows that perform well across the common range of change magnitudes. This is the default a network member uses.
- **Loose** — wider windows suited to potentially large values, such as full (non-delta) writes.
- **Uncompressed** — full bit width, no compression.

`CompressionLevelExtensions.IsLooseOrUncompressed(this CompressionLevel)` returns true for `Loose` or `Uncompressed`.

Where it's set

Three places, and they don't share a default:

- `NetworkMember<T0>`'s constructor takes `deltaCompressionLevel`, defaulting to `CompressionLevel.Tight`. This is the level the member uses for its own delta writes.
- Every `Writer.Write*` method takes a `compressionLevel` parameter that defaults to `CompressionLevel.Loose`, not `Tight`. A direct call — `writer.WriteInt32(value)` — is Loose unless you pass a level explicitly.
- `Writer.DeltaCompressionLevel` is the stream-wide default the writer falls back to; it also defaults to `Tight`. A `NetworkMember` raises it to its own configured level around its delta write, then restores it afterward.

Pick the level per member based on what the value looks like: `Aggressive` for a quantity that mostly sits still and moves in small steps, `Loose` or `Uncompressed` for something that jumps around or needs exact width.

Wire accuracy

`NetworkMember<T0>`'s constructor also takes `accuracy` (`float.NaN` by default). NaN defers to the per-type default:

- `Writer.DefaultFloatingAccuracyAsSingle` — `1/1000`, the default for `float`/`double`/`decimal`.
- `Writer.MinimumFloatingAccuracyAsSingle` — `1/100000`, the floor any accuracy must respect.
- `Writer.DefaultQuaternionAccuracyAsSingle` — `1/1000`, the default for `Quaternion` members.

Set a coarser accuracy (e.g. `0.01f`) at the field declaration for quantities that tolerate more error than position — a velocity is the usual example. A per-call accuracy passed to `Initialize` still overrides the construction-time value.

Accuracy is also the change threshold

Accuracy doesn't only control how finely a value is quantized on the wire — it decides whether a delta write

reports a change at all. `Writer.TryWriteDeltaSingle` (and the double/decimal equivalents) compare the quantized difference between the previous and next value against `accuracy`, and only encode a delta when that difference exceeds it. A coarser accuracy means the member goes quiet more often: a velocity given `0.01f` stops reporting movement the wire would never resolve anyway, on top of packing what it does send into fewer bits.

DeltaCheckMode

Every delta write method (`TryWriteDeltaInt32`, `TryWriteDeltaSingle`, `TryWriteDeltaBoolean`, and so on) takes a `DeltaCheckMode`:

- **Checked** — data is only written if the value changed.
- **Unchecked** — data is always written, regardless of whether it changed.

`Checked` is what lets a delta write emit nothing on an unchanged tick. `Unchecked` forces the write every time, skipping the change comparison entirely — useful where the caller already knows a write belongs on the wire and the comparison would be wasted work.

Writing a Custom Serializer

When to write one

The source generator emits a full and delta serializer pair for every networked type it can see into. Write your own when you want a specific wire shape for a type instead: a packed encoding, a custom precision trade-off, or a type the generator cannot reach at all. A hand-written serializer always wins over a generated one for the same type.

There are two ways to register a serializer: the attribute route, resolved at compile time by the source generator, and the runtime route, called directly against the generic handler classes. Both end up calling the same `UseWriteHandler` / `UseReadHandler` methods; the attribute route is generated code that calls them for you.

The attribute route

Mark a static method with one of four attributes:

- `[DefaultWriter]` - the full writer for a type.
- `[DefaultReader]` - the full reader for a type.
- `[DefaultDeltaWriter]` - the delta writer for a type.
- `[DefaultDeltaReader]` - the delta reader for a type.

All four live in `Nucleus.Serializers` and target methods only. The generator checks each attributed method against a fixed signature shape: an extension method on `Writer` (writers) or `Reader` (readers), so the first parameter is `this Writer writer` or `this Reader reader`. A full writer takes the value and a `CompressionLevel`; a full reader returns the value and takes a `CompressionLevel`. A delta writer takes a previous value, a next value and a `DeltaCheckMode`, and returns `bool`; a delta reader takes a previous value and returns the next value.

```
[DefaultWriter]
public static void WriteMyType(this Writer writer, MyType value, CompressionLevel compress
ionLevel)
{
    // write value's fields through writer
}

[DefaultReader]
public static MyType ReadMyType(this Reader reader, CompressionLevel compressionLevel)
{
    // read fields back through reader and construct MyType
}

[DefaultDeltaWriter]
public static bool TryWriteDeltaMyType(this Writer writer, MyType previousValue, MyType ne
xtValue, DeltaCheckMode deltaCheckMode)
{
    // return true only if something was written
}

[DefaultDeltaReader]
public static MyType ReadDeltaMyType(this Reader reader, MyType previousValue)
{
    // mirror exactly what the delta writer chose to write
}
```

At generation time these become calls into `GenericFullWriter<T0>.UseWriteHandler`, `GenericFullReader<T0>.UseReadHandler`, `GenericDeltaWriter<T0>.UseWriteHandler` and `GenericDeltaReader<T0>.UseReadHandler`, registered as custom. A generated registration for the same type is never allowed to overwrite a hand-written one; the handler classes track this themselves (see below), so the order the two run in at assembly load does not matter.

The runtime route

The four attributes are convenience over the same generic classes you can call yourself:

- `GenericFullWriter<T0>.UseWriteHandler(WriteHandler, bool isSerializerCustom)`
- `GenericFullReader<T0>.UseReadHandler(ReadHandler, bool isSerializerCustom)`
- `GenericDeltaWriter<T0>.UseWriteHandler(WriteHandler, bool isSerializerCustom)`
- `GenericDeltaReader<T0>.UseReadHandler(ReadHandler, bool isSerializerCustom)`

Each also has an accuracy-aware overload (`WriteWithAccuracyHandler` / `ReadWithAccuracyHandler`, adding a `float accuracy` parameter) for types whose encoding takes a precision hint; a type that does not need one only registers the plain overload.

The delegate shapes:

```
// Nucleus.Serializers.GenericFullWriter<T0>
public delegate void WriteHandler(Writer writer, T0 value, CompressionLevel compressionLev
el);
public delegate void WriteWithAccuracyHandler(Writer writer, T0 value, CompressionLevel co
mpressionLevel, float accuracy);

// Nucleus.Serializers.GenericFullReader<T0>
public delegate T0 ReadHandler(Reader reader, CompressionLevel compressionLevel);
public delegate T0 ReadWithAccuracyHandler(Reader reader, CompressionLevel compressionLeve
l, float accuracy);

// Nucleus.Serializers.GenericDeltaWriter<T0>
public delegate bool WriteHandler(Writer writer, T0? previousValue, T0? nextValue, DeltaCh
eckMode deltaCheckMode);
public delegate bool WriteWithAccuracyHandler(Writer writer, T0? previousValue, T0? nextVa
lue, DeltaCheckMode deltaCheckMode, float accuracy);

// Nucleus.Serializers.GenericDeltaReader<T0>
public delegate T0 ReadHandler(Reader reader, T0 previousValue);
public delegate T0 ReadWithAccuracyHandler(Reader reader, T0 previousValue, float accuracy
);
```

`isSerializerCustom` is what decides precedence. Every `UseWriteHandler` / `UseReadHandler` overload checks its own `IsWriteCustom` / `IsReadCustom` (or the delta equivalents, `IsWriteSerializerCustom` / `IsReadSerializerCustom`) flag first: if a custom handler is already registered, the call is a no-op. Register your handler with `isSerializerCustom: true` and it locks the slot; anything the generator later tries to register for the same type is silently declined. Calling `UseWriteHandler` directly is the same operation the attribute route performs after code generation, so a plugin, a test, or code that runs before the generated assembly-load registration can install a handler exactly the same way:

```
GenericFullWriter<MyType>.UseWriteHandler(WriteMyType, isSerializerCustom: true);
GenericFullReader<MyType>.UseReadHandler(ReadMyType, isSerializerCustom: true);
GenericDeltaWriter<MyType>.UseWriteHandler(TryWriteDeltaMyType, isSerializerCustom: true);
GenericDeltaReader<MyType>.UseReadHandler(ReadDeltaMyType, isSerializerCustom: true);
```

Why the delta pair has to agree

A delta writer returns `bool`: it may write nothing at all when `DeltaCheckMode.Checked` finds no change to report. The reader has no independent way to know whether a value was written - it only knows what the writer decided. If the reader does not mirror the writer's decision exactly (reading when the writer wrote, and not reading when it didn't), the stream desynchronizes: every read from that point on pulls bits meant for something else, and the corruption goes undetected until values obviously stop making sense.

This is also why testing a delta serializer as a symmetric pair, not writer and reader in isolation, matters more here than almost anywhere else in the engine. "The writer doesn't throw" and "the reader doesn't throw" both pass for a reader that silently reads the wrong value.

Composing inside Nullable<T>

A member typed `MyType?` needs your full serializer to work under the engine's nullable handling rather than around it. `NullableEncapsulatedChecker<T0>` is how the engine itself detects a `Nullable<>` -encapsulated type: `GetIsNullable()` and `GetUnderlyingType()` return whether `T0` is `Nullable<>` and, if so, the underlying type.

`Writer` exposes `WriteNullable<T0>` and `TryWriteNullable<T0>` for the full and delta paths on any

`struct`. Write your full and delta writer/reader for the underlying value type as usual; `WriteNullable` / `TryWriteNullable` wrap it with the presence bit and dispatch to your registered handler for the underlying type. You do not write a separate serializer for the `Nullable<T0>` form - the underlying type's serializer is what composes.

Helpers to lean on

Two building blocks exist so you don't hand-roll the same encoding twice:

- **`ByteArrayDifferenceSerializer`** - `TryWrite(Writer, byte[] previousValue, byte[] nextValue)` and `Read(Reader, byte[] previousValue, byte[] result)` encode the per-byte difference between two same-length byte arrays: one flag bit per index, plus eight uncompressed bits when that byte changed. Both peers must already know the array length out of band; length is not part of the encoding. Reach for this inside a delta writer for any type that boils down to a fixed-length byte buffer.
- **`DecimalPacked`** - the struct the engine's own `decimal` writer and reader convert through (`Writer.Full.cs`, `Reader.Full.cs`) to pack a `decimal` more tightly than its raw bit representation, taking an accuracy hint via its `DecimalPacked(decimal value, float accuracy)` constructor and converting back with `AsDecimal(float accuracy)`. Use it wherever a type you're serializing embeds a `decimal` and you want the same packing rather than writing the four raw ints.

Known by-design refusals

Some framework types intentionally have no delta serializer. `Matrix4x4` (`Writer.TryWriteMatrix4X4`, `Reader.ReadDeltaMatrix4X4`), `Channel` (`Writer.TryWriteDeltaChannel`, `Reader.ReadDeltaChannel`), `NetworkSystemGroup` (`Writer.TryWriteDeltaNetworkSystemGroup`, `Reader.ReadDeltaNetworkSystemGroup`), and arrays through `Writer.TryWriteDeltaArrayAndCount<T0>` all throw rather than attempt an encoding. These are not hot-path types, and a silent best-effort delta encoding for them would be more likely to mis-encode than to save bandwidth. If your own type is in the same position - rarely sent, or without a meaningful "difference" - throwing from your delta writer is the correct behavior, not a gap to fill in.

Verifying it

`Nucleus.Tests/Serializers` is where round-trip and delta-pair symmetry coverage for the framework's own serializers lives; a custom serializer's own test should follow the same shape. Write through a real `Writer`, initialized with `Initialize(coreManager: null)`, and read back through a real `Reader` initialized from `writer.ToPackedBytes()` - never assert against the writer's return value alone, since "wrote successfully" and "wrote the right bytes" are different claims. For a delta pair, write a run of values across a single `Writer` and read the same run back across a single `Reader`, asserting each value in sequence; a bug that only desynchronizes the stream after the first mismatch will not show up on a single write/read pair.

Reaching Types You Cannot Annotate

The generator cannot reach a type in an assembly you do not control

The generator only emits a serializer for a type when it can find `[NetworkType]` on that type, or reach the type through a member it is already generating a serializer for. A type declared in an assembly you do not control, such as `System.Numerics.Vector2` or a type from a third-party math library, can never carry that attribute. Without another way in, it has no serializer.

The marker struct

The way in is a struct in your own assembly, carrying `[NetworkType]`, whose fields are the external types you need. Nothing about the struct's fields matters except their types:

```
[NetworkType]
public struct ExternalNetworkTypes
{
    public Vector4 Vector4;
    public Vector3 Vector3;
    public Vector2 Vector2;

    public Vector4Int Vector4Int;
    public Vector3Int32 Vector3Int32;
    public Vector2Int Vector2Int;

    public UnifiedColor UnifiedColor;
}
```

Every field is a marker, not a value. Nothing reads this struct at runtime and no instance of it is ever made. The generator walks the fields of a type carrying `[NetworkType]` and emits a serializer for each field's type, which is the only way to reach a type declared in an assembly nobody here can annotate.

The shipped marker

Nucleus ships one of these as `ExternalNetworkTypes` in `Nucleus.Serializers`. It covers `Vector4`, `Vector3`, and `Vector2` from `System.Numerics`, plus CodeBoost's `Vector4Int`, `Vector3Int32`, `Vector2Int`, and `UnifiedColor`. If your project only needs these seven types, you don't need a marker of your own.

Writing your own

For third-party types outside that list, declare your own marker struct in your own assembly and add one field per type you need a serializer for. The `[NetworkType]` attribute is not optional decoration: it is the only thing the generator reads to find the struct at all.

This was a real, recorded bug. `ExternalNetworkTypes` once carried a separate container attribute the generator never read, so every marker field in it was inert: the file read as the supported way to reach these types and did nothing at all. A type that also happened to be reached through a real networked member got its serializer anyway and hid the fault. That's what happened to `Vector3`: it got a working serializer only because `TransformComponent` already networks a `NetworkMember<Vector3>` for position and scale. `Vector2`, declared one line below `Vector3` in the same struct, had no other path in and silently had no serializer at all.

The lesson carries over directly: a marker struct that isn't actually carrying `[NetworkType]` fails silently, not loudly, unless one of its fields happens to be reachable some other way.

Finding out a type has no serializer

When a member's type has no serializer, the generator reports a `SERIALIZERS001` warning naming the member, once, instead of silently dropping it. See Generator Errors and Warnings for how to read generator diagnostics.

Seeing that warning for a type from an assembly you don't control is almost always a sign you need a marker struct, not a hand-written serializer: the type is a plain data type and the generator can serialize it field-by-field once it can see it. Reach for a hand-written serializer instead only when the type's shape genuinely needs custom read/write logic, not just visibility. See Writing a Custom Serializer for that case.

Generator Errors and Warnings

Diagnostics appear in your build output or the Unity console

The source generator runs as part of the normal C# compile. In a plain .NET project it reports through the ordinary build output or your IDE's error list, same as any other compiler diagnostic. In Unity, the generator DLL is imported with the `RoslynAnalyzer` label so Unity runs it as part of scripting compilation; its diagnostics land in the Unity console alongside regular compile errors.

Every diagnostic carries an id in the form `FAMILY###`, a severity (error or warning), and a message built from a plain format string rather than a fixed sentence per id. Reading the id tells you which stage raised it; reading the message tells you what to fix. Look up the page by the message text you actually see, not by the id alone.

The id families

Id	Severity	Stage
`NETWORKTYPEFINDER00`	Warning	Type discovery (`NetworkTypeFinder` walking your networked types and members)
`NETWORKTYPEFINDER01`	Error	Type discovery
`GENERATION000`	Error	Generation - invalid type
`GENERATION001`	Error	Generation - invalid Roslyn type
`GENERATION002`	Error	Generation - comparer/operation stage
`GENERATION003`	Warning	Generation - comparer/operation stage (declared, not currently raised)
`GENERATION004`	Warning	Generation - wire hash collision between two networked messages or calls
`SERIALIZERS000`	Error	Serializer generation
`SERIALIZERS001`	Warning	Serializer generation

`NETWORKTYPEFINDER*` fires first, while the generator is still deciding which types and members are networked at all. `GENERATION*` and `SERIALIZERS*` fire afterward, while it builds comparers, readers, and writers for what discovery found. `GENERATION004` is the exception: it is a late, non-fatal check that two networked messages (or two networked calls) hashed to the same wire id. It only reports; it never stops the rest of the assembly from generating, because renaming one of the colliding types is a fix to make, not a reason to withhold everything else the build would otherwise emit. While two types collide, the engine sends neither, registers neither, and hands neither to a handler.

The commonest cause: a member with no serializer

By far the most common diagnostic is `SERIALIZERS001`, with a message shaped like:

`[Type.Member] Type [SomeType] for [Write/Read] SerializeMethod could not be found; this value will not be networked.`

This means the generator could not find a serializer for that member's type. It has two fixes:

- **Marker struct** - if the type lives in an assembly you don't control (a `System.Numerics` type, a third-party math type), add it as a field on a marker struct carrying `[NetworkType]` so the generator can reach it. See Reaching Types You Cannot Annotate.
- **Hand-written serializer** - if the type's shape genuinely needs custom read/write logic rather than just visibility. See Writing a Custom Serializer.

A `UnityEngine` type (`Vector3`, `Quaternion`, and so on) does not raise this warning: the generator reaches the type's public fields and writes a plain serializer for it. Declare the `System.Numerics` equivalent anyway. See Unity Types and System.Numerics.

`SERIALIZERS000` is the error-level sibling of the same stage: a required serializer is missing or malformed in a way the generator cannot route around at all, for example `A [Write/Read] serializer could not be found for [Type].` or a type exceeding the maximum networked member count. Treat it the same way as `SERIALIZERS001`, just with a build-breaking severity.

The second commonest: a type the generator can't fill in

`NETWORKTYPEFINDER01` covers a family of structural problems in how a networked type is declared, including type arguments that aren't currently supported and a member type discovery doesn't recognize. The one worth calling out specifically: a networked component or system that is not declared `partial`, or that is nested inside another type in a way the generator cannot reach, fails as an ordinary Roslyn compile error (for example a missing-partial-modifier error) rather than a Nucleus-specific diagnostic, because the generator is trying to emit a partial declaration into a type it does not control. If your networked type is producing raw C# compiler errors about a generated file rather than a `NETWORKTYPEFINDER`/`GENERATION`/`SERIALIZERS` id, check that the type and every type containing it is `partial` and check its nesting.

Read the top diagnostic, not the longest list

The generator's stages run in order, and an early error aborts the run for that type before later stages get a chance to run. A discovery-stage error on `NETWORKTYPEFINDER01` commonly produces a long tail of `SERIALIZERS001` warnings underneath it, because every member the generator never got to walk correctly also has no serializer. Fixing the first error in the list is usually enough to make the rest disappear on the next build. Scroll to the top of the diagnostics for a type, fix that one, and rebuild before working through the rest.

A generator diagnostic is a compile-time problem, never a runtime one

A generator diagnostic is a compile-time problem: the generator could not produce correct code for your types. It has nothing to do with runtime behavior. If your build is clean but replicated values are wrong, systems never converge, or packets look malformed at runtime, that is a wire or convergence problem, not a generator diagnostic, and belongs on a different page.

How a Value Reaches the Wire

> New to Nucleus? Start with How replication works first — this page assumes you already have the pipeline overview and goes one level deeper, into the encoding itself.

Setup: Reference the source generator (plain .NET) or Installing the Unity integration (Unity).

The path, end to end

A `NetworkMember<T0>`'s `Value` setter is the start of the path. It mutates the member's live ring slot and notifies the owning system that something changed — nothing is written yet.

At serialize time, the component's generated writer encodes each changed member against the shared baseline both peers hold, producing a delta payload. That payload becomes the body of a framed subpacket: a `StatePacketType` (`Delta`, `Full`, `Controller`, and the rest), the subpacket's bit length, then the body. Multiple kinds get packed into one combined stream for the tick this way — a spawn, a controller change, and a batch of deltas can all ride together.

`PacketSegmentWriter` takes that combined stream and fits it into one or more `StatePackets`, splitting at the maximum transmission unit when the tick's data doesn't fit in one. Each `StatePacket` carries a `SegmentNumber` and `ExpectedSegmentCount` so the receiver can rejoin a tick's segments before demuxing the frames inside. The segmented packets are then handed to the transport for the peer.

On the receiving end this runs in reverse: rejoin segments into one reader, walk the frames in order (`Full`, then `Controller`, then `Delta` — so a spawn exists before anything addresses it), and each component's generated reader decodes its members back out.

Values are packed at bit granularity, not byte granularity

Values are packed at bit granularity, not byte granularity. A member carries a length indicator alongside its data, so the wire cost of "a float" or "a Vector3" has no fixed answer — it depends on how far the value moved since the last tick it was sent. A value that barely changed writes short; one that changed a lot writes long. (How that length indicator itself is encoded isn't covered here — the granularity and the cost trade-off are what matter at this level.)

Every serializer is generated at build time, never by reflection

Every `Write`/`Read` method your components use is emitted by the Nucleus source generator at build time, and each one registers itself at assembly load through a module initializer. There is no reflection-based serialization path, and no runtime type inspection to find or build a serializer — a type either has generated code for it or it doesn't compile against the network path. That's what keeps the engine AOT-friendly: nothing here depends on `System.Reflection.Emit` or similar runtime code generation that AOT platforms disallow.

Three things you can tune

- **Compression level and accuracy — how tightly.** `CompressionLevel` (`Tight` by default, or `Aggressive`) is fixed at a `NetworkMember<T0>`'s construction. `Accuracy` sets the wire accuracy used for delta encoding, loss recovery, and lossy serialization; it defaults per-type (tighter for a `Quaternion` than a general float) and can be coarsened for a value that tolerates more error, like a velocity.
- **Transmission mode and send interval — how often, or whether at all.** `TransmissionMode.Interval` (the default) sends ordinary deltas paced to a `SendInterval`;

`SendInterval.Normal` sends every changed tick, and a wider interval accumulates changes and rides at most one delta per span. `TransmissionMode.Divine` projects the value instead of sending ordinary deltas, correcting only when needed, so it can go quiet for stretches where `Interval` would still be sending — the mechanism itself isn't something you need to reason about to use it, and it is Pro-only.

- **`[ReplicationIgnore]` — whether the code exists at all.** A member declared with `ReplicationIgnoreAttribute` gets no generated serialization: the generator skips it entirely rather than serializing-and-discarding it, so it's not a runtime cost you pay and choose not to use — it's code that was never emitted.

Most state rides an unreliable channel by design

Most state rides an unreliable channel. Correctness doesn't come from retransmitting a lost packet — it comes from acknowledgment, redundant resends, and targeted recovery for a peer that's fallen behind. See the reliability and recovery section for how that works; this page only needs you to know that "unreliable" is the normal case for state, not an edge case to work around.

Glossary

- **Full** — a complete serialization of every member, used for spawns and resyncs. Reconstructs a system from nothing.
- **Delta** — a member encoded against the shared baseline, carrying only what changed.
- **Baseline** — the last value both peers agree the member held, which a delta is encoded relative to.
- **Framed subpacket** — a `StatePacketType`, its bit length, then its body; the unit the combined per-tick stream is built from.
- **Segment** — one `StatePacket` produced by splitting the tick's combined stream to fit the maximum transmission unit.
- **Bundle id** — part of how a spawned system's type is addressed on the wire; not explained further here.
- **Projection** — the mechanism behind `TransmissionMode.Divine`: a value rides a predicted path between corrections instead of being resent every interval.

Serialization, Codegen and the Wire

Pages in this category:

- Networked Type Attributes
- Compression, Accuracy and Per-Type Registrations
- How a Value Reaches the Wire
- Type Identity on the Wire: Bundles and the Registry
- What the Generator Emits
- Wiring the Source Generator into a .NET Project
- Generator Errors and Warnings
- The Shapes on the Wire
- Writer and Reader
- Reaching Types You Cannot Annotate
- Writing a Custom Serializer

Networked Type Attributes

The attributes

The generator reads a small set of attributes to decide what to serialize and what to leave alone. Most code needs none of them — a type reached through a `NetworkMember<T0>` field is walked automatically, and a member on a `NetworkComponent` is networked by default.

[NetworkType]

Applies to a class, struct, or enum. Forces the generator to create serializers for that type even though nothing in the assembly reaches it through a `NetworkMember<T0>` field.

You will almost never need this. It exists for a type the generator has no other way to discover — one that is only ever boxed, reflected into, or otherwise referenced outside the normal member graph. Anything a `NetworkMember<T0>` already walks to gets its serializers generated automatically and does not need the attribute.

[NetworkIgnore]

Applies to a field or property. Excludes that member from generation entirely — the generator does not walk it, does not include it in serialization, and produces no code for it.

[NetworkBundle(ushort)]

Assembly-level. Names the bundle identifier half of every type identity the assembly declares:

```
[assembly: NetworkBundle(1)]
```

A type's wire identity is the tuple of its bundle id and a per-assembly local id, so bundle ids only need to be unique across the assemblies present in a build. The always-present main bundle defaults to `0` when the attribute is absent; every other bundle should declare its own unique id exactly once. See the type identity page for how the local id side of that tuple is assigned.

[ReplicationIgnore]

Applies to a `NetworkMemberBase` field, or to a whole `NetworkComponent` class (in which case it excludes every member the component declares). Keeps the value off the wire entirely while leaving how it is saved alone — the inverse of `[WorldStateIgnore]`.

It is an attribute rather than a constructor argument, and the reason is performance. A constructor argument is a runtime value, so honoring one would mean the generator still emitting every member's serialize and deserialize lines, each testing a flag at runtime on every peer, for the life of the build. An attribute is answered while code is being generated, so the member's serialize lines are never emitted at all — there is no branch to predict and no instruction to retire, because there is no code. A tick on which only marked members changed produces no packet.

Two consequences follow from that:

- **The value is the constructed default on every peer but the server.** Nothing ever arrives to populate it, so reading one on a client returns the type's default rather than what the server holds.
- **A peer promoted to server by adoption never received these values.** It holds the defaults rather than what the previous server had, and saving from it writes those defaults over whatever the store already held. Server-only state does not survive a handover.

Custom serializer attributes

`[DefaultWriter]`, `[DefaultReader]`, `[DefaultDeltaWriter]`, and `[DefaultDeltaReader]` mark a method as the default (delta) writer or reader for a type, and all four derive from the abstract `SerializerAttribute` base. These are for hand-written serialization, covered on the custom serializer page.

Attributes that do not exist

`[NetworkTypeContainer]`, `[NetworkAccuracy(float)]`, `[NetworkSerializeMode(CompressionLevel)]`, and `[NetworkExcludeCount]` have no declaration anywhere in this tree. Accuracy and compression level are not attributes — they are constructor arguments on `NetworkMember<T0>` itself, for example `new NetworkMember<float>(CompressionLevel.Aggressive, accuracy: 0.01f)`.

Persistence

`[WorldStateIgnore]` is the persistence-side inverse of `[ReplicationIgnore]` — it keeps a member out of a saved world while leaving replication alone. It's covered in the persistence section.

The Shapes on the Wire

Packet types

Every packet the transport hands off carries a `PacketType` in its header.

- **Message** — one-shot single-packet messages, deserialized immediately.
- **State** — tick-aligned data that may span several segments.
- **Reconcile** — correction data sent to clients; only ever received on a client, which replays forward from that point after applying the correction.
- **Raw** *(DEBUG builds only)* — raw data with no special formatting.
- **Rpc** — system-addressed remote calls, deserialized as they arrive rather than on a tick boundary.

`Raw` exists only under `#if DEBUG` and sits at value 3. `Rpc` is declared past it and held at value 4 whether or not `Raw` is compiled in, so a debug peer and a release peer read the same header regardless of build configuration.

State subpacket types

A `State` packet's body is not one thing — it is a sequence of framed subpackets, each tagged with a `StatePacketType`. Seventeen kinds exist:

Kind	Carries
`Delta`	Delta-serialized system state: members encoded against their shared baseline, for a change the sender made itself. The default, cheapest kind — a world that never grants write access emits only this one.
`Full`	Full-serialized system state, used for spawns and resyncs.
`Controller`	Controller changes, as a list of `(SystemId, ControllerConnectionId, ControllerRetentionToken)` entries.
`ClientState`	Tick-aligned client state: upstream from the controlling client to the server, and from the server to the system's observers, so an observer predicting the system can simulate from it.
`Reconcile`	Server?controller state correction carrying the server's full state for a system at the current tick.
`Ack`	A standalone acknowledgment carrying the receiver's highest fully-applied tick that carried state. Written only when that value has advanced and there's no other outbound traffic to piggyback it on.

`Recovery`	Targeted loss recovery: the served-through tick followed by absolute full values for each recovered system's changed components. Systems spawned since the acknowledged tick ride `Full` instead.
`Despawn`	System despawns, as a count followed by the stopped systems' Ids. Server-only; written after that tick's state subpackets.
`WriteAccess`	Who may write a system's state, as `(SystemId, access, may-this-connection-write)` entries. Server?client only.
`RpcAccess`	Who may send a system a remote call, as `(SystemId, access, may-this-connection-send)` entries. Server?client only.
`DeltaRelayed`	Delta-serialized state the sender is passing on from another peer, rather than a change it made itself.
`DeltaAbsolute`	Delta-serialized state whose changed members carry self-contained absolute values (not baseline-encoded), for a change the sender made itself.
`DeltaAbsoluteRelayed`	Same as `DeltaAbsolute`, but relayed from another peer.
`RosterAdd`	Peer-roster additions, as a count followed by added peers' `Connection.Id`s. Server-only; written ahead of every other kind.
`RosterRemove`	Peer-roster removals, as a count followed by departed peers' `Connection.Id`s. Server-only; written after that tick's despawns.
`SceneMove`	A system's scene changing, as a count followed by `(SystemId, SceneHandle)` entries. Server-only, to established observers only.
`Reparent`	A system's parent changing, as a count followed by `(SystemId, ParentId)` entries. Server-only, to established observers only.

`DeltaAbsolute`/`DeltaAbsoluteRelayed` exist because a plain `Delta` can't say two things a receiver needs and can't re-derive: whether the payload is baseline-encoded or self-contained, and whether the sender authored the change or is relaying somebody else's. Carrying those as the subpacket kind rather than as per-system bits pays for itself once enough systems share the same answer in a tick.

Framing and apply order

Each subpacket is written as its `StatePacketType`, then its bit length, then its body. This makes the stream self-describing: a receiver that doesn't recognize or can't bind an entry steps over it whole using the declared length, instead of losing alignment for everything after it.

A tick's segments are rejoined into one `Reader` before any of this is read. Within that combined stream the declared write order is `RosterAdd`, spawns (`Full`), `Controller`, `SceneMove`, `Reparent`, then the delta kinds, then `Despawn`, then `RosterRemove`. The receiver applies frames in that order — `Full` before `Controller` before `Delta` — so a spawn exists before control or state addresses it, and a despawn is applied only after that system's final state has landed.

RpcRoute

An RPC packet carries an `RpcRoute` naming where the call is headed, packed into two bits by the generated serializer:

- **Server** — a client's call to the server; it stops there.
- **Observers** — a call to every connection observing the system. Legal in both directions: from the server it *is* the fan-out; from a client it's a request for one, admitted and judged by the server's handlers before going out to the other observers.
- **Target** — the server's call to a single observing connection. Server-only; one arriving from a client is discarded and logged, since no engine path produces it.
- **TargetRelay** — a client's call to a single connection it named, passed on once the server admits it. The server rewrites the route to `Target` before forwarding, so the recipient can't tell a routed call from one addressed to it directly.

Direction is validated against the sender's role on receipt, not inferred from which link the packet arrived on — a host holds connections in both directions, so the role has to be checked explicitly.

Segmentation

A `State` packet's combined stream can be too large for one transport unit, so it's split across segments. `StatePacket` carries `SegmentNumber` and `ExpectedSegmentCount` for this: segments span the *whole* combined stream for a tick — every subpacket kind is packed and numbered together — so when everything fits in one transmission unit, `ExpectedSegmentCount` is 1.

`PacketHeader.MaximumOutboundHeaderBytes` (48) is the space reserved ahead of a segment's body for the packet header and `StatePacket` wrapper fields (packet type, tick deltas, acknowledgment ticks, segment number and count) before the transport decides how much room is left for payload. Reserving it up front means the transport never has to run a second segmentation pass after the header grows into space it thought was payload.

`SegmentMode` names how a segment is filled — `Strict` (data must fit the maximum segment size), `Overfill` (exceed it rather than split), `Split` (always fill to the maximum, splitting the data across multiple segments) — but it's an internal enum, not something reachable from outside the engine.

IncomingPacket and OutgoingPacket

`IncomingPacket` is what a received packet looks like once it's on its way into the engine: `Payload` (the raw bytes to read), `Channel`, `Sender` (the `Connection` that produced it), `Invoker` (which role sent it — Server or Client, derived from `Sender.IsServer`), and `Transport` (the transport it arrived on). `ReturnPayloadArray()` returns `Payload`'s backing array to the shared `ArrayPool`.

`OutgoingPacket` is the send-side counterpart: `Payload` (the bytes to send), `Channel`, `Sender`, `Receiver` (the connection to route to), `Writer` (the `Writer` that produced the payload), and `Return()`, which returns the `Writer` to its pool and, if the payload was copied into a pooled array of its own, returns that array too.

The convention across both types is one owner copies once, and one owner returns: a payload lives in its

writer's buffer unless a later stage copies it out, and whichever struct ends up holding that copy is the one responsible for returning it.

This page is for reading the wire, not for writing packets

`PacketHeader` is a public type, but every member on it is `internal`, and `SegmentMode` is `internal` as well. Nothing here is a public API for writing packets — this page is for reading what's on the wire (a log line, a capture, a framing error), not for constructing packets yourself.

Wiring the Source Generator into a .NET Project

Using Unity? See Code Generation in Unity.

Reference the analyzer

Your project needs one analyzer reference: `Nucleus.CodeAnalysis.SourceGenerators.dll`, added as an `Analyzer` item rather than a `Reference`. Take it from the same release as the engine DLLs; the path is wherever you copied that set:

```
<ItemGroup>
    <Analyzer Include="..\lib\Nucleus\Nucleus.CodeAnalysis.SourceGenerators.dll" />
</ItemGroup>
```

Without this reference the project still compiles, but nothing runs the generator over your types: no `Write`/`Read` methods, no module-initializer registration, and nothing replicates. There's no build error to point at it — the failure is silent.

Keep generated output visible and current

Add this alongside the analyzer reference:

```
<PropertyGroup>
    <EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>
</PropertyGroup>
```

`EmitCompilerGeneratedFiles` also writes the generator's output to disk instead of keeping it purely in memory, so you can inspect what it produced. With nothing else set, it lands under `obj/<Configuration>/<TargetFramework>/generated/Nucleus.CodeAnalysis.SourceGenerators/`. The generator already adds this code to the compilation itself, and the `obj` folder is outside your project's own `Compile` items, so nothing gets compiled twice and there is nothing to exclude.

The compiler never deletes old files from that folder, and `dotnet clean` doesn't either, so a file from a previous build can masquerade as this build's output. To see what the current build really produced, delete the `generated` folder and rebuild with `dotnet build --no-incremental`; a plain incremental build that finds nothing changed skips the compiler and leaves the folder empty.

Reference the engine

There's no NuGet package for Nucleus. Reference the built DLL set, `Nucleus.dll`, `CodeBoost.dll`, and `SynapseSocket.dll`, as ordinary `Reference` items; Adding Nucleus to a .NET project shows the project file.

Mixing DLLs from different builds isn't supported; take all four, the generator included, from one release.

Target framework

`Nucleus.dll` targets `netstandard2.1`. The generated registrar uses `[ModuleInitializer]`, which the C# compiler recognizes by full name from any assembly rather than requiring a specific target framework. When your compilation doesn't already have a usable `System.Runtime.CompilerServices.ModuleInitializerAttribute` (true on frameworks older than .NET 5), the generator emits its own internal copy of that attribute into your assembly so the registrar compiles. This happens automatically per assembly; there's nothing in your project to reference for it. Module initializers do need C# 9, though, so a project whose default language version is older (`netstandard2.1` and .NET Core 3.x projects default to C# 8) needs

`<LangVersion>9.0</LangVersion>` or newer.

Give your assembly a bundle id

If your project isn't the main assembly, declare a bundle id so its types don't collide with the default:

```
[assembly: NetworkBundle(1)]
```

See Type Identity on the Wire: Bundles and the Registry for how bundle and local ids combine into a type's wire identity, and which ids are already taken.

The generator skips assemblies that don't reference Nucleus

The generator only runs over an assembly that references Nucleus. A project with the analyzer reference wired in but no reference to Nucleus itself is silently skipped — the same gate applies on the Unity side.

Type Identity on the Wire: Bundles and the Registry

Using Unity? See Bundle Ids for Unity Assemblies.

The identity

Every spawned `NetworkSystem` or `NetworkComponent` is named on the wire by a `(BundleId, LocalId)` pair, not by its type name. The bundle identifier comes from a `[assembly: NetworkBundle(n)]` attribute declared once per assembly; the local identifier is assigned by the source generator within that assembly, starting from an ordinal sort of the type names it finds.

Systems and components use different local identifier widths: a system's local identifier is a `byte`, a component's is a `ushort`. That is a real limit, not a stylistic choice — an assembly can declare far more networked components than networked systems.

An assembly that declares no `[assembly: NetworkBundle(n)]` at all defaults to bundle zero.

NetworkTypeRegistry

`NetworkTypeRegistry` is the static, process-wide map from a type to its wire identity and back to a factory that constructs it. Each bundle's generator emits a module initializer that registers its own types here, so the registry is the union of whatever bundles are loaded when the process starts.

Registration:

- `RegisterSystem<TSystem>(ushort bundleId, byte localId)`
- `RegisterComponent<TComponent>(ushort bundleId, ushort localId)`

Resolving a type to its identity, and back:

- `TryGetSystemId(Type type, out ushort bundleId, out byte localId)`
- `TryGetComponentId(Type type, out ushort bundleId, out ushort localId)`
- `TryCreateSystem(ushort bundleId, byte localId, out NetworkSystem networkSystem)`
- `TryCreateComponent(ushort bundleId, ushort localId, out NetworkComponent networkComponent)`
- `TryGetSystemType(ushort bundleId, byte localId, out Type systemType)`
- `TryGetComponentType(ushort bundleId, ushort localId, out Type componentType)`

`TryGetSystemType`/`TryGetComponentType` resolve an identity to a `Type` without constructing anything — useful when code only needs to decide whether it wants the object, not build one.

By-name forms exist for anything that has to name a type across builds rather than within a single session (an identity resolves to whatever type currently sorts into that position; a name either finds the type it names or finds nothing):

- `TryGetSystemIdByName(string systemTypeName, out ushort bundleId, out byte localId)`
- `TryCreateSystemByName(string systemTypeName, out NetworkSystem networkSystem)`
- `TryGetSystemName(Type systemType, out string systemTypeName)`
- `TryGetComponentName(Type componentType, out string componentTypeName)`

- `TryCreateComponentByName(string componentTypeName, out NetworkComponent networkComponent)`

`TryReturnSystem(NetworkSystem networkSystem)` returns a system to the pool its registered factory rented it from, resolved by the system's runtime type. It's the counterpart to `RegisterSystem` for anything holding a system by its base type — the engine's own wire construction, and any integration that rents systems ahead of a spawn that might never bind them.

```
if (NetworkTypeRegistry.TryGetSystemId(typeof(MySystem), out ushort bundleId, out byte loc
alId))
{
    // bundleId/localId are the wire identity for MySystem.
}

if (NetworkTypeRegistry.TryCreateSystem(bundleId, localId, out NetworkSystem system))
{
    // system was rented from MySystem's pool.
}
```

Reserved values

`NetworkTypeRegistry.FrameworkBundleId` is `ushort.MaxValue`, reserved for the framework's own built-in types so a user's main assembly can default to bundle zero without colliding.

`NetworkTypeRegistry.BaseNetworkSystemLocalId` is `0`, reserved for the base `NetworkSystem` type itself. The base type is concrete and usable directly, but the generator only numbers `NetworkSystem` subclasses, so `NetworkSystemRegistration` registers the base type by hand — under `FrameworkBundleId`, not a separate "main" bundle. Generator-assigned system local identifiers therefore start at one.

Bundle identifier framing

`BundleIdWireFormat` is internal — it's design rationale for why bundle assignment order matters, not an API you call. Identifiers are handed out from both ends of the sixteen-bit space: a game counts up from zero (bundle zero, then its content shards at one, two, three...), while the framework counts down from the top (`FrameworkBundleId` at `ushort.MaxValue`, the Unity integration one below it). Before a bundle identifier is written to the wire, the two ends of the range are folded together so an id near either end costs the fewest bits — bundle zero, the framework, the game's first content shard, and the Unity integration all land among the cheapest identifiers.

The practical consequence: assign bundle identifiers from an end of the space, and low. An identifier chosen from the middle of the space gets none of that benefit.

Declared compositions

A composition is the set of component types named together at one spawn call site. The generator collects every distinct composition it finds in the build and registers it:

- `RegisterDeclaredComposition(params Type[] componentTypes)`
- `IsDeclaredComposition(List<uint> sortedComponentIds)`
- `GetComponentCompositionId(ushort bundleId, ushort localId)` packs a component's wire identity into the single value the composition check sorts and compares.

This answers "could this build have made this" about an object described by a peer rather than constructed locally. It matters for a receiver resolving a code-only spawn: a composition nothing in the build declares is

one no legitimate call site could have produced, which the receiver can use to refuse it rather than trust it blindly.

What this prevents

Because identity is a `(BundleId, LocalId)` pair scoped per assembly rather than a name or an index into a single global list, a content shard built and shipped separately from the main game cannot end up handing out the same identity as a type the main game already uses. Each bundle owns its own local identifier space; only the combination has to be unique, and that uniqueness is enforced by each assembly declaring its own `[assembly: NetworkBundle(n)]`.

What the Generator Emits

The per-component partial

A `NetworkComponent` you declare is a partial class. At build time, `NetworkComponentBuilder` fills in the other half: a full `Write`/`Read` pair, a delta `WriteDelta`/`ReadDelta` pair, and a `[Flags] ulong` enum recording which fields a delta touched. `Write` and `Read` walk every replicated `NetworkMember` field in declaration order; `WriteDelta` and `ReadDelta` do the same, reserving one bit per member up front and only touching a member whose bit is set. The change-flags enum is named `{YourTypeName}Flags`, with one member per declared field starting at `None = 0`.

A component with no replicated members emits neither pair at all, inheriting the empty base implementation. Both peers compile the same decision, so the wire format agrees without either side needing to check at runtime.

The generator finds types by walking fields, not by scanning for attributes

The generator finds types by walking fields, not by scanning for an attribute. Any type reached through a `NetworkMember<T0>` field - including a plain struct you define yourself - gets a full serializer with no marker required. `[NetworkType]` exists for the opposite case: a type nothing reaches this way, which needs the attribute to force a serializer into existence.

An inherited field is written and read too, ahead of the declaring type's own fields, so subclassing a plain base class with public fields still nets every value on the wire.

Full and delta halves are generated together

A type's full serializer and its delta serializer are produced by the same pass, from the same member list. If a member's type has no full writer, the generator reports it in a warning naming the member and drops it from both halves; it never emits one half without the other. A member the generator genuinely cannot serialize is therefore a build-time diagnostic, not something discovered later on the wire.

Each assembly registers its own types as it loads

Each assembly gets one generated `NetworkTypeRegistry` class carrying a `[ModuleInitializer]` -attributed method. It registers every discovered `NetworkSystem` subclass and every constructible `NetworkComponent` type into `Nucleus.Serializers.NetworkTypeRegistry` as the assembly loads, calling `RegisterSystem<T>(bundleId, localId)` and `RegisterComponent<T>(bundleId, localId)` for each. Nothing needs to call this and there is no reflection walk at runtime to build it: the module initializer runs once, at load, before any of the assembly's own code.

Registration order is deterministic: systems and components are each sorted with `OrderBy(..., StringComparer.Ordinal)` on their full type name before numbering, so a rebuild from the same source always assigns the same local ids.

Local ids are numbered per bundle, from two different starting points

Local ids are numbered per assembly (per bundle), and the two counters start differently:

- **Systems** start at `1`. Local id `0` is reserved for the base `NetworkSystem`.

- **Components** start at `0`.

Every bundle numbers its own systems from one and its own components from zero; the numbering resets per assembly and is paired with that assembly's bundle id to form a type's wire identity.

Equality comparers

`ComparerBuilder` generates a static equality-comparer method for every discovered networked type, invoked through `NetworkTypeEqualityComparer<T0>.Compare`. Delta serializers use these to decide whether a member changed. A container type compares member by member, recursing into the comparer generated for each member's type; a primitive array is compared by content via `SequenceEqual` rather than by reference, so an in-place edit is detected as a change.

Enum packing

An enum member is packed to the minimum bit width that round-trips every value the enum can hold, when its underlying type is unsigned and none of its declared values are negative - the same rule applies whether or not the enum is `[Flags]`. The delta writer for a bit-packable enum writes only when the value changed; where the width can't be determined this way, the enum falls back to serializing through its underlying type's own serializer.

SharedSignatures and CodeBoost's [CreateSignature]

Generated code and hand-written engine code both need to compile against the same shapes - a generated `Write` override has to see the exact same `NetworkMember<T0>.Write` signature the hand-written base class declares. CodeBoost's `[CreateSignature]` attribute marks the members whose signature must mirror into `Nucleus.CodeAnalysis.SourceGenerators/SharedSignatures`, a tree of `.g.cs` files under that path (92 of them, as of this writing) that the generator project reads without depending on the whole engine assembly.

These `.g.cs` files are tracked source, not build output. They're regenerated deliberately when a signature changes, then committed and edited like any other file the generator depends on - never gitignored, never deleted on the assumption that a build will replace them.

A single type can declare at most 62 network members

`Serializers.Constants.MaximumNetworkMemberCount` is 63, and the usable count is one less: the generator reports `SERIALIZERS000` for a type that declares 63 or more network members, so a single type can declare at most 62. The ceiling exists because the delta flags backing each type's change set is a `ulong` and the shift that sets a member's bit is computed as a 64-bit operation to avoid aliasing member 32 onto member 0. The check covers the structs and classes the generator writes serializers for; it is not applied to a `NetworkComponent`, whose own limit is described in How replicated state works.

Writer and Reader

Writer and Reader are the bit-level building blocks

`Writer` and `Reader` are the bit-level building blocks every Nucleus serializer, RPC struct, and transport is written against. A `Writer` packs values onto a growable bit buffer; a matching `Reader` walks the same buffer back out, in the same order, at the same compression level. Both are pooled types: rent one, write or read, then return it.

Renting and returning a Writer

`WriterPool` is the entry point. It never allocates a new `Writer()` for you to hold onto past the call.

- `WriterPool.Rent()` - an uninitialized writer, for cases with no `CoreManager` yet.
- `WriterPool.Rent(CoreManager coreManager)` - the common case; initializes the writer against that manager.
- `WriterPool.RentByBitCapacity(uint bitCapacity, CoreManager coreManager)` / `RentByByteCapacity(uint byteCapacity, CoreManager coreManager)` - rent from a capacity-bucketed pool instead of the general one, pre-sized to avoid an early buffer resize. Buckets are 1000-byte windows; a writer rented this way should be returned with `ReturnByCapacity`, not `Return`, so it goes back into the same bucket.
- `WriterPool.RentList()` / `RentList(int writerCount)` and `WriterPool.RentDictionary<T0>()` - pooled collections of writers, for code that needs to build several payloads at once (for example, one writer per observer).

Returning:

- `WriterPool.Return(Writer? writer)` - the general-pool counterpart to `Rent()` / `Rent(CoreManager)`.
- `WriterPool.ReturnAndNullifyReference(ref Writer? writer)` - returns and nulls the caller's reference in one call, so a stale reference can't be reused after return.
- `WriterPool.ReturnByCapacity(Writer? writer)` / `ReturnByCapacityAndNullifyReference(ref Writer? writer)` - the counterpart to the `RentByBitCapacity` / `RentByByteCapacity` bucketed pool.
- `WriterPool.Reset(List<Writer?> writers)` / `Return(List<Writer?> writers)` / `ReturnAndNullifyReference(ref List<Writer?> writers)` - return every writer in a rented list (and, for the last two, the list itself).

Renting and returning a Reader

`ReaderPool` is not a mirror of `WriterPool`. There is no capacity-bucketed rent or return on the reader side - a reader's size is fixed by the bytes it was handed, so there's nothing to pre-size. Instead, `ReaderPool.Rent` comes in three forms depending on what you're initializing from:

- `ReaderPool.Rent(byte[] packedBytes, byte excessBits, CoreManager coreManager, Connection senderConnection)` - from a raw byte array plus the excess-bit count in its final byte.
- `ReaderPool.Rent(PackedBytes packedBytes, CoreManager coreManager, Connection senderConnection)` - from an already-built `PackedBytes`.
- `ReaderPool.Rent(CoreManager coreManager, Connection senderConnection)` - an empty

reader, for code that will add read spans itself.

Returning:

- `ReaderPool.Return(Reader? reader)` / `ReaderPool.ReturnAndNullifyReference(ref Reader? reader)`.
- `ReaderPool.Reset` / `Return` / `ReturnAndNullifyReference` overloads for both `List<Reader?>` and `Dictionary<T0, Reader>`.

Writing values

Every full-value write method takes a `CompressionLevel` (`Aggressive`, `Tight`, `Loose`, or `Uncompressed`), defaulting to `Loose`. The numeric families:

```
void WriteInt8(sbyte value, CompressionLevel compressionLevel = CompressionLevel.Loose);
void WriteUInt8(byte value, CompressionLevel compressionLevel = CompressionLevel.Loose);
void WriteInt16(short value, CompressionLevel compressionLevel = CompressionLevel.Loose);
void WriteUInt16(ushort value, CompressionLevel compressionLevel = CompressionLevel.Loose)
;
void WriteInt32(int value, CompressionLevel compressionLevel = CompressionLevel.Loose);
void WriteUInt32(uint value, CompressionLevel compressionLevel = CompressionLevel.Loose);
void WriteInt64(long value, CompressionLevel compressionLevel = CompressionLevel.Loose);
void WriteUInt64(ulong value, CompressionLevel compressionLevel = CompressionLevel.Loose);
void WriteBoolean(bool value, CompressionLevel compressionLevel = CompressionLevel.Loose);
void WriteChar(char value, CompressionLevel compressionLevel = CompressionLevel.Loose);
```

`WriteSingle`, `WriteDouble`, and `WriteDecimal` add an `accuracy` argument, which controls how much precision the compression is allowed to discard:

```
void WriteSingle(float value, CompressionLevel compressionLevel = CompressionLevel.Loose,
float accuracy = DefaultFloatingAccuracyAsSingle);
void WriteDouble(double value, CompressionLevel compressionLevel = CompressionLevel.Loose,
 float accuracy = DefaultFloatingAccuracyAsSingle);
void WriteDecimal(decimal value, CompressionLevel compressionLevel = CompressionLevel.Loos
e, float accuracy = DefaultFloatingAccuracyAsSingle);
```

Other full-value writes:

```
void WriteString(string value, CompressionLevel compressionLevel = CompressionLevel.Loose)
;
void WriteGuid(Guid value, CompressionLevel compressionLevel = CompressionLevel.Loose);
void WriteDateTime(DateTime dateTime, CompressionLevel compressionLevel = CompressionLevel
.Loose);
void WriteMatrix4X4(Matrix4x4 value, CompressionLevel compressionLevel = CompressionLevel.
Loose, float accuracy = DefaultFloatingAccuracyAsSingle);
void WriteQuaternion(Quaternion value, CompressionLevel compressionLevel = CompressionLeve
l.Loose, float accuracy = DefaultQuaternionAccuracyAsSingle);
```

Collections write their count alongside the values:

```
void WriteUInt8ArrayWithCountAsBytes(byte[] bytes, CompressionLevel compressionLevel = Com
pressionLevel.Uncompressed);
void WriteArraySegmentWithCountAsBytes(ArraySegment<byte> value, CompressionLevel compress
ionLevel = CompressionLevel.Uncompressed);
void WriteListWithCount<T0>(List<T0> value, CompressionLevel compressionLevel = Compressio
nLevel.Loose);
void WriteArrayWithCount<T0>(T0[] value, CompressionLevel compressionLevel = CompressionLe
vel.Loose);
void WriteDictionary<T0, T1>(Dictionary<T0, T1> value, CompressionLevel compressionLevel =
 CompressionLevel.Loose);
```

`WriteNullable<T0>` writes a presence flag then the value:

```
void WriteNullable<T0>(T0? value, CompressionLevel compressionLevel = CompressionLevel.Loo
se) where T0 : struct;
```

And the generic form dispatches to whichever writer is registered for `T0`:

```
void Write<T0>(T0 value, CompressionLevel compressionLevel = CompressionLevel.Loose);
```

This is what a hand-rolled struct serializer or custom transport payload is built from.

Writing deltas

The delta family writes a value only relative to a previous one, and every method returns whether anything was actually written - `false` means the value was unchanged and nothing landed on the wire (under `DeltaCheckMode.Checked`; `DeltaCheckMode.Unchecked` always writes). These take a `DeltaCheckMode` instead of a `CompressionLevel` - the compression level used is `Writer.DeltaCompressionLevel`.

```
bool TryWriteDeltaInt8(sbyte previousValue, sbyte nextValue, DeltaCheckMode deltaCheckMode
);
bool TryWriteDeltaUInt8(byte previousValue, byte nextValue, DeltaCheckMode deltaCheckMode)
;
bool TryWriteDeltaInt16(short previousValue, short nextValue, DeltaCheckMode deltaCheckMod
e);
bool TryWriteDeltaUInt16(ushort previousValue, ushort nextValue, DeltaCheckMode deltaCheck
Mode);
bool TryWriteDeltaInt32(int previousValue, int nextValue, DeltaCheckMode deltaCheckMode);
bool TryWriteDeltaUInt32(uint previousValue, uint nextValue, DeltaCheckMode deltaCheckMode
);
bool TryWriteDeltaInt64(long previousValue, long nextValue, DeltaCheckMode deltaCheckMode)
;
bool TryWriteDeltaUInt64(ulong previousValue, ulong nextValue, DeltaCheckMode deltaCheckMo
de);
bool TryWriteDeltaSingle(float previousValue, float nextValue, DeltaCheckMode deltaCheckMo
de, float accuracy = DefaultFloatingAccuracyAsSingle);
bool TryWriteDeltaDouble(double previousValue, double nextValue, DeltaCheckMode deltaCheck
Mode, float accuracy = DefaultFloatingAccuracyAsSingle);
bool TryWriteDeltaDecimal(decimal previousValue, decimal nextValue, DeltaCheckMode deltaCh
eckMode, float accuracy = DefaultFloatingAccuracyAsSingle);
bool TryWriteDeltaString(string previousValue, string nextValue, DeltaCheckMode deltaCheck
Mode);
bool TryWriteDeltaGuid(Guid previousValue, Guid nextValue, DeltaCheckMode deltaCheckMode);
bool TryWriteDeltaDateTime(DateTime previousValue, DateTime nextValue, DeltaCheckMode delt
aCheckMode);
bool TryWriteDeltaQuaternion(Quaternion previousValue, Quaternion nextValue, DeltaCheckMod
e deltaCheckMode, float accuracy = DefaultQuaternionAccuracyAsSingle);
bool TryWriteDelta<T0>(T0 previousValue, T0 nextValue, DeltaCheckMode deltaCheckMode);
```

A member written this way can choose between `TransmissionMode.Divine` and `TransmissionMode.Interval`. Divine can go quiet for stretches where Interval would still send on every tick; Interval sends on a fixed cadence regardless of whether the value moved. Both are black boxes from the writer's side - `TryWriteDelta*` behaves the same either way, returning whether it wrote.

Raw bits and positioning

Below the typed writes, a `Writer` is a flat bit buffer you can address directly:

```
void WriteBits(ulong value, uint bitCount);
void WriteBits(ulong value, uint bitCount, uint position);
void WriteBits(Writer otherWriter);
uint SkipBits(uint bitCount);
uint SetBitBufferIndex(uint value);
```

`WriteBits(value, bitCount)` writes at the current position and advances it. The positional overload writes `bitCount` bits at an absolute `position` without moving the current write index - useful for patching a length or flag written earlier once the real value is known. `WriteBits(Writer otherWriter)` appends another writer's full bit buffer onto this one. `SkipBits(bitCount)` advances the position by `bitCount` without writing anything (reserving space) and returns the position it started from; `SetBitBufferIndex(value)` jumps the write position directly and also returns the prior position - both are how you go back and fill in a reserved span with `WriteBits(value, bitCount, position)`.

Capacity is grown explicitly, not implicitly per write:

```
void CreateCapacityAsBits(uint bitCount);
void CreateCapacityAsBytes(uint byteCount);
void CreateCapacityForArray(int writeCount);
```

Getting the bytes out

```
uint BitCount { get; }
uint BitIndex { get; }
uint BitCapacity { get; }
uint PackedByteCount { get; }
byte GetExcessBitCount();
ArraySegment<byte> ToPackedByteArraySegment(out byte excessBitCount);
PackedBytes ToPackedBytes();
```

`BitCount` is the number of bits written so far; `BitIndex` is that count's position within its final byte; `BitCapacity` is the buffer's current allocated size in bits; `PackedByteCount` is how many bytes the written bits pack into. `GetExcessBitCount()` returns how many bits of padding the final byte carries. `ToPackedByteArraySegment` and `ToPackedBytes` both hand back the written bits - as a raw `ArraySegment<byte>` with the excess count out-parameter, or wrapped in a `PackedBytes` struct - ready to hand to a transport or a `Reader`.

`PackedBytes` (and its read-only interface, `IReadOnlyPackedBytes`) is the value type that carries those bytes plus the excess-bit count between a writer and a reader, or across a transport boundary.

For a surface that must not mutate the writer or reader it wraps, use `IReadOnlyWriter` and `IReadOnlyReader` - read-only views exposing the counters and byte-extraction methods without any write or read methods.

Connections on the wire

```
void WriteConnection(Connection connection, CompressionLevel compressionLevel = Compressio
nLevel.Loose);
Connection ReadConnection(CompressionLevel compressionLevel = CompressionLevel.Loose);
```

These write and read a `Connection` reference by id. Treat that id as a claim, not a fact: anyone can write any connection id into a payload they send. The only trustworthy answer to "who actually sent this" is the context's sender connection - `Managers.Rpc.RpcContext.SenderConnection` - which comes from the transport, not from anything inside the payload. Never authorize an action based on a `ReadConnection` result alone.

Reading values back

A `Reader` mirrors the writer's typed reads one for one (`ReadInt8`, `ReadSingle`, `ReadString`, `ReadDelta<T0>`, and so on). A full read takes the `CompressionLevel` the write used. A delta read takes the previous value and no `DeltaCheckMode`; its compression level is `Reader.DeltaCompressionLevel`. Every read that has a default uses the same default as its write (`Loose`, including `ReadString`), so a pair called with no level on either side agrees. `ReadQuaternion` is the exception that has no default level at all, so it must be told the level the write used. Passing the level explicitly on both sides is still the clearest way to keep a pair in step when you change one of them. Beyond the per-value reads, a `Reader` carries:

```
CoreManager CoreManager { get; }
Connection Sender { get; }
uint BitsRemaining { get; }
bool HasBitsRemaining { get; }
CompressionLevel DeltaCompressionLevel { get; set; }
```

`Sender` is the connection the reader was initialized with (`ReaderPool.Rent(..., senderConnection)`). `BitsRemaining` and `HasBitsRemaining` tell you whether there's more to read; there is no `ReadSpan` property on `Reader` - `ReadSpan` is a separate internal type, and the reader instead exposes `HasReadSpan` (from `IReadOnlyReader`) to say whether at least one has been added. `DeltaCompressionLevel` must match the value the writer used for its `TryWriteDelta*` calls, or the delta reads will decode incorrectly and desync the rest of the stream.

Nucleus Networking Documentation

Nucleus is a C#/.NET networking engine with an optional Unity integration. Pick a place to start:

- Start Here
- Unity
- Core API
- Reference & Project Meta

API type reference

Where to start

Every session hangs off one object: `CoreManager`. Construct one and it builds the whole manager graph for you; everything else in the engine is reached from a manager on that graph, not constructed on its own.

```
public readonly NetworkLoopManager NetworkLoopManager;
public readonly TransportManager TransportManager;
public readonly ServerManager ServerManager;
public readonly ClientManager ClientManager;
public readonly SystemManager SystemManager;
public readonly InterestManager InterestManager;
public readonly Scenes.SceneManager SceneManager;
public readonly PacketManager PacketManager;
public readonly MessageManager MessageManager;
public readonly RpcManager RpcManager;
public readonly ViolationManager ViolationManager;
```

These eleven fields are `readonly` and set once, in the constructor, in this order. A Pro build adds two more, assigned through partial constructor hooks rather than the constructor body itself:

```
public BundleManager BundleManager;
public WorldPersistenceManager WorldPersistenceManager;
```

`BundleManager` and `WorldPersistenceManager` are not `readonly` because the language does not allow a partial hook to assign a readonly field outside the declaring partial's own constructor body. In a Free build the hooks that would construct them compile away to nothing, and the fields themselves do not exist on the type at all — code referencing them fails to compile against a Free build rather than failing at runtime.

`CoreManager.Instance` is a static convenience slot naming the oldest live `CoreManager` in the process, for the common case of a process running exactly one. It is not meant for a process running several managers at once (a host pair, a two-peer test) — pass the manager explicitly there instead. `CoreManager.EnsureSetInstance(CoreManager)` moves the slot to a manager of your choosing; it fails and logs if the manager passed is null or already deinitialized.

Namespace map

Namespace	What lives there
`Nucleus.Managers.Core`	`CoreManager` itself
`Nucleus.Managers.NetworkLoop`	The tick driver and its step providers
`Nucleus.Managers.Transports`	`TransportManager`, the active `Transport`
`Nucleus.Managers.Server`	`ServerManager`
`Nucleus.Managers.Client`	`ClientManager`
`Nucleus.Managers.Systems`	`SystemManager` and system lifecycle
`Nucleus.Managers.Interest`	`InterestManager`
`Nucleus.Managers.Scenes`	`SceneManager`
`Nucleus.Managers.Messages`	`MessageManager`

`Nucleus.Managers.Rpc`	`RpcManager`
`Nucleus.Managers.Violations`	`ViolationManager` and the shipped violation types
`Nucleus.Managers.Bundles`	`BundleManager` (Pro)
`Nucleus.Managers.Persistence`	`WorldPersistenceManager` (Pro)
`Nucleus.Systems`	`NetworkSystem` and the members you declare on one
`Nucleus.Components`	Engine-neutral replicated components, such as `TransformComponent`
`Nucleus.Connections`	`Connection` and peer-side state
`Nucleus.Serializers`	Reader/writer contracts and generated serializer plumbing
`Nucleus.Packets`	Wire packet types
`Nucleus.Physics`	Physics-adjacent components and adapters
`Nucleus.Transports`	Transport base types, plus the `Synapse`, `Yak`, and `Sockets` sub-namespaces for the shipped transport implementations

Everything reachable from a `CoreManager` field lives under `Nucleus.Managers.*`; the types those managers hand you (systems, components, connections, packets, serializers) live in the flatter namespaces above.

Reading the markers

Three attributes tell you how a member is meant to be used before you call it. They are declared in CodeBoost, the utility library Nucleus builds on, in the `CodeBoost.Types` namespace (see CodeBoost reference):

- `[ServerOnly]` — only meaningful, or only safe to call, from server-side code.
- `[ClientOnly]` — only meaningful, or only safe to call, from client-side code.
- `[InternalApi]` — exposed for convenience but primarily for internal use; it can change or move without warning, and takes an optional `Details` string explaining why.

None of these are compiler-enforced access restrictions — they are promises about intended use, not guardrails that stop a misuse from compiling.

A type or member declared in a `*.Pro.cs` file is a build-time absence, not a runtime failure: `CoreManager.BundleManager` and `CoreManager.WorldPersistenceManager` simply do not exist on the compiled type in a Free build. Code that references them against a Free build fails to compile, it does not throw at runtime.

What is not extensible from outside the assembly

A `Transport` is built on `CommonSocket`, whose connect, send, and receive members are `internal`. `InternalsVisibleTo` is granted to exactly one assembly, `Nucleus.Integrations.BlitzRelay`, and to nothing else. Writing a custom transport from outside the engine assembly is not currently possible — today that's an in-tree exercise, done as a fork or a PR against `Nucleus.Transports`, not as an external package.

Full type listing

This page and the rest of the hand-written reference never try to enumerate every type. They cover the surface you're expected to reach for directly. A generated, exhaustive API reference is planned but not yet published.

Working in Unity

If you're building against the Unity integration rather than plain C#, see the manager components reference for the `UnityManager`-derived components that front each core manager on the manager GameObject.

CodeBoost reference

CodeBoost is a separate library that ships beside Nucleus and that the engine compiles against for logging, pooling and a handful of low-allocation types. It is public as its own submodule under its own terms, not part of the Nucleus API surface, and yours to use directly. CodeBoost.dll ships inside the Unity package too, so a Unity project reaching for its pooling or logging is exactly as supported as a plain .NET host.

Logging

`ILogger` is the swap point. It declares `GetLoggerSetting()`, `DisableUnconditionalDevelopmentStacktrace()`, and `LogInformation`/`LogWarning`/`LogError`. `LoggingService` holds the active instance in its static `Logger` field and starts with a `ConsoleLogger` already registered, so a plain .NET host gets console output with no setup. Call `LoggingService.UseLogger(ILogger)` to replace it.

```
LoggingService.UseLogger(new MyServiceLogger());
```

`LoggerSetting` carries three `LoggerLevel` values (`Editor`, `DevelopmentBuilds`, `ReleaseBuilds`), each one of `Disabled`, `Error`, `Warning`, or `Information`. `LoggingService` reads whichever applies to the current environment and caches it. Check `LoggingService.IsInformationEnabled` / `IsWarningEnabled` / `IsErrorEnabled` before formatting an interpolated log argument, so the string isn't built when the level would drop it anyway. For a level chosen at runtime, `Logger.IsEnabled(LoggerLevel)` answers the same question, `Logger.Log` takes the level as an argument, and a `LogEntry` carries a level and a message together so a line can be decided in one place and written in another; see Logging.

`DisableUnconditionalDevelopmentStacktrace()` on `ILogger` controls whether a development build appends a stack trace to every log call regardless of level; `ConsoleLogger` returns `false`, because nothing else adds a trace to console output, so a development build traces its info and warning logs too. Errors always carry one.

CodeBoost provides two object pools

`ObjectPool<T0>` (constraint `new()`) rents and returns plain objects through a thread-local stack backed by a shared global stack. `ResettableObjectPool<T0>` does the same for a type implementing `IPoolResettable`, calling `OnRent()` after renting and `OnReturn()` before storing the instance back. Both expose `Rent()`, `Return(T0)`, and `ReturnAndNullifyReference(ref T0)`.

The collection pools follow the same shape for built-in collections: `ListPool<T0>`, `HashSetPool<T0>`, `DictionaryPool<T0, T1>`, `QueuePool<T0>`. `Utf8EncodingPool` pools `UTF8Encoding` instances the same way, without a resettable contract since the type has no mutable state to reset.

```
List<int> list = ListPool<int>.Rent();
try
{
    // use list
}
finally
{
    ListPool<int>.Return(list);
}
```

Types worth knowing

Several Nucleus signatures pass these CodeBoost types directly:

- `RingBuffer<T0>` — a fixed-size collection that overwrites its oldest entries once full.
- `BoostedQueue<T0>` — a queue-shaped collection used in place of `Queue<T0>` where Unity 2022's codegen can't compile a `Queue<T0>` reference.
- `RentedArray<T0>` — an `IDisposable` wrapper around an `ArrayPool<T0>` rental; disposing it returns the array.
- `RoundRobinCursor<T0>` — walks a read-only list in wrap-around batches sized to sweep the whole list once over a configured time window.
- `Vector2Int`, `Vector3Int32`, `Vector3Int64`, `Vector4Int` — integer vectors.
- `QuaternionInt16`, `QuaternionInt32` — integer-component quaternions.
- `UnifiedColor` — an RGBA color stored as four bytes, with float accessors.

Build-time pool-reset checking

`[PoolResettableMember]` marks a field or property that must be reset on `IPoolResettable.OnReturn()` and initialized on `OnRent()`. `[PoolResettableMethod]` marks a method that handles that reset/initialize work for one or more of those members, for cases where it isn't done inline in `OnRent`/`OnReturn`. `[PoolDisposableMember]` marks a member that must be disposed rather than merely reset.

An analyzer in `CodeBoost.CodeAnalysis.Analyzers` checks every `IPoolResettable` type against its declared `[PoolResettableMember]` fields and fails the build if one is left out of the reset path. A pooled type that forgets to clear a marked member is a build error, not a runtime surprise.

Environment and Mathematics

`IApplicationState`, reached through the static `ApplicationState` class, answers `IsEditor()`, `IsDevelopment()`, `IsGuiBuild()`, `IsHeadlessBuild()`, `IsPlaying()`, and `IsQuitting()`. `LoggingService` uses it to pick which `LoggerLevel` applies.

`MathCb`, in `CodeBoost.Mathematics`, is a static partial class of floating-point and vector utility methods. `RoundingType` (`Down`, `Up`, `UpNonZero`, `ToEven`, `AwayFromZero`) selects the rounding rule for the integer divide extension methods.

Compatibility

Nucleus runs anywhere netstandard2.1 runs

Nucleus's core targets `netstandard2.1`, with nullable reference types enabled and unsafe blocks allowed. Anything that implements netstandard2.1 runs it: .NET 5 and later, Mono, and Unity, plus .NET Core 3.x with two additions to the consuming project, a `PackageReference` to `System.Runtime.CompilerServices.Unsafe` 5.0.0 (which `CodeBoost.dll` depends on) and `<LangVersion>9.0</LangVersion>` (the generated code uses module initializers). The core has no engine dependency, so this is one axis independent of the Unity integration below.

Unity integration

The Unity integration compiles under `UNITY_EDITOR || UNITY_2021_3_OR_NEWER`. Older Unity versions, outside the editor, don't compile it.

Integration and demo source is pinned to `LangVersion 9`, matching what Unity's compiler accepts. This constrains the source shipped in the package, not the compiled DLLs it references — the core engine DLL is built separately and isn't subject to this pin.

Optional dependencies

A few integration paths are gated behind package presence, and only compile in when the define is set:

Dependency	Define	Gates
`com.unity.addressables` 1.19+	`NUCLEUS_ADDRESSABLES`	Addressables-based asset loading
`com.unity.modules.ai`, the AI module, enabled in a new project	`NUCLEUS_NAVMESH`	`NetworkNavMeshAgent` and its inspector
`Unity.ResourceManager`	—	Referenced alongside Addressables
Relay transport	`BLITZ_RELAY`	The Blitz Relay transport (Pro only)

Without these packages installed, the corresponding code simply isn't compiled in — no stub, no runtime check.

The demos have requirements of their own: the Input System, uGUI 2.0 or newer (for TextMeshPro), AI Navigation and Multiplayer Play Mode packages, and the Vehicles and Terrain modules. Every demo assembly is gated on them, so a project without one of them still compiles. The demos are skipped, and the editor warns once per session naming the packages to install.

Toolchain for building from source

The source generators (`Nucleus.CodeAnalysis.SourceGenerators`) target `netstandard2.0` and are consumed as Roslyn analyzers, not as a runtime dependency. Any toolchain building Nucleus from source needs an SDK that can host netstandard2.0 analyzers alongside the netstandard2.1 core.

These targets are closed, not pending

These are closed, not pending:

- No WebSocket or WebGL transport, and so no browser target.
- No Steam, EOS, or console transport ships.

A "does platform X work" question is almost always a transport question, not a Nucleus core question — the core is platform-agnostic; what reaches a given platform is whichever transport you plug in.

Component Index

Core managers

Component	Description
`UnityCoreManager`	Owns the process's single CoreManager and bootstraps every other `UnityManager` on its GameObject.
`UnityClientManager`	Inspector-side driver for the core client manager.
`UnityServerManager`	Inspector-side driver for the core server manager.
`UnityTransportManager`	Adds and configures the CoreManager's transport, and chooses whether the peer starts as a server, client, or neither.
`UnityNetworkLoopManager`	Carries the authored tick rate and gives the default physics world its driver.
`UnityPacketManager`	Inspector-side driver for the core packet manager.
`UnityMessageManager`	Inspector-side driver for the core message manager.
`UnitySystemManager`	Exposes the system manager's runtime knobs as serialized fields.
`UnityInterestManager`	Carries the world-wide interest rules, re-resolve interval, and per-tick spawn cap.
`UnityPhysicsManager`	Inspector-side seat for the process-global physics execution mode and fixed-step alignment.
`UnitySceneManager`	Exposes the scene manager's placement policy and supplies a scene loader if none is placed.

State

Component	Description
`NetworkTransform`	Replicates a GameObject's position, rotation, and scale, in local or world space.
`NetworkAnimator`	Replicates an `Animator`'s parameters, layer weights, and speed.
`NetworkNavMeshAgent`	Replicates a `NavMeshAgent` as the path leg it is walking rather than as its raw pose.

Scenes

Component	Description

`NetworkSceneLoader`	Base behaviour for loading networked scenes; registers itself as the peer's scene loader.
`UnitySceneLoader`	Loads networked scenes additively through Unity's own scene loading.
`NetworkSceneManifest` (asset)	Maps the scene identifiers named on the wire to the scenes this peer loads and the content bundle each ships in.

Physics

Component	Description
`PhysicsSimulationDriver`	Aligns one scene's physics world with the network tick instead of Unity's own FixedUpdate.
`ProjectedRigidbody`	Declares a rigidbody's replicated physics component and drives its sync, adopting, interpolating, or converging as needed.

Lag compensation

Component	Description
`UnityLagCompensationManager`	Inspector-side seat for lag compensation (Pro): the rewind window, whether 2D queries reach 3D hitboxes, whether queries start in hitboxes, and the spawn compensation cap.
`NetworkHitboxRoot`	Gathers every `NetworkHitbox` on a networked object and registers them with the server, so shots can be judged against where they were; disabling it makes the object unhittable.
`NetworkHitbox`	One lag-compensated volume, a sphere, box or capsule or a 2D circle, box or capsule, copied from a collider beside it or typed in.

Transports

Component	Description
`NetworkTransport`	Base for an Inspector-configurable transport component.
`SynapseTransport`	Inspector-configurable component for the Synapse UDP transport: port, remote host, and timeouts.
`BlitzRelayTransport`	Inspector-configurable component for the Blitz Relay transport, which carries a session through a relay.

Bundles

Component	Description
`NetworkBundleLoader`	Base behaviour for loading content bundles; registers itself as the peer's bundle loader.
`UnityAssetBundleLoader`	Loads content bundles as Unity asset bundles and registers the prefabs they ship.
`NetworkBundleManifest` (asset)	Maps the bundle identifiers named on the wire to the asset bundle files this client loads them from.

Systems and spawning

Component	Description
`NetworkSystemObject`	Marks the root of a networked prefab or scene object; the single link between the GameObject and its NetworkSystem.
`NetworkPrefabCollection` (asset)	A serialized, enumerable container of networked prefabs shipped with a content unit.
`NetworkPlayerSpawner`	Spawns one prefab per authenticated client and hands that client control of it.
`NetworkInterestObject`	Authors an object's interest rules once, for every NetworkSystem it carries.
`NetworkHostVisibility`	Hides an object's renderers on a host while its own client half would not have been holding the object.
`NucleusBehaviour<TComponent0>`	Base for a script whose lifetime is one networked system; exposes the typed component it carries and this peer's role and control.

Developer tools

Component	Description
`NetworkCorrectionVisualizer`	Flashes a line from where a predicted position was shown to where a reconcile corrected it, on the client that predicts the object.
`NetworkInterestOverlay`	While selected on a server or host, draws a line to every networked object, colored by what the object's controller holds of it.

Configuration reference

There is no engine-wide configuration object. Every setting is a public field or property on the manager that owns it, assigned after `new CoreManager()` returns. This page lists them by manager, with defaults and whether each can still change after start-up.

Using Unity? See The Nucleus manager components for setting the same values from the inspector.

Construction-only values

Most settings can change at any time. Two are exceptions:

- **Tick rate.** `NetworkLoopManager.TickRate` is get-only, fixed by the `tickRate` argument passed when the loop starts. `DefaultTickRate` is 30. A requested rate below `MinimumTickRate` (5) or above `MaximumTickRate` (128) is clamped into range and logged, not rejected.
- **`ServerConfiguration.MaximumConnections`** is read-only to your code. Change it with `transport.SetMaximumConnections(uint)`, and Synapse applies the new cap the next time the server starts.

The step provider is not on this list. `NetworkLoopManager.UseNetworkLoopStepProvider(INetworkLoopStepProvider)` swaps it at runtime, stopping the previous provider and returning it. To use your own provider from the start, pass it to the `CoreManager` constructor instead: swapped in after construction, it replaces a default provider that has already been stepping the loop from the thread pool.

SystemManager

Setting	Default	Notes
`StateInterpolation`	0	Capped at `MaximumStateInterpolation` (7) when applied.
`GlobalRedundancy`	0	Only usable when the transport's default channel is unreliable. Capped at `StateInterpolation` on the server's sends.
`ClientRedundancy`	3	Extra resends a pure client adds to everything it sends upstream, on top of `GlobalRedundancy`. Not capped.
`StateRetentionMilliseconds`	1000	Capped at `MaximumStateRetentionMilliseconds` (2000).
`PartialStateApplyEnabled`	true	A tick that loses some of its packets applies the ones that arrived. Off, the whole tick is discarded and recovered. The receiver's choice alone.

`PredictionHistoryTicks`	`UnsetPredictionHistoryTicks` (0)	How many ticks of history `Predict` and `Validate` members keep. Unset, every client keeps `DefaultPredictionHistoryTicks` (64) and the server keeps two.
`MaximumReplayTicks`	`UnsetMaximumReplayTicks` (0)	Most ticks a reconcile replays; a longer one lands the server's values at its start tick without replaying. 0 sets no limit.
`ReplayHoldRadius`	`UnsetReplayHoldRadius` (0)	Client: only systems it does not control within this distance of a restored system are held during a replay. 0 holds every one.
`IsResultValueSendEnabled`	true	Read on the controller: sends `Validate` result values beside the hash so `NetworkSystem.ResultMismatched` can name what differed. Off keeps the hash alone.
`DefaultControllerRetentionPolicy`	`ControllerRetentionPolicy.Release`	Resolved most specific first: a system's own override, then its group's, then this.
`ControllerRetentionSeconds`	120 (`DefaultControllerRetentionSeconds`)	`UnlimitedControllerRetentionSeconds` (0) removes the limit.
`MaximumRetainedControllerRecords`	64 (`DefaultMaximumRetainedControllerRecords`)	
`HostMigrationEnabled`	false	

InterestManager

Setting	Default	Notes
`EvaluationCadenceTicks`	5 (`DefaultEvaluationCadenceTicks`)	How often, in ticks, a system's interest is re-evaluated per connection.
`MaximumControlledInterestObjects`	1 (`DefaultMaximumControlledInterestObjects`)	`UnlimitedControlledInterestObjects` (0) measures against every controlled object.
`MaximumSpawnsPerTick`	500 (`DefaultMaximumSpawnsPerTick`)	Pro only. Caps how many spawns a peer is admitted per tick; the rest queue and drain over following ticks.

SceneManager

Setting	Default	Notes
`JoinPlacement`	`JoinScenePlacement.EveryOpenScene`	Where a newly authenticated client is placed.
`LoadRequestTimeoutSeconds`	180	A value of 0 or less disables the timeout.
`AutomaticRequestOnBlockedSpawnEnabled`	false	
`SceneCarryFailure`	`SceneCarryFailureAction.SurrenderControl`	The world-wide answer; overridable per object via `NetworkSystem.SceneCarryFailureOverride` and per case via a registered `ISceneCarryFailureResolver`.

RpcManager

Setting	Default	Notes
`MaximumInboundRpcsPerConnectionPerDrain` (`DefaultMaximumInboundRpcsPerConnectionPerDrain`)	64	`UnsetMaximumInboundRpcsPerConnectionPerDrain` (0) removes the bound.

ServerManager

Setting	Default	Notes
`AuthenticationTimeoutSeconds`	30	`UnsetAuthenticationTimeout` (0) disables the timeout; a connecting client that never authenticates is disconnected after this.
`KickTimeoutMilliseconds`	500	How long a kicked or refused client's link may stay open after its kick before it is closed outright. Overloads that take a `timeoutMilliseconds` override it for one kick.

ClientManager

Setting	Default	Notes
`DisconnectResetMode`	`DisconnectResetMode.ClearReceivedWorld`	What the client keeps of its received world on disconnect.

ViolationManager

Setting	Default	Notes

`DefaultAction`	`ViolationAction.Ignore`	A constant. Used only when a violation type is raised with no default of its own and no handler is registered; `PacketTransformRejectedViolation` (Pro) is the one such type, and that raise always logs a warning. Every other type carries its own default, mostly `Ignore`.

TransportManager

Setting	Default	Notes
`DefaultChannel`	`Channel.Unreliable`	
`HostLoopbackDelivery`	`HostLoopbackDelivery.Immediate`	Settable only to `Immediate`; any other value is refused and logged. The alternative exists only for the engine's own tests.
`RoundTripTimeIntervalMilliseconds`	`1000` (`DefaultRoundTripTimeIntervalMilliseconds`)	0 disables round-trip time measurement.

BundleManager

Pro only.

Setting	Default	Notes
`AutomaticRequestOnBlockedSpawnEnabled`	`true`	A blocked spawn also asks the client for the bundle it needs, so content hotloads on demand.

Per-transport structs

`Transport.Configuration` and `Transport.ServerConfiguration` hold settings specific to a transport rather than to a manager.

`Configuration`

Field	Default	Notes
`Port`	0	
`MaximumTransmissionUnit`	1200	Packets longer than this are split into multiple packets.
`ConnectingTimeoutSeconds`	10	How long a client waits for the server to answer its connect. Zero falls back to `ConnectedTimeoutSeconds`.

`ConnectedTimeoutSeconds`	15	Least applied value is one second. Times out a connection that has received no remote data within this window.

`ServerConfiguration`

Field	Default	Notes
`MaximumConnections`	`UnsetMaximumConnections` (0)	Accepts any number at 0. Set it with `transport.SetMaximumConnections(uint)`; Synapse applies it at the next server start.

Local vs. shared values

Most manager settings are purely local: a server and its clients can each run different `StateInterpolation`, `EvaluationCadenceTicks`, timeout, or retention values without breaking the connection, because each peer only governs its own side of the exchange.

Tick rate is the exception that must match. A server and client running different `TickRate` values are ticking at different real-world speeds, which the replication and interpolation built on top of the tick assume does not happen.

Editor menus and authoring tools

The menu commands

Nucleus adds four items under the `Nucleus` menu. Three are build steps over the project's assets, and the fourth checks those assets without changing them; none is a runtime action. A fifth item, **Window > Nucleus > Network Inspector**, opens a play-mode readout of the running session (see Developer tools in the editor).

Nucleus/Rebuild Network Prefab Collection

Scans the project for every prefab whose root carries a `NetworkSystemObject`, groups them by the asset bundle they're assigned to, and writes each group's `NetworkPrefabCollection` asset. The collection with no asset bundle assignment is the base build's shard, stored under `Assets/Resources/NetworkPrefabCollection.asset` so it loads by name at runtime.

Each prefab is stamped with a local prefab id and a bundle id. Ids are assigned once and then preserved: a prefab that already carries a valid, unique id keeps it on every rescan. Only an unset or duplicated id gets the next free number for its shard. Ids restart at one per shard, because the runtime registry resolves a prefab by the `(bundle, local)` pair, not by the local id alone.

A rebuild that renumbers a duplicated id says so: it logs a warning naming the prefab that carried the id, the prefab that already holds it, and the new id. It logs an error when two collections resolve to the same prefab bundle, because the second of them to load is refused at runtime; give each content bundle assembly its own `[NetworkBundle(n)]`.

Nucleus/Validate Network Prefabs

Checks every prefab in the project and logs each problem against the prefab it names, without changing anything: the prefab identity problems a player build refuses (below), a `NetworkSystemObject` on a prefab's child, a networked component with no `NetworkSystemObject` at or above it, and a `NetworkHitboxRoot` that would record nothing. It runs only on request, because it loads every prefab. Developer tools in the editor lists each check.

Nucleus/Rebuild Network Scene Manifest

Scans the scenes enabled in Build Settings (and, with the Addressables package installed, scenes marked Addressable) and records each one's scene id, path and content bundle in the project's `NetworkSceneManifest`, at `Assets/Resources/NetworkSceneManifest.asset`. A scene keeps its id across the rescan; only a scene the manifest has never seen gets a new one, assigned from the highest id already recorded. Removing a scene from the build leaves its manifest entry in place rather than deleting it, so its id is never handed to a different scene later.

Nucleus/Build Content Bundles

Runs the prefab collection rebuild and the scene manifest rebuild first, then builds every asset bundle a `NetworkPrefabCollection` is assigned to into `Assets/StreamingAssets`, and fills in `NetworkBundleManifest` with each built bundle's id and file name. Running the two rebuilds first matters: a scene or prefab records the bundle id it ships under by reading the collection assigned to that bundle, so building from stale identity is what bakes the wrong id into a shipped bundle.

The automatic steps

Three more steps run without a menu command, so an author never has to remember to invoke them.

`NetworkPrefabBuildCheck` runs before every player build. It makes the same identity check as Validate Network Prefabs, and fails the build when any networked prefab does not carry the identity a rebuild would give it: no base collection, two collections sharing a prefab bundle, a prefab missing from its collection, a prefab with no id or with an id another prefab holds, or a prefab stamped with a different bundle than its collection. The build error lists every problem and tells you to run **Nucleus > Rebuild Network Prefab Collection**, then build again. It stops the build rather than repairing the prefabs itself, because a repair made in the middle of a build could number a server and a client that were built apart differently.

`SceneNetworkObjectStamper` runs on scene save. It assigns a `SceneObjectId` to any `NetworkSystemObject` in the saved scene whose id is unset or collides with another object in the same scene, keeping the first occurrence of any value it finds and handing the next free incremental id to the rest. Duplicating an object that already carries a stamped id is exactly what creates a collision, since the copy carries the same serialized id until the next save reassigns it.

`NetworkSceneManifestAutoRebuild` mints a scene id the moment a scene enters the build: when the Build Settings scene list changes, when a scene asset is imported, moved or deleted, when the project's Addressables entries change, and as a backstop right before a player build. It calls the same reconciliation the manual scene manifest rebuild uses, quietly.

When to re-run each command

Change	Command to run
Added, renamed, moved or duplicated a networked prefab	Rebuild Network Prefab Collection
Added or removed a scene from the build	Nothing to run manually — the auto-rebuild mints the scene id on its own; run the manual command only if you want the log line confirming what mapped
Changed an asset bundle assignment (prefab or scene)	Rebuild Network Prefab Collection, then Build Content Bundles
Ready to ship the built asset bundles	Build Content Bundles (this also re-runs the prefab and scene rebuilds first)

Two authoring traps

A copied prefab carries the original's stamped id. Duplicating a prefab asset in the Project window duplicates its `NetworkSystemObject` component, id and all. Until the next `Rebuild Network Prefab Collection`, the copy has the same prefab id as the original. At runtime the registry keeps whichever of the two registered first and logs an error for the other, so a spawn of one can build the other, and a player build fails until you rebuild. Rebuild after every prefab duplication, not just after adding a new prefab. The rebuild keeps the id on the prefab the collection already lists, which is the original, and on a base over its own variant, gives the copy the next free id, and logs a warning for each prefab it renumbers.

A scene object id only has to be unique within its own scene. `SceneNetworkObjectStamper` reassigns collisions inside one scene, but the same id in two different scenes is fine and expected — the scene half of the correlation comes from the scene's own id, not from the object's. Don't assume a scene object id has to be globally unique across the project; it doesn't.

Why every Nucleus inspector still shows the script row

A custom `[CustomEditor]` replaces Unity's whole default inspector, including the greyed-out script row that lets you ping or open the component's source. `NucleusInspectorGUI.DrawScriptField` redraws that row explicitly, and every Nucleus custom editor calls it before drawing anything else, so that row survives even though the rest of the inspector is custom-drawn. `NucleusInspectorGUI` also provides `DrawPropertiesWithoutScript` (the rest of the default inspector, minus the script row already drawn), plus two field-clamping helpers (`DrawClampedIntSlider`, `DrawNonNegativeLongField`) used by inspectors that need to correct an out-of-range serialized value without flattening a multi-object selection.

`NetworkSceneReferenceDrawer` is a property drawer, not a component inspector: it draws a `NetworkSceneReference` field as a scene-asset picker, with the scene id it currently resolves to (from the project's scene manifest) shown read-only beside it. Nothing about the id is stored on the reference itself — only the scene's GUID is — so the readout always reflects the current manifest rather than a baked-in number.

The custom editors

Twenty-seven components have a `[CustomEditor]`, which is where to look for authoring-time validation and inspector-only readouts:

- `NucleusBehaviourBaseEditor` — covers every `NucleusBehaviourBase` subclass (`editorForChildClasses: true`); adds a runtime readout (linked system, started state, id, state, control) while playing.
- `UnitySceneLoaderEditor` — covers every `UnitySceneLoader` subclass; validates the assigned scene manifest and warns when none is set.
- `UnityAssetBundleLoaderEditor` — covers every `UnityAssetBundleLoader` subclass; validates the assigned bundle manifest and warns when none is set.
- `NetworkSystemObjectEditor`
- `NetworkTransformEditor`
- `NetworkAnimatorEditor`
- `NetworkNavMeshAgentEditor`
- `NetworkHostVisibilityEditor`
- `NetworkInterestObjectEditor`
- `NetworkPlayerSpawnerEditor`
- `NetworkCorrectionVisualizerEditor`: while playing, adds the flash count and the size of the last correction.
- `NetworkInterestOverlayEditor`: shows the overlay's options only while it is switched on, with a key to its colors.
- `NetworkHitboxEditor`
- `NetworkHitboxRootEditor`
- `UnityClientManagerEditor`
- `UnityServerManagerEditor`
- `UnityTransportManagerEditor`
- `UnityInterestManagerEditor`
- `UnityLagCompensationManagerEditor`
- `UnityNetworkLoopManagerEditor`

- UnityPhysicsManagerEditor
- UnitySceneManagerEditor
- UnitySystemManagerEditor
- PhysicsSimulationDriverEditor
- ProjectedRigidbodyEditor
- BlitzRelayTransportEditor
- SynapseTransportEditor

FishNet to Nucleus name map

Read this first

This table shows where FishNet concepts and Nucleus concepts line up. It is not an alias list and it does not import FishNet names into Nucleus. Nucleus has its own vocabulary; write Nucleus names in Nucleus code, not FishNet ones translated in place. Some rows also mark where the shapes genuinely differ, not just the names.

Objects and scripts

FishNet	Nucleus
`NetworkObject`	`NetworkSystem` in the core, `NetworkSystemObject` in the Unity integration
`NetworkBehaviour`	`NucleusBehaviour<T>` (Unity), layered over `MonoBehaviour` plus `NetworkSystemObject` plus `NetworkComponent`

These are not one-to-one. A FishNet `NetworkObject` is one networked identity per `GameObject`. A Nucleus `NetworkSystem` is not tied to a `GameObject` at all in the core, and in Unity one `GameObject`'s `NetworkSystemObject` can link several `NetworkSystems` at once, each carrying its own `NetworkComponent`.

Replicated state

FishNet	Nucleus
`SyncVar<T>`	`NetworkMember<T0>`
`SyncList<T>`	`NetworkListMember<T0>`
`SyncDictionary<TKey, TValue>`	`NetworkDictionaryMember<T0, T1>`
`SyncHashSet<T>`	`NetworkHashSetMember<T0>`
(no FishNet counterpart)	`NetworkArrayMember<T0>`
(no FishNet counterpart)	`NetworkQueueMember<T0>`
`OnChange` (per-member callback)	`NetworkComponent.OnMembersChanged(ulong memberFlags, MemberChangeDirection memberChangeDirection)`

`OnMembersChanged` is component-wide, not per-member: it fires once per component with a flag mask of which members changed, and a `MemberChangeDirection` (`Write` or `Read`) saying which side of the wire the raise is reporting. There is no per-member changed event; this is a different shape, not a missing feature.

Ownership and calls

FishNet	Nucleus

`Owner` / `IsOwner`	`ControllingClient` / `IsController(ControllerType.Client)`
`GiveOwnership(connection)`	`SetController(connection)`
`RemoveOwnership()`	`SetController(null)`
`[ServerRpc]` / `[ObserversRpc]` / `[TargetRpc]`	an `IRpc` struct sent with `TrySendRpc<T0>(RpcTarget, Channel, T0, RpcSelfDelivery)` and an `RpcTarget` (`RpcTarget.Server`, `RpcTarget.Observers`, `RpcTarget.ObserversExcept(...)`, `RpcTarget.To(connection)`)
`RequireOwnership`	`RpcSendAccess`
`IBroadcast`	`IMessage`

There are no attribute-driven RPCs in Nucleus. A remote call is a plain struct implementing `IRpc`, sent explicitly through `TrySendRpc` with a target and channel, rather than a method decorated with an attribute. That is a design choice, not a gap.

`ControllerType` also has a `Server` and an `AnyController` value alongside `Client`, for checking whether the local peer is the server, the controlling client, or either.

Managers and infrastructure

FishNet	Nucleus
`TimeManager`	`NetworkLoopManager`
`InstanceFinder`	`CoreManager.Instance` (core) or `NucleusUnity.BoundCoreManager` (Unity)
`ObserverCondition`	`IInterestCondition`
`Authenticator`	`IClientAuthenticator`
`NetworkConnection`	`Connection`
`ClientId`	`Connection.Id`
kicking a client	`ServerManager.KickClient(connection)`

`CoreManager.Instance` is a real static slot, set the first time a `CoreManager` is constructed and guarded by `EnsureSetInstance` for a process that wants to name a different one as the instance later. `NucleusUnity.BoundCoreManager` is the Unity-side equivalent, bound by the integration's bootstrap.

Both slots name exactly one manager. A process running more than one `CoreManager` at a time — a host pair in the same process, a two-peer test — has more than one valid manager and no slot can name both. In that situation, resolve and pass the `CoreManager` explicitly instead of reading the static.

These names are the same in both frameworks

These read the same in Nucleus as in FishNet; no rename to look up.

- `Channel.Reliable`, `Channel.Unreliable`
- `IsHostStarted`, on a `NucleusBehaviour` as well as on `TransportManager` and `NetworkSystem`

`IsServerStarted` and `IsClientStarted` keep their names on `TransportManager` and `NetworkSystem`, but a `NucleusBehaviour` has no property by either name. Inside a script, ask `IsStarted(Invoker.Server)` or `IsStarted(Invoker.Client)` instead.

There is no automated migration tool

There is no automated FishNet-to-Nucleus converter. Port code by hand, using this map to find the replacement concept, then write it in Nucleus's own shape rather than FishNet's.

Reference & Project Meta

These pages are for looking things up: type and component references, configuration, and compatibility.

- API type reference
- Compatibility
- Component Index
- Configuration reference
- Editor menus and authoring tools
- Versioning, upgrading and release notes
- Installing and updating Nucleus in Unity
- FishNet to Nucleus name map
- Performance and benchmarks
- Project assets and loaders
- CodeBoost reference

Installing and updating Nucleus in Unity

Driving the core API directly? See Adding Nucleus to a .NET project.

What's in the package

Nucleus for Unity ships as prebuilt DLLs plus source for its integration layer. The distributed build unpacks into an `Assets/Nucleus` folder:

- `Nucleus.dll`, `CodeBoost.dll`, `SynapseSocket.dll` — each with a `.pdb` alongside it for debuggable stack traces.
- `Nucleus.CodeAnalysis.SourceGenerators.dll` — the Roslyn source generator that makes networked types compile. No `.pdb` ships for it.
- `Nucleus.xml` and `SynapseSocket.xml` beside their DLLs: the XML documentation for the public API, so your IDE shows hover docs for every engine type and member you can use.
- `LICENSE.txt`, and `VERSION.txt`, which names the release (see Identifying what you're running).
- An `Integrations` folder holding four source trees, each its own assembly definition: `Unity`, `Blitz Relay`, `Newfarm Client`, `Newfarm Migration`.

Install it either by importing the release's `.unitypackage` (**Assets > Import Package > Custom Package**) or by copying the whole `Assets/Nucleus` folder into your project. Those are the two supported methods: there is no UPM package, no Asset Store listing, and no git-URL install.

The one rule that decides whether it works

Replace the files. Never replace the `.meta` files.

A project that already has this folder has its own Unity-generated `.meta` for every path in it, and those `.meta`s carry state Unity needs: asset GUIDs, and — for the source generator — the label that makes it run at all. `Nucleus.CodeAnalysis.SourceGenerators.dll.meta` carries:

```
labels:
- RoslynAnalyzer
```

That label is what tells Unity to run the DLL as a Roslyn source generator instead of referencing it as a normal plugin. Overwrite that `.meta` with the one from the new build — or delete it and let Unity regenerate a plain one — and the generator stops running. Nothing shows an error for this. The project compiles, the components look fine, and nothing replicates, because the generated networking code was never produced. If replication silently stops after an update, check this first: open `Nucleus.CodeAnalysis.SourceGenerators.dll.meta` and confirm `RoslynAnalyzer` is still listed under `labels`.

All four DLLs come from one build

`Nucleus.dll`, `CodeBoost.dll`, `SynapseSocket.dll`, and `Nucleus.CodeAnalysis.SourceGenerators.dll` must all be from the same build output. Mixing them — a fresh `Nucleus.dll` next to a stale `CodeBoost.dll`, for example — does not fail to compile. The two DLLs disagree about what's inside each other at runtime, and that surfaces as a `MissingMethodException` when the mismatched call actually runs, not as a build error. When updating, replace all four together.

Unity floor and assembly layout

The integration is gated behind:

```
#if UNITY_EDITOR || UNITY_2021_3_OR_NEWER
#define UNITY_ENGINE
#endif
```

Unity 2021.3 is the effective minimum for a build target — older versions never define `UNITY_ENGINE`, so the integration code doesn't compile in.

`Nucleus.Integrations.Unity.asmdef` is the main integration assembly. It sits beside two sibling assemblies for the same code: `Nucleus.Integrations.Unity.Editor.asmdef` under `Editor/`, and `Nucleus.Integrations.Unity.Tests.asmdef` under `Tests/`.

Three scripting defines control optional pieces:

- `NUCLEUS_ADDRESSABLES` — a `versionDefines` entry keyed to `com.unity.addressables` version `1.19` and up. Set automatically once that package is present at a matching version; nothing to do by hand.
- `NUCLEUS_NAVMESH` — a `versionDefines` entry keyed to `com.unity.modules.ai`, Unity's AI module. Set automatically while the module is enabled, as it is in a new project. With the module disabled, `NetworkNavMeshAgent` is left out and the rest of the integration still compiles.
- `BLITZ_RELAY` — a `defineConstraints` gate on the Blitz Relay assembly itself. The relay transport only compiles when this define is set in your project; the main Unity integration assembly references it unconditionally but the relay code stays out until you opt in.

Updating

Replace the four DLLs and their `.pdb` and `.xml` files, `LICENSE.txt`, `VERSION.txt`, and the `Integrations` source trees with the new build's copies. Leave every `.meta` file in the project untouched; they're already there from your first install, and they're what keeps the source generator running and the asset GUIDs stable.

If the editor is in Play mode when you replace the files, Unity reimports and triggers a domain reload mid-play. Expect play mode to reset; don't replace files while relying on play-mode state you haven't saved.

Identifying what you're running

`Assets/Nucleus/VERSION.txt` names the installed release (it reads `Nucleus <version>`), and a built player writes `Thank you for using Nucleus Networking <version>.` to its log at launch. The DLLs carry no version number of their own. See Versioning, upgrading and release notes.

Not using Unity?

For a dedicated server or any other non-Unity host, see Adding Nucleus to a .NET project.

Performance and benchmarks

Every figure here comes from a test against the real engine

The benchmark suites live in `Nucleus.Tests/Benchmarks` and write through the real serializer, the real interest system, and the real spawn/controller paths - not a stand-in model of them. A change that regresses wire cost or per-object overhead fails a test in that project; it does not require someone to notice a number drift in a document. Treat any figure quoted for Nucleus as only as trustworthy as the suite that produced it.

What each suite measures

Wire cost (what a tick actually serializes onto the wire):

Suite	Measures
`BandwidthTests`	Baseline movement and rotation cost, `TransmissionMode.Interval` versus `TransmissionMode.Divine`, in bytes per send.
`NetworkSystemDeltaBandwidthTests`	Delta cost at the `NetworkSystem` level rather than a single member.
`DivineBeltBandwidthTests`	Divine transmission under sustained, varied motion (a belt of moving objects), not just a single synthetic delta.
`InterestPacingBandwidthTests`	Cost as interest pacing changes what is sent and to whom.
`OrdinaryWorldWireCostTests`	Steady-state cost of an ordinary populated world, as a whole-scene baseline.
`PackingBandwidthTests`	Cost of the bit-packing layer across representative value ranges.
`QuaternionPackingBenchmarkTests`	Cost of packed rotation specifically.

Resolution and per-object cost (what it costs the server to decide what to send, independent of wire size):

Suite	Measures
`InterestResolutionBenchmarkTests`	Cost of resolving interest across a population of objects and observers.
`ObserverWalkBenchmarkTests`	Cost of walking an object's observer set.
`NetworkComponentBenchmarkTests`	Per-component overhead on a `NetworkSystem`.
`ControllerReleaseBenchmarkTests`	Cost of releasing a controller.
`RpcDispatchBenchmarkTests`	Cost of dispatching a received remote call to its handlers as the number of objects handling the same call grows.

Running the suites and reading a result

Run `Nucleus.Tests` in **Release** . Release is the only configuration that runs the optimized code path; a

Debug run measures the JIT's debug build, not the engine you ship. Each test logs its own figure (for example, `BandwidthTests` logs bytes per send via `Logger<BandwidthTests>`) and most also assert a bound, so a regression fails the run rather than only showing up in the log.

```
dotnet test Nucleus.sln -c Release --filter FullyQualifiedName~Nucleus.Tests.Benchmarks
```

Read the logged number together with the scenario that produced it - tick rate, object count, and transmission mode are part of the result, not decoration on it.

Runtime counters

`TransportManager` exposes two counter families that answer different questions and should never be compared to each other directly:

- `TotalBytesSent` and `TotalBytesReceived` count every byte that crossed the transport, including headers, framing, and every redundant resend or arrival.
- `TotalStatePacketPayloadBitsSent` and `TotalStatePacketPayloadBitsReceived` count only serialized StatePacket payload bits, with headers and framing stripped out, and each serialized payload counted once no matter how many peers it reached.

The byte counters answer "what did the link carry." The bit counters answer "what did replication actually cost." They do not reconcile, and a feature that fans one serialized payload out to many peers or resends over an unreliable link will always move the byte counters faster than the bit counters. See Network statistics for the full counter reference.

Per-connection measurement

Each `Connection` tracks its own link quality:

- `RoundTripTimeMilliseconds` and `RoundTripTimeDeviationMilliseconds` - a smoothed round trip time and its spread. Both read as `UnsetRoundTripTimeMilliseconds` / `UnsetRoundTripTimeDeviationMilliseconds` until the first reply arrives, so check `IsRoundTripTimeUnset` before trusting either.
- `PacketLossPercentage` - the share of recent probes that went unanswered, measured the way a ping tool measures it.

A server does not re-measure a remote client's link itself once bootstrapped; it adopts the figures the client reports (`ReportedRoundTripTimeMilliseconds`, `ReportedRoundTripTimeDeviationMilliseconds`) into its own `RoundTripTimeMilliseconds` and `RoundTripTimeDeviationMilliseconds`. A misbehaving client only mishandles its own recovery by misreporting; the server never judges another peer by what one client reports.

Quoting a figure

A bandwidth number is meaningless without its world: tick rate, object count, transmission mode, edition, and transport all change the result, so state all five whenever a figure leaves the test log. A Free build costs more than Pro on plain replication - Pro packs values more tightly - but that difference is a black box here; this page does not explain how the packing works.

Project assets and loaders

Prefabs

NetworkPrefabCollection

A `ScriptableObject` (`Nucleus/Network Prefab Collection` in the Create menu) that lists the networked prefabs shipped with one content shard. The base build ships a default shard, and each asset bundle ships its own alongside the prefabs it carries.

- `_prefabs` — the prefabs in the shard, exposed as `Prefabs`.
- `_prefabBundleId` — the bundle identifier this shard registers its prefabs under; zero is the base build's default shard. Exposed as `PrefabBundleId`, stamped by the editor scan from the shard assembly's `[NetworkBundle]` attribute.
- `_bundleAssemblyName` — the simple name of the assembly whose `[NetworkBundle]` attribute supplies that identifier; empty for the base build's default shard. Exposed as `BundleAssemblyName`.

`RegisterAll()` claims the shard's bundle identifier and registers every prefab; `UnregisterAll()` removes them and releases the claim, called as the shard's content unloads. Two loaded collections claiming the same bundle identifier is an error, not a silent overwrite.

The base build's collection is loaded from `Resources/NetworkPrefabCollection` automatically at startup. A project with no prefab collection there registers nothing, and a dynamic spawn for a prefab identity nobody registered fails to resolve on the receiving peer.

NetworkPrefabRegistry

The runtime lookup a collection registers into, keyed by `(BundleId, LocalId)`. A shard registers under its own bundle id, so separately built content never collides. `NetworkPrefabRegistry.TryGet` is what a spawn resolves its prefab through; a spawn whose identity has no entry cannot instantiate.

The prefab pool seam

Every rent and return of a dynamic spawn's GameObject goes through `INetworkPrefabPool`:

```
public interface INetworkPrefabPool
{
    GameObject Rent(GameObject prefabGameObject, ushort prefabBundleId, ushort prefabLocal
Id, Vector3 position, Quaternion rotation, Vector3 scale);
    void Return(GameObject pooledGameObject, ushort prefabBundleId, ushort prefabLocalId);
}
```

Two implementations ship:

- **DefaultNetworkPrefabPool** — instantiates on rent and destroys on return. No pooling, and the default a project starts with.
- **NetworkPrefabPool** — recycles: a returned instance is deactivated and parked under a shared inactive root, keyed by prefab identity, and a later rent reactivates the longest-rested parked instance instead of instantiating. `ReturnCooldownSeconds` keeps a just-returned instance parked before it is eligible for reuse; `InstantiatedCount` and `ReusedCount` report the cold-miss and recycle counts.

Because a recycled instance does not re-run `Awake`, a script needing per-life setup binds through the marker's `SystemLinked`/`SystemUnlinked` events, or initializes in `OnEnable` instead.

The active pool is assigned through `NucleusUnity.PrefabPool`:

```
NucleusUnity.PrefabPool = new NetworkPrefabPool { ReturnCooldownSeconds = 1f };
```

Assigning `null` restores `DefaultNetworkPrefabPool`. A project that never touches this property keeps the default, instantiate-and-destroy behavior.

Scenes

NetworkSceneManifest

A `ScriptableObject` (`Nucleus/Network Scene Manifest` in the Create menu) mapping the scene identifiers the server names on the wire to the scene assets a peer loads them from, and the content bundle each scene ships in. It is also the identifier registry itself: a prefab is stamped on the prefab asset, but a scene has no comparable place to carry one, so the entry keyed by the scene's GUID is where the identifier lives.

`_sceneEntries` holds the list, exposed as `Entries`; each entry is a `SceneEntry`:

Field	Purpose
`SceneId`	The identifier the server names this scene by on the wire.
`SceneGuid`	The scene asset's GUID — what the identifier is really keyed to.
`ScenePath`	The scene asset's path, handed to Unity's loader.
`RequiredBundleId`	The content bundle the scene ships in; the base build's identifier means it ships in the player.
`DeliveryMode`	How the peer gets the scene (`SceneDeliveryMode`), derived from the project by the build step.

A project has one manifest, loaded by convention from `Resources/NetworkSceneManifest` (`NetworkSceneManifest.LoadProjectManifest()`), so nothing has to be dragged into an inspector.

NetworkSceneReference

A struct field type that names a networked scene by dragging the scene asset in, instead of typing its wire number by hand. It stores only the scene's **GUID** (`_sceneGuid`) — never the scene identifier itself — so renaming or moving the scene asset never repoints the reference or silently starts naming a different scene. The identifier is resolved through the project's manifest at the moment it's used (`TryGetSceneId`), so a scene added after the reference was authored resolves as soon as the manifest is rebuilt.

Handle-to-Scene lookups

Once a scene instance is loading, code resolves it through the `UnitySceneManager` component (the CoreManager's Unity-side scene driver), not by walking Unity's own `SceneManager` by name or path — several instances of the same scene asset can be loaded at once, and `GetSceneByName`/`GetSceneByPath` cannot tell them apart.

```
public bool TryGetScene(uint sceneHandle, out Scene scene);
public bool TryGetSceneHandle(Scene scene, out uint sceneHandle);
```

`TryGetScene` turns the server's scene instance handle into the live Unity `Scene`; it answers false while the scene is still loading. `TryGetSceneHandle` is the inverse, for code holding a scene or object that needs to know which instance it belongs to — a true result with `NetworkSystem.UnsetSceneHandle` means the

peer loaded that scene itself and no instance will ever claim it.

The correlation between a handle and its `Scene` is tracked internally; these two methods are the supported way to read it.

NetworkSceneLoader and UnitySceneLoader

`NetworkSceneLoader` is the abstract `MonoBehaviour` base for loading networked scenes: it implements `ISceneLoader` and registers itself as the peer's scene loader automatically on `Awake` (toggle `_automaticRegistrationEnabled` off to call `Initialize` explicitly). Loading and unloading are left to the derived class via `LoadSceneAsync`/`UnloadSceneAsync`.

`UnitySceneLoader` is the shipped implementation, built on Unity's own additive scene loading. It resolves a scene through a manifest, takes the scene's content bundle first if it needs one, then loads the scene additively.

- `_sceneManifest` — the manifest this loader reads. Left empty, it falls back to the project's own manifest, so nothing needs assigning for the common case.
- `_localPhysicsEnabled` — loads each scene into its own local physics world (`LocalPhysicsMode.Physics3D`) with a `PhysicsSimulationDriver` stepping it each tick, so a stacked scene's physics runs independently of the default world.

A project doesn't have to place a `UnitySceneLoader` itself: `UnitySceneManager` adds one automatically if no `NetworkSceneLoader` of any kind is present in the loaded scenes.

Content bundles

`NetworkBundleManifest` ships in every build. `NetworkBundleLoader` and `UnityAssetBundleLoader` are Pro-only files — absent entirely from a Free build — so a Free project can carry a bundle manifest but has nothing built in that loads against it.

NetworkBundleManifest

A `ScriptableObject` (`Nucleus/Network Bundle Manifest` in the Create menu) mapping the bundle identifiers the server names on the wire to the asset bundle files a client loads them from.

- `_bundleEntries` — the mapped bundles, exposed as `Entries`; each `BundleEntry` pairs a `BundleId` with a `FileName`.
- `_rootDirectory` — the directory file names resolve against; empty falls back to `Application.streamingAssetsPath`.

NetworkBundleLoader and UnityAssetBundleLoader (Pro)

`NetworkBundleLoader` is the abstract `MonoBehaviour` base for loading content bundles: it implements `IBundleLoader` and self-registers on `Awake`, the same way `NetworkSceneLoader` does for scenes. `LoadBundleAsync`/`UnloadBundleAsync` are left to the derived class.

`UnityAssetBundleLoader` is the shipped implementation over Unity asset bundles. Its `_bundleManifest` field maps bundle identifiers to files; left unassigned, a load request logs an error and fails rather than resolving anything. Once a bundle loads, it registers every `NetworkPrefabCollection` the bundle ships, so spawns waiting on that content can land.

What a project actually needs

A project with only base-build prefabs and scenes needs:

- One `NetworkPrefabCollection` at `Resources/NetworkPrefabCollection` (registers automatically).

- One `NetworkSceneManifest` at `Resources/NetworkSceneManifest`, if it uses networked scenes at all.

Everything else is optional:

- **Prefab pool** — unassigned, spawns use `DefaultNetworkPrefabPool` (instantiate/destroy). Assign a `NetworkPrefabPool` through `NucleusUnity.PrefabPool` only to recycle instances.
- **Scene loader** — a project doesn't have to add one; `UnitySceneManager` provides a `UnitySceneLoader` automatically. Add one manually only to override the loading pipeline.
- **Content bundles** — only needed once the project ships prefabs or scenes in asset bundles rather than the base build, and only Pro builds have a shipped loader for them. A `UnityAssetBundleLoader` with no manifest assigned fails every bundle load with a logged error, rather than silently doing nothing.

Reporting a bug

Bugs go to the issue tracker. The form there asks for your version, edition, transport and topology, because those four facts decide which code path ran.

This page covers the part the form cannot do for you, which is the project you attach so that someone else can watch the fault happen.

A reproduction project is what gets a bug fixed

Nucleus faults are usually timing faults. A description of what you saw narrows the search a little, but a project that reproduces it narrows the search to a single tick. Most reports that go unfixed were never ignored; they simply could not be reproduced.

An issue without a reproduction project is investigated at lower priority, and it may be closed if it cannot be reproduced.

Every attachment must be under 1 MB

Zip your project and keep the archive under 1 MB. Attachments over 1 MB are deleted, and there are no exceptions.

That limit sounds tight, and it is not. A Unity project carrying one scene, a handful of prefabs and the Nucleus package fits comfortably once the regenerated folders are gone, and every reproduction project can be reduced to that.

Delete these folders before zipping, because Unity rebuilds every one of them the first time the project is opened:

- `Library/` holds the imported asset cache, and it is usually most of the archive on its own.
- `Temp/` and `obj/` hold intermediate build files.
- `Build/` holds any player you have built from the project.
- `Logs/` holds the editor logs, which belong pasted into the issue rather than zipped.
- `UserSettings/` holds your personal editor layout, which nobody else needs.

The project should contain only what the bug needs

- **Remove every third-party asset.** Replace anything from the Asset Store with primitives and untextured materials, because a project that cannot be opened without buying something cannot be used to investigate your bug.
- **Leave the Nucleus package in place, exactly as you have it installed.** Which build you are running is part of the report, so do not strip it out to save space.
- **Reduce the project to one scene showing one fault.** If two things are wrong, please open two issues.
- **Name the steps precisely.** Say which scene to open, what to press, and what to watch for.

Say these things in the issue itself

- **Describe the topology the fault needs.** Say whether it takes a host or a dedicated server plus a client, and how many peers have to be running. A large class of faults disappears the moment the same code runs as a host, and the reverse is true just as often.
- **Say how often it happens.** "Every time" and "roughly one run in four" lead to completely different investigations. An intermittent fault is still worth reporting, as long as the report says that it is intermittent.

- **Say whether it also happens in a build**, or only in the editor.
- **Mention any simulated packet loss** that you had switched on, and give the percentage.

Attachments larger than 1 MB are deleted

If your zipped project is still over 1 MB after you have deleted the regenerated folders, then the extra size is coming from something in `Assets/` that the bug does not need. Find it and remove it. Cutting a project down to the smallest thing that still shows the fault is useful work in its own right, because it tells you which parts of your setup the fault actually depends on.

Please do not attach an oversized project and explain the size in the issue instead. An attachment larger than 1 MB is deleted, and while deleting it does not close your issue, it does leave that issue without a reproduction project. An issue without one is investigated at lower priority, and it may be closed if it cannot be reproduced.

Do not send your whole game

A full project is slower to investigate than no project at all, because the first job becomes cutting it down, and only you know which parts matter.

Versioning, upgrading and release notes

Checking your build

Every Unity release names itself in two places. `Assets/Nucleus/VERSION.txt` reads `Nucleus <version>`, so you can check what a project has installed without building anything, and a running player writes `Thank you for using Nucleus Networking <version>.` to its log at launch, so you can check what a built player is running.

The DLLs themselves carry no version: `Nucleus.csproj` sets `GenerateAssemblyInfo` to `false`, and `AssemblyInfo.cs` declares only the title, product, company, copyright and a licence metadata string, with no `AssemblyVersion` or `AssemblyFileVersion` attribute. There is no protocol-version handshake between peers at connect time either, so the version a peer runs is something you check, not something the connection checks for you.

What this page will become

As releases accumulate, this page holds two things:

- **Release notes** — one entry per build, describing what changed.
- **API change register** — a table of deliberate breaking changes, each row keyed to the build it landed in and naming the direct replacement for whatever it removed.

Until then, the section below is the closest thing to that register: the breaking changes a current beta tester is likely to hit.

Changes to know about right now

Removed	Replacement
`IsAuthority`	`IsServerStarted` or `IsController`, depending on which you meant
`IsHostOwnClient`	removed, no direct replacement
`IsHostOwnServer`	removed, no direct replacement

Separately, a framework-wide naming audit renamed roughly 1,300 members in one pass. If a member you remember by name no longer resolves and isn't in the table above, it's almost certainly a casualty of that audit rather than a removal — search the current API for a similarly-named member before assuming it's gone.

Mixed builds are not supported

Both peers in a connection must run the same build. The wire format carries no version number, so a client and server on different builds will not fail cleanly — they'll either misread each other's packets or silently disagree about what a message means. There is no handshake to catch this for you. If you see corruption or unexplained desync, confirm both ends are on the same build before looking anywhere else.

Can't find where something moved?

If a member you're looking for isn't in the table above and you can't locate its replacement, see the support page.

Violation reference

What a violation is

A violation is a struct implementing `IViolation`, raised through `ViolationManager` when a peer sends something the protocol does not permit. The offending operation is always rejected regardless of what happens next; the violation only decides the `ViolationAction` taken against the `Connection` that sent it: `Ignore` or `Kick`. Logging is separate from the action, carried on `ViolationContext<T0>.Log`, a `LogEntry` the raise site may seed and the deciding handler may change or clear. Register a `ViolationHandler<T0>` to override the default action per type, or subscribe a `ViolationDetectedHandler<T0>` (or `IViolationObserver`) to be told about one without changing the outcome.

There are fifteen shipped violation types. Two exist only in Pro.

Peer-behavior violations

These fire from state application, spawning, and RPC dispatch. All default to `Ignore`.

Type	Payload	Check first
`UncontrolledStateChangeViolation`	`SystemId`	Whether the sender actually holds control of that `NetworkSystem` right now; control handoffs racing a state send look identical to this from the receiving side.
`PredictedStateChangeViolation`	`SystemId`	Whether the member is really predicted; a controller never serializes a predicted member's state, so this usually means the member's transmission setup changed without the controller rebuilding.
`EmptyCollectionDeltaViolation`	`SystemId`, `MemberId`	A compliant sender cannot produce a collection delta with no operations, so first suspect a diverged serializer pair between the two builds, not the sender's intent.
`InvalidStateAckViolation`	`AckedTick`	Whether the two peers' tick counters have actually diverged (a restart, a save/load, a manual tick reset) rather than assuming a forged ack.

`RetentionExceededViolation`	`AckedTick`	The peer's latency against `SystemManager.StateRetentionMilliseconds`; a single raise is ordinary heavy loss, not abuse.
`RecoveryUnconfirmedViolation`	`UnconfirmedServeCount`	Whether the peer's connection is dropping the recovery serve itself (packet loss eating the recovery, not just the state it was fixing) before assuming it is stalling on purpose.
`UnauthorizedSpawnViolation`	`SystemId`, `SystemBundleId`, `SystemLocalId`	Whether the server just despawned that system id and the client's own recovery pass re-served it as a full before learning of the despawn; that path raises nothing, so a raise here means the id was never one the server despawned.
`RpcSendPermissionViolation`	`SystemId`	Whether permission was revoked while calls already sent were still in flight; a well-behaved client reaches this on that race alone.
`RpcFloodViolation`	`ReceivedRpcCount`, `AllowedRpcCount`	`RpcManager.MaximumInboundRpcsPerConne against what the client actually needs to send in one drain, before assuming a flood attempt.

Malformed-data violation

`MalformedDataViolation` fires on the server when reading data a client sent throws, so the data is not applied. A well-behaved client never sends data the server cannot read, so it defaults to `Kick` in a release build and logs `Sent data the server could not read.` as a warning. A debug build (`#if DEBUG`) raises it with `Ignore` and no log line, and the read site logs the exception instead, since there the usual cause is a serializer under development. It is never raised on a client, for data with no known sender, for a server Connection (a host that dialled its own address reads from one as a client), or for a host's own client; in those cases the read site logs the exception and nobody is kicked, so a host never drops its own link or its own player over a bad read.

Type	Payload	Check first
`MalformedDataViolation`	`Kind` (`State`, `FullBody`, `Recovery`, `ClientState`, `Message`, `Rpc`), `SystemId` (zero when not for a system)	Whether both builds share the same serializers and member layout for that `Kind`; a version mismatch between client and server builds reads exactly like a forged packet. Run a debug build to see the exception.

Protocol-direction violations

These fire when a client answers about a bundle or scene it was never asked about. The server drives every load and release with a reliable request; nothing in the engine sends these messages upstream, so a well-behaved client cannot produce any of the four. They default to `Kick`.

Type	Payload	Check first
`UnexpectedBundleRequestViolation`	`BundleId`	Nothing on a well-behaved client sends this upstream; if it fires, look for custom code calling the bundle-request path on the wrong side.
`UnsolicitedBundleReportViolation`	`BundleId`, `IsLoadReported`	Whether the server actually requested that bundle load, or actually recorded the client holding it before an unload report; a mismatched build id here looks identical to a forged report.
`UnexpectedSceneRequestViolation`	`SceneHandle`, `IsLoadRequested`	Nothing on a well-behaved client sends this upstream; look for custom code calling the scene-request path on the wrong side.
`UnsolicitedSceneReportViolation`	`SceneHandle`, `IsLoadReported`	Whether the server actually requested that load or release; a scene handle that was never requested and never recorded held (including the unset handle) lands here too.

Pro-only violations

`PacketTransformRejectedViolation` and `UnauthorizedDespawnViolation` do not exist in a Free build; their events live on Pro partials of `ViolationManager`.

Type	Payload	Default	Check first
`PacketTransformRejectedViolation`	`ChannelId`, `PayloadByteCount`	No explicit default is passed at the raise site, so it takes `ViolationManager.DefaultAction` (`Ignore`), and because it is the plain framework default it always logs a warning.	Whether the peer is running the same `IPacketTransform` configuration; a stray packet from something that is not this game reaches the port too, so this is not evidence of an attack against a legitimate peer.

`UnauthorizedDespawnViolation`	`SystemId`	`Ignore`	Whether the client and server builds actually agree on that prefab's removal policy; a version skew between them reaches this honestly.

Reading a violation you just saw

Most types default to `Ignore` and stay silent unless something sets `Log` because a well-behaved peer can reach them through an ordinary race (a control handoff, a revocation, a recovery re-serve) or ordinary loss (`RetentionExceededViolation`, `RecoveryUnconfirmedViolation`, `PacketTransformRejectedViolation`). Treat a single raise of any of those as lossy-edge noise and do not kick for it.

The four protocol-direction violations are different: no engine path on a compliant client produces them at all, which is why they default to `Kick`. A raise there means either a deliberately crafted client or your own code driving the request/report path from the wrong side - check the latter first.

Adding Nucleus to a .NET project

Using Unity? See Installing the Unity integration.

Reference Nucleus

There is no NuGet package for Nucleus (no `PackageId`, no nuspec), so `dotnet add package` is not the route. Nucleus ships as prebuilt DLLs: `Nucleus.dll`, `CodeBoost.dll`, and `SynapseSocket.dll`, plus the source generator `Nucleus.CodeAnalysis.SourceGenerators.dll` (the Unity integration's `Assets/Nucleus` folder holds the same set). Copy all four, from one release, into your solution, for example a `lib/Nucleus` folder, and reference the three runtime DLLs from your `.csproj`:

```
<ItemGroup>
    <Reference Include="Nucleus"><HintPath>..\lib\Nucleus\Nucleus.dll</HintPath></Referenc
e>
    <Reference Include="CodeBoost"><HintPath>..\lib\Nucleus\CodeBoost.dll</HintPath></Refe
rence>
    <Reference Include="SynapseSocket"><HintPath>..\lib\Nucleus\SynapseSocket.dll</HintPat
h></Reference>
</ItemGroup>
```

Nucleus targets `netstandard2.1`, so a .NET 5 or later host (or Mono) can reference it as is. A .NET Core 3.x host can too, with two additions: a `PackageReference` to `System.Runtime.CompilerServices.Unsafe` version 5.0.0, which `CodeBoost.dll` depends on at a newer version than .NET Core 3.x ships, and `<LangVersion>9.0</LangVersion>`, because the generated code uses module initializers, a C# 9 feature, and .NET Core 3.x projects default to C# 8.

Reference the source generator

This step is not optional, and skipping it fails silently: the project still compiles, but nothing replicates.

Nucleus generates its `Write`/`Read` serializer methods and module-initializer registration at compile time. For the generator to run over *your* project's types, your `.csproj` needs its own `Analyzer` item for `Nucleus.CodeAnalysis.SourceGenerators.dll`:

```
<ItemGroup>
    <Analyzer Include="..\lib\Nucleus\Nucleus.CodeAnalysis.SourceGenerators.dll" />
</ItemGroup>
```

An `Analyzer` item runs the DLL as a Roslyn source generator over your project instead of linking it as a reference. Without it, a `NetworkComponent` partial in your project gets no generated `Write`/`Read` and no module-initializer registration: the type just never serializes. Add it to every project that declares networked types.

Check the generated output

Add `EmitCompilerGeneratedFiles` so a copy of the generator's output lands on disk where you can inspect it:

```
<PropertyGroup>
    <EmitCompilerGeneratedFiles>true</EmitCompilerGeneratedFiles>
</PropertyGroup>
```

With nothing else set, the files land under `obj/<Configuration>/<TargetFramework>/generated/Nucleus.CodeAnalysis.SourceGenerators/`, one `.g.cs` file per networked type with the type's name in the file name. The generator already adds this

code to the compilation itself, and the `obj` folder is outside your project's own `Compile` items, so there is nothing to exclude.

The compiler only adds and overwrites files there. It never deletes old ones, and `dotnet clean` leaves them too, so a file for a type you've since renamed or removed stays behind and can make a broken setup look fine. To check that the generator really runs, delete the `generated` folder and rebuild with `dotnet build --no-incremental`; a plain incremental build that finds nothing changed skips the compiler and leaves the folder empty. No file for your type after that means the generator didn't run over it.

Give your assembly a bundle id

The generator numbers each assembly's `NetworkSystem` and `NetworkComponent` types with local identifiers, paired with a bundle identifier so a type's wire identity is the tuple of the two. The main bundle defaults to `0` when `NetworkBundleAttribute` is absent. If your project is a second, separately compiled assembly, it needs its own unique id or it collides with the main bundle:

```
[assembly: NetworkBundle(1)]
```

Declare this once per assembly, in any file.

[NetworkType] and ExternalNetworkTypes

The generator reaches any type held in a `NetworkMember<T0>` field automatically — it does not need `[NetworkTypeAttribute]`. That attribute is only for forcing serializer generation on a type the generator otherwise can't reach on its own.

For a type declared in an assembly you can't annotate (you can't add `[NetworkType]` to someone else's struct), add a marker field for it to `Nucleus.Serializers.ExternalNetworkTypes`, the struct Nucleus itself uses for types like `System.Numerics.Vector3` and `Vector2`. Every field on that struct is a marker only — nothing reads it at runtime, and no instance is ever constructed. The generator walks its fields and emits a serializer for each field's type.

Choose your track: Unity or the core API

The one-sentence rule

Working in the Unity editor: read the Unity track. Calling Nucleus from plain C# — a console host, a dedicated server, another engine — read the API track.

Why the split is real

It is not a documentation convenience laid over one API. Dropping `UnityCoreManager` into a scene builds the entire session for you.

`UnityCoreManager.Awake` adds a `UnityNetworkLoopStepProvider` to drive its loop, constructs a `CoreManager` with it, then calls a private `EnsureManagers` that adds ten components to the same GameObject if they are not already there: `UnityNetworkLoopManager`, `UnityTransportManager`, `UnityServerManager`, `UnityClientManager`, `UnitySystemManager`, `UnityInterestManager`, `UnityPacketManager`, `UnityMessageManager`, `UnitySceneManager`, and `UnityPhysicsManager`. Once every component exists, `Awake` calls `ManagersInstantiated` on each `UnityManager` in its children — the same pattern `CoreManager`'s own constructor uses internally, calling `ManagersInstantiated` on every `ManagerBase` it owns once all of them are built.

A Unity reader never writes `new CoreManager(...)`, never chooses a loop step provider, and configures almost everything through inspector fields on these ten components. The API track's constructors and manager classes still exist underneath, but the component layer is the whole interface a Unity project needs.

When the Unity track still sends you to the API track

The component layer is a seat on top of the core managers, not a replacement for them. `UnityCoreManager` exposes the `CoreManager` it built as a public property, and several Unity manager components mirror one of its own fields onto an inspector field rather than adding new behavior.

`UnityNetworkLoopManager` is the clearest example: its inspector-configured tick rate is read back as `TickRate => (uint)_tickRate`, and `UnityCoreManager.Awake` reads that value before construction to pass as the `tickRate` argument to `new CoreManager(tickRate, ...)`. The inspector field and the core manager's own value are the same setting seen twice, not two settings. Any page documenting what a setting actually does, or what a core manager member is for, is written once on the API track and pointed to from the matching Unity field — never duplicated.

What the API track assumes the Unity track supplies

The core engine declares seams it does not implement, and expects an integration to fill them in. An API reader does not need to read the Unity track, but should know these seams exist and that Unity fills them:

- `ISceneLoader`: loads and releases scenes by handle and id; the core has no notion of a scene asset, path, or build index.
- `IInterestPositionReader`: reads a `NetworkSystem`'s world position for level of detail. (A distance condition reads positions through its own `TryReadSourcePosition`.) **This is a Pro seam**, not something every edition's Unity project has.
- `UnityPhysicsManager`: the inspector seat for the process-wide physics execution mode a `PhysicsSimulationCoordinator` reads.

A dedicated server or another engine implements these same seams its own way; nothing in the core requires Unity specifically.

Subjects that live in one track only

Some subjects have no counterpart on the other side, so don't search for a page that would be empty:

- **Unity-only**: component references (the fields on each `UnityManager`), prefab id stamping, and editor tooling.
- **API-only**: writing a custom `INetworkLoopStepProvider`, driving ticks yourself instead of letting a provider do it, and hosting a dedicated .NET server with no engine attached.

How the cross-links work

A page whose subject is split across both tracks opens with a one-line pointer to its counterpart — the Unity page for a setting points at the core manager member it mirrors, and the API page for that member points back at the inspector field. Following the link is never a detour into the same material restated; each side documents only what is actually different about it.

How replication works

The tick

Nucleus advances in fixed steps, not whenever a script happens to run. Every tick invokes eleven named steps in the same order: `EarlyVariableUpdate`, `EarlyStateUpdate`, `LateStateUpdate`, `Reconcile`, `EarlyFixedUpdate`, `LateFixedUpdate`, `VariableUpdate`, `EarlySerialize`, `LateSerialize`, `TickAdvance`, `LateVariableUpdate`. The tick advances last, at the end of `TickAdvance`.

That fixed order is what lets two independent peers agree on what happened and when. If game code could read or write networked state at an arbitrary point in a frame, a server and a client would each be comparing values captured at different, unpredictable moments. Instead, every peer applies incoming state at the same step, lets game code run in between, and writes outgoing state at the same later step. Everyone changes state on the same beat.

Two steps matter most for replication: `EarlySerialize`, where game code sets a networked value, and `LateStateUpdate`, earlier in the same tick, where values received from the network are applied. Because the apply step runs before the write step, a peer sees the world update before its own game code for that tick runs.

The journey of one value

Take a single networked member, declared on a component:

```
public partial class DemoScoreComponent : NetworkComponent
{
    public readonly NetworkMember<int> Score = new();
}
```

One trip across the network looks like this:

Written on the server. The peer that owns the object and decides its truth (the server, usually the server) sets `Score.Value` during `EarlySerialize`. `NetworkMember<T0>.Value` is the current value; setting it also rotates the value ring so the value it replaces becomes readable as `PreviousValue`.

Change detection notices it. A member does not serialize just because it exists. Setting `Value` flags the member as changed for this tick, and only changed members are considered for sending.

Serialized during the tick's serialize steps. During `LateSerialize`, the framework walks every system with changed members and serializes them.

Travels as part of one combined packet. Nucleus does not send one packet per member or per component. A tick's outgoing state for a connection is written into a single combined stream and handed to the transport during `LateSerialize`.

Applied on the receiver before that peer's own tick work runs. The receiving peer reads and deserializes incoming packets early (`EarlyVariableUpdate`), then applies the contained state during `LateStateUpdate`. Both steps come ahead of `EarlyFixedUpdate`, `VariableUpdate`, and that peer's own `EarlySerialize`. Game code on the receiver reading `Score.Value` during its own tick sees the value the server wrote, not something stale from before the packet arrived.

Snapshot, then delta

A peer that has never seen a system before cannot make sense of a partial update, so the first thing it receives is a full snapshot of every member's value. Every tick after that carries only what changed: a delta.

This is why `NetworkMember<T0>` exposes both `Value` and `PreviousValue`. A delta encodes against the previous value already held on both ends rather than restating the whole thing, which is cheaper the more ticks a value has been streaming. The moment a peer starts observing a system — joining, a scene load, an object first coming into range — it gets a full again, because there is no previous value on that peer's side yet to delta against.

Interpolation: correct, a little old, on purpose

A receiving peer does not render the newest value the instant it applies. `NetworkMember<T0>.InterpolatedValue` sweeps from `PreviousValue` to `Value` across the member's send interval rather than jumping the moment a new value lands. If you read `InterpolatedValue` on a peer that does not control the object, you are looking at a value that is deliberately a little behind the true, just-applied one — buffered and played back smoothly instead of popping between sparse updates. `Value` itself is never delayed; only the interpolated read is.

`SystemManager.StateInterpolation` sets how many ticks of buffer a peer holds before it starts applying received state at all, trading a little more latency for smoother arrivals when packets don't land on a perfectly even cadence.

When a packet doesn't arrive

Loss recovery here is not a reliable channel retransmitting a dropped packet. There is no such channel underneath state replication. Instead:

- Every outbound packet carries an acknowledgment in its header: the highest tick the receiver has fully applied. Acks only move forward.
- `SystemManager.GlobalRedundancy` lets an unreliable state tick resend itself on the following ticks, so a single lost packet is often covered by a later one that repeats the same change.
- When the gap between the last acknowledged tick and the current one grows too wide, the sender re-serves specifically the components that never landed, rather than resending everything or waiting on a generic retransmit.

The effect is that a lost packet is repaired by the next relevant state, not by a transport-level resend of the exact bytes that went missing.

The three gates on whether a peer sees a value at all

A member existing and changing is not enough for a given peer to receive it. Three separate questions decide that, for each system and each connection:

- **Is the system started?** An unstarted system has nothing to serialize or apply.
- **Does that peer observe it?** A peer only receives state for systems it is currently observing; state for a system outside a peer's interest is not sent to it.
- **Is that peer allowed to write it?** Only a peer with write access to a member's system has its writes accepted and forwarded to everyone else; an unpermitted write does not propagate.

All three are evaluated per connection, so the same system can be visible and writable to one peer while being invisible to another.

Where each step is documented

This page is the map, not the manual for any one stop on it:

- The full ordering and per-step framework work: the tick pipeline reference.
- Delta and full encoding, compression, and accuracy: the serialization reference.
- Interpolation windows, snap thresholds, and send intervals: the `NetworkMember` reference.
- Redundancy, interpolation depth, and retention tuning: the `SystemManager` reference.
- Acknowledgment, redundancy, and targeted recovery in full: the reliability and loss recovery reference.
- Interest and observation rules: the interest system reference.
- Write access and multi-writer systems: the write access reference.

Start Here

The first pages to read, in order, whether you are driving Nucleus through Unity or the core API directly.

- What Nucleus is
- Choose your track: Unity or the core API
- Terminology
- Installing the Unity integration
- Your first networked scene
- Replicating your first object
- Testing with two editors
- Nothing Is Replicating: A First-Run Checklist
- How replication works
- Adding Nucleus to a .NET project
- Your first .NET host
- Replicating your first system
- Running two peers in one process
- Where to go next

Installing the Unity integration

Driving the core API directly? See Adding Nucleus to a .NET project.

What you get

Nucleus for Unity ships as prebuilt DLLs, not source, for the engine itself. Import the release's `.unitypackage` (**Assets > Import Package > Custom Package**), or copy the whole `Assets/Nucleus` folder into your project. It contains:

- `Nucleus.dll`, `CodeBoost.dll`, `SynapseSocket.dll` — the engine and its dependencies.
- `Nucleus.CodeAnalysis.SourceGenerators.dll` — the Roslyn source generator that makes networked types compile.
- A `.pdb` next to `Nucleus.dll`, `CodeBoost.dll`, and `SynapseSocket.dll`, for debuggable stack traces. The source generator DLL doesn't ship one.
- `Nucleus.xml` and `SynapseSocket.xml`, the API documentation your IDE shows on hover.
- `LICENSE.txt`, and `VERSION.txt`, which names the release you installed.
- An `Integrations` folder.

`Integrations` holds four sub-folders — `Unity`, `Blitz Relay`, `Newfarm Client`, `Newfarm Migration` — each its own assembly definition. Mostly this is C# source and `.asmdef` files, not DLLs. Blitz Relay is the exception: alongside its own source and `.asmdef`, it also ships one precompiled dependency, `BlitzRelay.Protocol.dll`.

The one rule that matters: copy files, not .metas

Copy the DLLs and their `LICENSE.txt`/`.pdb` files. Do **not** copy the `.meta` files that ship alongside them into a project that already has its own Unity-generated `.meta`s for those paths, and never hand-edit or drop the `.meta` for `Nucleus.CodeAnalysis.SourceGenerators.dll`.

That file's `.meta` carries:

```
labels:
- RoslynAnalyzer
```

That label is what tells Unity to run the DLL as a Roslyn source generator instead of just referencing it as a normal plugin. Without it, the generator never runs: your project compiles — components, fields, everything looks fine — but nothing actually replicates, because the generated networking code was never produced. This fails silently, with no compile error, so it's worth checking directly: open `Nucleus.CodeAnalysis.SourceGenerators.dll.meta` in your project and confirm `RoslynAnalyzer` is present under `labels`.

Keep the three engine DLLs matched

`Nucleus.dll`, `CodeBoost.dll`, and `SynapseSocket.dll` must all come from the same build. If you update one without the other two — for example, dropping in a fresh `Nucleus.dll` next to an old `CodeBoost.dll` — you won't get a compile error. You'll get a runtime failure, since the DLLs assume they're talking to matching versions of each other. When updating, replace all three together.

Unity version floor

Every file in the Unity integration opens with the same conditional compilation gate, for example

`Managers/Core/UnityCoreManager.cs`:

```
#if UNITY_EDITOR || UNITY_2021_3_OR_NEWER
#define UNITY_ENGINE
#endif
```

The integration only compiles under the editor or Unity 2021.3 and newer. That makes 2021.3 the effective minimum supported version for a build target; anything older won't define `UNITY_ENGINE` and the integration code won't compile in.

What's in Integrations, and what's gated

The `Unity` folder is the integration itself — managers, components, everything that talks to `UnityEngine`. It has no `defineConstraints`, so it compiles as soon as it's in your project.

`Blitz Relay`, `Newfarm Client`, and `Newfarm Migration` are separate assemblies alongside it. The Blitz Relay assembly is special: its `.asmdef` declares `"defineConstraints": ["BLITZ_RELAY"]`, so that assembly only compiles when the `BLITZ_RELAY` scripting define is set in your project. The Unity integration assembly itself carries no such constraint — it references Blitz Relay unconditionally, but the relay code only builds in when you opt in with that define.

Nothing Is Replicating: A First-Run Checklist

Scene compiles, both peers connect, and the second peer sees nothing: no spawned object, no scene content, no state. Work through these five checks in order — a first-run setup gap accounts for almost every case.

1. Prefab not registered

A dynamically spawned object is instantiated on a client from `NetworkPrefabRegistry`, keyed by the prefab's `(bundleId, localId)` wire identity. If the prefab was never registered under that identity, the spawn arrives and has nothing to instantiate:

```
No networked prefab is registered for identity [{bundleId}, {localId}]; spawned NetworkSys
tem Id [{id}] cannot be instantiated. Content shards must load and register before spawns
arrive.
```

That log line is your confirmation. Causes: the prefab isn't in a `NetworkPrefabCollection` that has been built, its `NetworkSystemObject.PrefabId` is unset (rebuild the collection so it gets stamped), or its content shard/asset bundle hasn't loaded and called `RegisterAll` yet on the receiving peer.

2. Scene not networked

An object placed directly in a scene (`NetworkSystemObject.IsSceneObject` true, correlated by `SceneObjectId`) only binds once that scene is a live networked scene instance with a `sceneHandle` — not merely a scene Unity has loaded. The server opens the instance, and a client is placed in it through `SceneManager.TryRequestSceneLoad(connection, sceneHandle, ...)`. Load the scene with plain Unity APIs instead of going through Nucleus's scene system, or leave the scene out of the project's `NetworkSceneManifest`, and the scene's objects never get a handle to bind against.

3. No interest rule is admitting the object

By default every started `NetworkSystem` is registered to every authenticated client and nothing is filtered — but `UnityInterestManager` ships one interest rule on by default: `AuthoredSceneInterestCondition`, the scene gate. It withholds an object from a player until that player holds the same scene instance the object lives in. If the second peer hasn't been placed in that scene (see #2), the object is fully spawned on the server and still invisible to that client.

Check the **World Rules** list on `UnityInterestManager`, and the **Rules** list on the object's own `NetworkInterestObject`. Either can only restrict who receives the object, never expand it.

4. Wrong role branch

Spawn and write calls gated on the wrong role silently do nothing. `NetworkSystem` exposes `IsServerStarted`, `IsClientStarted`, `IsHostStarted`, and `IsController(ControllerType)` for exactly this; a spawn written inside a block guarded by the wrong one of these never runs on the peer that needed to run it. On a host, `IsServerStarted` and `IsClientStarted` are both true at once, which is the case most likely to hide a role mistake in testing.

5. Transport never started

`UnityTransportManager` connects nothing on its own unless **Automatic Start Mode** is set to `Server`, `Client`, or `Host` — its default is `None`. If you start the connection from code instead, confirm you actually called one of:

```
await unityTransportManager.StartServerAsync();
await unityTransportManager.StartClientAsync();
await unityTransportManager.StartHostAsync();
```

`TransportManager.IsServerStarted` / `IsClientStarted` only go true once a connection reaches `LocalConnectionState.Connected` (the client side also needs to be authenticated) — a transport added but never connected leaves every system with nothing to send to.

A host's own client is not proof of anything

If you're testing with a host and a second client, don't diagnose from what the host's own client half receives. A host's loopback packets — what the server side sent to itself — are discarded without ever being applied: in a Release build the discard happens before a single bit is read, so the header itself is never parsed; a Debug build reads the header first and discards after, for diagnostics, except for `PacketType.Raw`, which is exempt in Debug and compiled out of the enum entirely in Release. Either way, what the host's client half needs was already handed to it in-process at the point the server sent it, so this discard is expected and costs nothing. Seeing no traffic parsed on a host's own loopback is not evidence that serialization was skipped; check the second, genuinely remote peer instead.

Replicating your first object

Driving the core API directly? See Replicating your first system.

The marker component

`NetworkSystemObject` is the single point of correlation between a GameObject and a `NetworkSystem`. Every networked prefab and every networked scene object needs exactly one, at its root, and it carries the identifiers that tell the two peers they're talking about the same object:

- `PrefabId` — the prefab's identifier within its content shard, or zero when unset.
- `PrefabBundleId` — the identifier of the content shard that owns the prefab.
- `SceneObjectId` — the identifier of a scene object, or zero when this isn't one.

All three back onto private, editor-stamped fields (`_prefabId`, `_prefabBundleId`, `_sceneObjectId`); nothing sets them from script. `NetworkSystemObject` carries `[DisallowMultipleComponent]`, so a GameObject can only ever have one.

Start every networked object the same way: create the prefab, add `NetworkSystemObject` to its root, and stop there for this step — the identifier isn't stamped yet.

Giving the prefab an id

A prefab with no `PrefabId` can't spawn over the network. Run **Nucleus > Rebuild Network Prefab Collection** from the Unity menu (`NetworkPrefabCollectionBuilder.Rebuild`, on `[MenuItem("Nucleus/Rebuild Network Prefab Collection")]`) after adding a new networked prefab, and again any time you add or remove one. It scans the project for prefabs carrying a `NetworkSystemObject`, stamps each with a prefab id and its bundle's id, and writes the shard's prefab collection asset.

The id lives on the prefab asset itself, which is where the trap comes from: duplicate a networked prefab in the Project window and the copy carries the same stamped `_prefabId` as the original, because duplication copies the serialized data verbatim. Until you rebuild, both prefabs claim the same identifier and the wrong one can end up resolved on the receiving end, and a player build fails with the clash listed. Rebuilding gives the copy a fresh identifier and leaves the original's alone: the rebuild keeps an identifier on the prefab the collection already lists, and on a base over its own variant, so you don't need to clear anything by hand.

Adding NetworkTransform

Add `NetworkTransform` to the same GameObject and the object's position, rotation, and scale replicate with no code: the controller writes the transform each frame, and every other peer applies it through an interpolation buffer. It belongs to the nearest `NetworkSystemObject` at or above it, so on this object it uses the root's marker.

Two fields are worth knowing on day one:

- **Transform Space** (`_transformSpace`, a `TransformSpace`) — `Local` by default, replicating the offset the object holds inside its parent, which is what lets a carried object (one riding a moving platform, say) cost nothing while it rides. Switch to `World` for an object that may hang from something unreplicated; a `World` object's pose is unaffected by reparenting.
- **Send Interval** (`_sendInterval`, a `SendInterval`) — defaults to `Normal`. Only applies under the `Interval` transmission mode; the default transmission mode projects the path instead and isn't paced by this field.

The inspector also exposes **Interpolate Scale Enabled**, **Kinematic Management Enabled**, **Transmission Mode**, and **Extrapolation Enabled**. They exist and are worth knowing about later; the two above are the ones that matter for getting an object moving today.

Writing the first script

A script that needs the networked object inherits `NucleusBehaviour<TComponent0>` rather than plain `MonoBehaviour`. `TComponent0` names the one `NetworkComponent` type this script's system must carry:

```
using Nucleus.Components;

public partial class FirstObjectComponent : NetworkComponent
{
    public readonly NetworkMember<bool> IsActive = new();
}

using Nucleus.Connections;
using Nucleus.Integrations.Unity.Systems;
using Nucleus.Systems;
using UnityEngine;

public class FirstObjectController : NucleusBehaviour<FirstObjectComponent>
{
    protected override void OnSystemLinked()
    {
        Debug.Log($"[{gameObject.name}] linked. CoreManager: {CoreManager}, NetworkSystem:
 {NetworkSystem}");
    }

    protected override void OnServerStarted()
    {
        Debug.Log($"[{gameObject.name}] server role is up.");
    }

    protected override void OnClientStarted()
    {
        if (IsController(ControllerType.Client))
            Debug.Log($"[{gameObject.name}] this peer controls the object.");
    }

    protected override void OnControllerChanged(Connection previousControllerConnection, C
onnection currentControllerConnection)
    {
        Debug.Log($"[{gameObject.name}] controller changed from {previousControllerConnect
ion} to {currentControllerConnection}.");
    }

    private void Update()
    {
        if (!IsController(ControllerType.Client) || !IsStarted(Invoker.Client))
            return;

        // Safe to act on the object here: this peer controls it and the client role is up
.
    }
}
```

`NucleusBehaviour<TComponent0>.Awake` calls

`NetworkSystemObjectPool.RequireSystem<NetworkSystem, TComponent0>` for you, so there's no manual rent to write. From `NucleusBehaviourBase` you inherit:

- `NetworkSystem` — the linked system, null until it links.
- `CoreManager` — resolved in `Awake`.
- `IsController(ControllerType controllerType)` — false until a system is linked, otherwise the system's own answer.
- `IsStarted(Invoker invoker)` — whether this peer's server or client role is started, meaningful only once a system is linked.
- `EnsureIsController(ControllerType controllerType)` and `EnsureIsStarted(Invoker invoker)`: the same checks, but instead of failing silently they log a warning naming the calling member, once per calling member on each behaviour. `Update()` above uses the plain forms deliberately: it runs every frame, and a non-controlling or not-yet-started peer failing the check there is the normal case, not a bug, so even that one warning would be noise. Reach for the loud forms in one-off code, such as a button handler or an RPC, where a failure means something is actually wrong.

The lifecycle hooks

A first script overrides four things: `OnSystemLinked`, `OnServerStarted`, `OnClientStarted`, and `OnControllerChanged`. Each has a stopped counterpart (`OnSystemUnlinked`, `OnServerStopped`, `OnClientStopped`) for symmetry, but those aren't needed to get an object moving.

Linking doesn't wait for a role to start — it adopts whatever's already running. A scene object's `Awake` runs on every peer regardless of which one has a server or client up yet, so when the system links, the behaviour immediately checks whether the server and client roles are already started and calls `OnServerStarted`/`OnClientStarted` right then if they are, rather than waiting for a state-change edge that a role already running will never raise again. A role that starts later still raises the matching hook when it does.

For an object placed in the scene before anyone connects, this means `OnSystemLinked` fires first (from `Awake`, on every peer), and `OnServerStarted`/`OnClientStarted` fire immediately after it on whichever peer already has that role up — on a host, both fire back to back.

Replicating your first system

Using Unity? See Replicating your first object.

Declare the data

A networked component is a `partial class` deriving from `NetworkComponent`, holding one `NetworkMember<T0>` field per value you want replicated:

```
public partial class MatchClockComponent : NetworkComponent
{
    public readonly NetworkMember<int> Round = new();
    public readonly NetworkMember<int> SecondsRemaining = new();
    public readonly NetworkMember<int> Score = new();
}
```

You never write the serialization for these fields yourself. The class is `partial` because the source generator emits the other half: the reads and writes that put each `NetworkMember<T0>` on the wire, and the bookkeeping that tracks which members changed on a given tick. You declare the fields and their types; the generator supplies everything that moves them across the network.

A `NetworkSystem` is the thing that gets spawned and despawned; the components attached to it are what carry state. A system can carry more than one component, which is what the spawning APIs below are generic over.

Spawn on the server

The server rents a system and attaches the components it needs in one call:

```
if (!NetworkSystemPool.TryRent<NetworkSystem, MatchClockComponent>(coreManager, out Networ
kSystem? matchSystem, canStartSystem: true))
    return;
```

`TryRent` returns `false` when no `CoreManager` could be found or the start was refused. `canStartSystem` defaults to `true`; pass `false` to rent a system without starting it yet. `TryRent` is generated in overloads that take up to 31 component type arguments (`TComponent0` through `TComponent30`), so a system with several components is still one call:

```
NetworkSystemPool.TryRent<NetworkSystem, MatchClockComponent, ScoreboardComponent>(coreMan
ager, out NetworkSystem? matchSystem, canStartSystem: true);
```

Once started, write to the component's members the same way on either peer:

```
if (matchSystem!.TryGetComponent(out MatchClockComponent clock))
    clock.Round.Value = 1;
```

Finding it on the receiver

A system rented purely from code carries `NetworkSystem.UnsetPlatformId` — there is no prefab and no scene object behind it, so there is nothing for a spawn handler to bind or instantiate. That id being unset is deliberate: this kind of spawn never reaches `INetworkSystemSpawnHandler`.

Without an object to hang a reference off, and without an `Id` the receiver could have learned in advance, there is no handle to grab when the spawn arrives. `NetworkSystemPool.Watch` is the answer: a standing subscription that matches on system type and component composition, and hands the receiver its own instance the moment a matching system starts.

```
NetworkSystemWatch? watch = NetworkSystemPool.Watch<NetworkSystem, MatchClockComponent>(co
reManager, acquiredSystem =>
{
    if (acquiredSystem.TryGetComponent(out MatchClockComponent clock))
    {
        // acquiredSystem is this peer's own instance, already carrying the server's state
.
    }
});
```

Composition, not type alone, is what a watch matches on: a watch for `NetworkSystem, MatchClockComponent` is never offered a plain `NetworkSystem` that carries a different component. `Watch` also takes an optional released handler for when the matching system stops, and is generated in the same up-to-64-component overloads as `TryRent`.

Registering a watch is safe at any time. If a matching system already exists when you watch, it is replayed against the new watch immediately, so a subscriber that starts late still gets it.

Drop the subscription with `Unwatch` when you no longer need it:

```
NetworkSystemPool.Unwatch(watch);
```

Reading a value and reacting to a change

Read a member's current value through its `Value` property:

```
int secondsRemaining = clock.SecondsRemaining.Value;
```

To react when values change rather than poll them, override `OnMembersChanged` on the component. It is the single callback for every inbound apply, and it also fires on the peer that wrote the change:

```
public partial class MatchClockComponent : NetworkComponent
{
    public readonly NetworkMember<int> Round = new();
    public readonly NetworkMember<int> SecondsRemaining = new();
    public readonly NetworkMember<int> Score = new();

    public override void OnMembersChanged(ulong memberFlags, MemberChangeDirection memberC
hangeDirection)
    {
        if (memberChangeDirection == MemberChangeDirection.Read)
        {
            // Applied from the wire — this peer received the change.
        }
    }
}
```

`memberChangeDirection` tells you which side of the wire the raise is reporting: `Write` fires on the peer that just committed the change for serialization, `Read` fires when a peer applies a change that arrived from the wire. A handler that only cares about remote changes checks for `Read`.

Returning what you rented

A rented system is returned to the pool when you're done with it:

```
NetworkSystemPool.Return(matchSystem);
```

Or, to return it and clear your own reference in the same call:

```
NetworkSystemPool.ReturnAndNullifyReference(ref matchSystem);
```

A system handed to you by a watch is borrowed, not owned: the same instance stays valid from your acquired handler until your released handler, and a wire-constructed instance returns to its pool immediately after your released handler returns, so drop your reference there and keep none past it.

Using Unity

The Unity track replaces both the rent and the watch with a marker component on a GameObject and `NetworkSystemObjectPool.RequireSystem`. See Replicating your first object for that version.

Running two peers in one process

Using Unity? See Testing with two editors.

Two ways to run both sides in one process

A test or repro often needs a server and a client without launching a second process. Nucleus gives you two ways to do that, and they answer different questions.

`Yak` (`Nucleus.Transports.Yak`) is an offline transport with no socket underneath it at all. One `Yak` holds both the server half and the client half inside a single transport on a single `CoreManager`, so what it gives you is a host: a server and its own local client, not a second, separate peer. It cannot connect two `CoreManager` instances. `Synapse` (`Nucleus.Transports.Synapse`) is the real UDP transport; running two `CoreManager` instances over it on `127.0.0.1`, the client dialing the port the server listens on, gives you two separate peers over a genuine socket pair, still inside one process.

Use `Yak` when a host on its own is enough: single-player or offline play, or game logic that doesn't need a remote peer. Use two loopback `Synapse` managers when you need a second, separate peer, and whenever the bug - or the thing you need to prove works - lives in framing, segmentation, batching, or anything else that only happens when bytes actually cross a socket.

Yak: no sockets at all

`Yak` overrides `IsLocalTransport()` to return `true`, which is how the rest of the engine knows it isn't talking to a real network:

```
public class Yak : SocketPairTransport<ServerSocket, ClientSocket>
{
    public override bool IsLocalTransport() => true;
    ...
}
```

Add it to a `CoreManager` the same way you'd add any transport:

```
CoreManager coreManager = new();
Yak yak = (Yak)await coreManager.TransportManager.AddTransportAsync<Yak>();

await yak.ConnectAsync(Invoker.Server);
await yak.ConnectAsync(Invoker.Client);
```

Connecting both halves makes that one `CoreManager` a host. Both socket halves belong to the same transport and hand data to each other directly. The engine still serializes every packet into bytes exactly as it would for a real transport; those bytes just never become a datagram, and no OS is involved.

Two CoreManagers over Synapse

When you need the real shape - framing, segmentation, batching - build two `CoreManager` instances, each with its own `Synapse` transport, pointed at the same loopback port, and each with its own loop provider passed at construction:

```
MainThreadStepProvider serverLoop = new();
MainThreadStepProvider clientLoop = new();

CoreManager server = new(tickRate: 60, networkLoopStepProvider: serverLoop);
Synapse serverSynapse = (Synapse)await server.TransportManager.AddTransportAsync<Synapse>(
);
serverSynapse.Configuration.Port = port;

CoreManager client = new(tickRate: 60, networkLoopStepProvider: clientLoop);
Synapse clientSynapse = (Synapse)await client.TransportManager.AddTransportAsync<Synapse>(
);
clientSynapse.Configuration.Port = port;
clientSynapse.RemoteHost = IPAddress.Loopback;

await serverSynapse.ConnectAsync(Invoker.Server);
await clientSynapse.ConnectAsync(Invoker.Client);

// Every frame, step both peers from this one thread:
serverLoop.Step(frameDeltaMilliseconds);
clientLoop.Step(frameDeltaMilliseconds);
```

`MainThreadStepProvider` is the small `NetworkLoopStepDriver`-based provider from Running a dedicated .NET server, which steps its loop only when you call `Step`. Because each manager gets its provider at construction, neither loop moves while the transports are added and connected, and both are then stepped from the same thread. `Synapse.RemoteHost` defaults to `IPAddress.Loopback`, so a single-process pair works without any extra addressing; you only set it explicitly on the client half to be clear about what's connecting to what.

Why the shortcut hides bugs

A `Yak` pair never touches a socket, so anything the socket layer is responsible for never runs. A call body is length-prefixed into a per-channel accumulator that a real transport batches, segments across datagrams if it's too large, and rejoins on the other end. Hand that same buffer straight from one in-process socket to the other and none of that machinery executes - the body just arrives.

That means a framing bug - an off-by-one in a length prefix, a segment that doesn't rejoin in the right order, an accumulator that doesn't flush at the right boundary - can pass every `Yak`-based test and still break the moment it crosses a real socket. If what you're testing is replication logic, spawning, or RPC routing, `Yak` is faster and sufficient. If what you're testing is the wire format itself, it has to run over `Synapse`.

Keeping two managers apart

`NetworkLoopManager`, `TransportManager`, and everything else hang off the `CoreManager` instance you created, not off the type - so driving two peers means holding two `CoreManager` references and passing the one you mean into whatever drives it. The one static hook, `CoreManager.Instance`, only ever names the oldest still-live manager (first write wins at construction, and a later one takes the slot over only through `EnsureSetInstance`); with two peers alive, reaching for `CoreManager.Instance` gets you whichever one happened to be built first, not "the" one you meant.

The loop enforces this at the step level, too. `NetworkLoopManager.InvokeNetworkLoopStep` claims the step for the calling thread; a second thread that calls in while a step is already in progress is turned away rather than run, and the manager counts it and reports it periodically rather than staying silent. One loop drives one manager - if you're hand-driving two peers, you drive them from the same thread, one `InvokeNetworkLoopStep` call at a time.

Hand-driving ticks

Nothing ticks on its own unless you give the `NetworkLoopManager` a provider that drives itself. For a deterministic repro, give it one that does nothing and drive every step yourself:

```
internal sealed class ManualStepProvider : INetworkLoopStepProvider
{
    public bool IsStarted { get; private set; }

    // Set by the code driving the loop: true before the steps of a pass that ticks.
    public bool IsTickFrame { get; set; }

    public void Initialize(NetworkLoopManager networkLoopManager) { }
    public void Start() => IsStarted = true;
    public void Stop() => IsStarted = false;
    public void Return() { }
}
```

Pass the provider to the `CoreManager` constructor, not by swapping it in later:

```
ManualStepProvider stepProvider = new();
CoreManager coreManager = new(networkLoopStepProvider: stepProvider);
```

Nothing steps the loop until the constructor's last line, so a provider named there means the default, thread-pool-driven provider never starts at all. Swapping yours in afterward with `UseNetworkLoopStepProvider` is too late: by then the default is already stepping the loop from the thread pool, racing whatever your code does until the swap.

With the provider silenced, drive each step yourself through `NetworkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps, StepDelta)`:

```
StepDelta delta = new(0, 0, 0);

coreManager.NetworkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.EarlyVariableUpdate,
 delta);
```

Drive the eleven steps in their fixed order

`NetworkLoopSteps` declares eleven steps, and they mean something in this order: state is read and applied at the start of a tick's steps, then serialized and sent near the end of them, with a reconcile and the fixed-update steps in between, and the tick advances last, at the end of `TickAdvance`. Driving them out of order applies or sends state at the wrong point in the tick and produces a bug that doesn't exist in the real loop.

Call the steps in this canonical order, once per step, per `CoreManager`, per tick. Set the provider's `IsTickFrame` to true before a pass that ticks; left false, the pass sends its messages but writes no state:

```
StepDelta delta = new(0, 0, 0);

stepProvider.IsTickFrame = true;

networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.EarlyVariableUpdate, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.EarlyStateUpdate, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.LateStateUpdate, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.Reconcile, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.EarlyFixedUpdate, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.LateFixedUpdate, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.VariableUpdate, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.EarlySerialize, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.LateSerialize, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.TickAdvance, delta);
networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.LateVariableUpdate, delta);
```

For two peers, drive one full pass of all eleven steps on one `CoreManager`, then the same pass on the other, per simulated tick.

Terminology

Roles and peers

A **server** is the peer that decides what is true. A **client** is a peer that connects to a server to play. A **host** is a single peer that is both: it runs a server and a client together, so the person running it plays too.

Three properties on `TransportManager` answer which of these the local peer is:

- `IsServerStarted` — a server connection is established.
- `IsClientStarted` — an authenticated client connection is established.
- `IsHostStarted` — both of the above are true at once.

`Invoker` is a separate, smaller idea: a byte enum, `Invoker.Client` = `0` and `Invoker.Server` = `1`, used wherever code has to say which role an operation acts for rather than what the local peer is. A host holds a connection for each role, and APIs like `Transport.TryGetConnection(Invoker, out Connection)` take an `Invoker` to say which one they mean (`TransportManager` answers the same question through `TryGetServerConnection` and `TryGetLocalClientConnection`):

```
public enum Invoker : byte
{
    Client = 0,
    Server = 1,
}
```

The replicated things

- **NetworkSystem** — the replicated entity. It holds NetworkComponents, is created and destroyed as a unit, and is what spawns, despawns, and is observed.
- **NetworkComponent** — a container of replicated data attached to a NetworkSystem. Your own code writes NetworkComponent subclasses to declare what a system replicates.
- **NetworkMember<T0>** — one replicated value inside a NetworkComponent, declared as a field:

```
public readonly NetworkMember<int> X = new();
```

- **NetworkSystemGroup** — groups several NetworkSystems so they spawn and despawn together, as one object, rather than each resolving interest on its own.

Control, not ownership

Nucleus says **controller** where other engines say owner. `ControllerType` is a flags enum naming who controls a system:

```
[System.Flags]
public enum ControllerType : byte
{
    None = 0,
    Server = 1 << 0,
    Client = 1 << 1,
    AnyController = Server | Client,
}
```

`NetworkSystem.IsController(ControllerType controllerType)` is the canonical "do I control this" check, true when the local peer is the server and no client controls the system (

`ControllerType.Server`), or when the local peer is the controlling client (`ControllerType.Client`).

Control is server-assigned. Only the server can call `NetworkSystem.SetController(Connection controllingClientConnection)` — passing `null` returns control to the server. The system's current controlling client, or `null` when the server controls it, is exposed as `NetworkSystem.ControllingClient`.

Connection and its identity

Connection is Nucleus's object for the link to one peer. `Connection.Id` is a `uint`, recycled once the connection it named disconnects — the next peer to connect can be handed the same id.

ConnectionHandle is what a game holds onto instead, when it needs to remember a peer for longer than that peer's link lasts. It is a small readonly struct:

```
public readonly struct ConnectionHandle
{
    public readonly uint Id;
    public readonly uint Generation;
}
```

`Generation` is what makes a stale handle fail rather than alias: identifier seven issued twice is two different handles, and only the current one resolves through `TransportManager.TryGetConnection(ConnectionHandle, out Connection)`. A bare `Connection.Id` is no better, because identifiers are recycled too — a recycled id can never resolve to whoever arrived next once it's wrapped in a handle.

Words for time

- **Tick** — one step of the network clock, `NetworkLoopManager.Tick`. It starts at `FirstTick` = `1` and counts up; it never starts at zero.
- **Tick rate** — `NetworkLoopManager.TickRate`, ticks per second, clamped between `MinimumTickRate` (5) and `MaximumTickRate` (128), defaulting to `DefaultTickRate` (30).
- **UnsetTick** — `NetworkLoopManager.UnsetTick` = `0`, the sentinel meaning "no tick," chosen so no live tick can ever equal it.
- **Subtick** — `NetworkLoopManager.SubtickPercentage`, the fraction of a tick (0 to 1) elapsed since the last one, used to smooth rendering between ticks rather than snapping to tick boundaries.

Interest, observer and spawn

A peer **observes** a system when it has been registered to receive that system's updates. Observation is resolved by `InterestManager`, which tracks each connected client's observer registration across every started system, so replication cost scales with what a client can perceive rather than with the size of the world.

Spawn means a system begins replicating to a peer — its data starts being sent and the peer's local representation is created from it. It does not mean an engine object is instantiated; a system with no engine object at all still spawns and despawns in exactly this sense.

Messages versus RPCs

A **message** is addressed to a peer: `Connection.SendMessage<T0>(Channel channel, T0 data) where T0 : IMessage, new()`.

An **RPC** is addressed to a NetworkSystem: `NetworkSystem.TrySendRpc<T0>(RpcTarget`

`rpcTarget, Channel channel, T0 rpc, RpcSelfDelivery rpcSelfDelivery = RpcSelfDelivery.None) where T0 : IRpc, new()`. The `RpcTarget` (`Server`, `Observers`, or a specific connection) says who receives it; the system it's sent on says which replicated entity it concerns.

Transport and channel

`Channel` says how data is sent:

```
public enum Channel : byte
{
    Reliable = 0,
    Unreliable = 1,
}
```

`Channel.Reliable` is sent ordered and guaranteed to arrive; `Channel.Unreliable` is not.

Scene handle versus scene id

Every scene load is additive, so the same scene asset can be opened more than once at a time, and two open copies need to be told apart. A `SceneLoadRequest` carries both:

- `SceneId` (`ushort`) — which scene asset.
- `SceneHandle` (`uint`) — which live, currently-open copy of it.

The id names the asset; only the handle names one particular loaded instance of it.

Testing with two editors

Driving the core API directly? See Running two peers in one process.

One editor shows nothing

Every interesting piece of networked behavior is a difference between what two peers see: a spawn the server created and the client received, a value the client can't write because it doesn't hold permission, an object that falls out of one peer's interest but not the other's. A single play session only ever shows one side of that difference. To see anything worth testing, run a server and a client at the same time, in two separate Unity processes, and watch both screens.

ParrelSync clones

The Unity integration project vendors ParrelSync at `Assets/ParrelSync`. A clone is a second Unity process pointed at a project folder whose name is the primary's with a `_clone` suffix — the checkout on disk has the primary (`Nucleus - Unity Integration`) alongside `Nucleus - Unity Integration_clone_0`, `_clone_1` and `_clone_2`.

A clone mirrors the primary's `Assets` folder rather than copying it. That's why the engine DLLs only need to be deployed into the primary project once: every open clone picks up the same files, and there's no separate deploy step per clone.

Multiplayer Play Mode

Unity's built-in Multiplayer Play Mode (MPPM) is the alternative to a clone: instead of a second Unity process, it runs additional "virtual players" inside the same editor. Either approach gives you a second peer to press play on; which one you reach for is a matter of taste and machine resources, not a Nucleus concern.

Which peer becomes which

Role resolution for every demo lives in one place: `DemoNetworkDriver.ResolveRole`, in `Nucleus.Integrations.Unity/Demos/Common/Scripts/DemoNetworkDriver.cs`.

```
internal static RoleMode ResolveRole(RoleMode startingRole, bool isCloneProject, bool isMa
inEditor)
{
    if (startingRole is not RoleMode.Auto)
        return startingRole;

    if (isCloneProject)
        return RoleMode.Client;

    return isMainEditor ? RoleMode.Server : RoleMode.Client;
}
```

It's called with `Application.dataPath.Contains("_clone")` for `isCloneProject` and `Unity.Multiplayer.PlayMode.CurrentPlayer.IsMainEditor` for `isMainEditor`. A ParrelSync clone or an MPPM virtual player resolves to `RoleMode.Client`; a plain main editor falls back to `RoleMode.Server`. The driver's own inspector tooltip on the role field says exactly this:

> Auto: main editor is server, a `_clone` or virtual player is client. Force Server/Client for two independent editors.

This is demo code, not engine behavior — it exists so the sample scenes work without any setup. A real project decides its own role, typically through `UnityTransportManager`'s `Automatic Start Mode` field or by calling `StartServerAsync` / `StartClientAsync` / `StartHostAsync` explicitly.

Setting both editors to start as host also works, on purpose. `StartHostAsync` starts the client half whether or not the server half bound its port, so the first editor to press play takes the port and hosts, and the second fails to bind and its client half joins the first as an ordinary client. The second editor needs no separate client setup.

The trap: two main editors

`isMainEditor` is true for any plain editor instance, not just the "real" one — a second, independently-installed editor pointed at a plain copy of the project also reports as the main editor. Only the `_clone` path check tells two ParrelSync-related instances apart. Pair anything that isn't a clone-plus-primary (two separate checkouts, two machines) and `Auto` resolves both to `Server`; set the driver's role field to `Server` on one side and `Client` on the other instead.

Starting a session

Press play in the server peer first, and give it a moment to start listening before pressing play in the client. Both peers connect over loopback, on the port configured on the scene's transport — there's nothing to configure outside the scene itself.

The redeploy hazard

Dropping newly built DLLs into an editor that's currently in play mode forces Unity to reimport them, which triggers a domain reload mid-session. That tears down and rebuilds every managed object in the running scene, including the ones the network session was mid-flight with. Stop play mode in every open editor and clone before redeploying.

What Nucleus is

Nucleus is a networking engine for server-authoritative multiplayer games. Its core is engine-agnostic C#, and Unity is one integration layer on top of it, not a dependency it needs.

The shape of the product

The core lives in `Nucleus/` and targets `netstandard2.1`. Its assembly definition sets `noEngineReferences: true` — a guard, not a formality, that exists so the folder could be junctioned into a Unity project without ever quietly putting `UnityEngine` on the engine's reference path. The core compiles and runs on any .NET host that satisfies netstandard2.1, engine or none.

Everything engine-specific sits above that core as an optional layer: the Unity integration (`Nucleus.Integrations.Unity/`), Blitz Relay (`Nucleus.Integrations.BlitzRelay/`), and Newfarm (`Nucleus.Integrations.Newfarm/`). None of them are things the core links against or needs to function. They consume the core; it does not know they exist.

What that buys you

The practical result: the same replication code runs in a Unity client and in a headless .NET dedicated server with no Unity install at all. A `NetworkComponent` and its members don't change shape depending on which process is running them.

Concretely, the core replicates in `System.Numerics` types — `Vector3`, `Quaternion` — not `UnityEngine.Vector3` or any other engine's vector types. A dedicated server built from the core alone never touches an engine assembly, and a `Vector3` value that started on a client's transform and crosses the wire arrives as the same `System.Numerics.Vector3` on the other side.

The three ideas everything else assumes

Replicated objects are NetworkSystems. A `NetworkSystem` is the networked entity — it holds and serializes the `NetworkComponents` attached to it. A `NetworkComponent` in turn holds the actual replicated values as `NetworkMember<T0>` fields:

```
public partial class ChangeDetectionStateComponent : NetworkComponent
{
    public readonly NetworkMember<float> Drift = new(accuracy: 0.1f);
    public readonly NetworkMember<int> CoarseCounter = new(accuracy: 5f);
}
```

A `NetworkMember<T0>` is the unit of replication. Nothing above it — the `NetworkComponent`, the `NetworkSystem` — carries wire values directly.

The server decides, clients propose. One side of a connection controls an object's state; every other peer receives what the server decided rather than simulating its own version. A client that wants to affect that state sends a request; it doesn't write the value itself.

Interest decides what a peer can see. Visibility isn't global. What a given connection is allowed to see of a given object is resolved per object, per connection, through the interest system (`InterestManager`, `IInterestCondition` and their implementations). A NetworkSystem existing doesn't mean every peer receives it.

State arrives over an unreliable channel by design

There is no reliable channel underneath replication quietly retransmitting every state change until it lands.

Acknowledgment rides the packet header of the traffic already flowing, and recovery is targeted at what a peer is actually missing rather than a blanket retransmit. If you're used to a reliable-by-default RPC layer, read the reliability and redundancy docs before assuming state "just arrives" — here, arriving is something the system has to actively account for, not a channel guarantee you get for free.

Where the pieces live

- `Nucleus/` — the engine-agnostic core: systems, components, members, serializers, interest, transports.
- `Nucleus.Integrations.Unity/` — the Unity layer: MonoBehaviours, inspector-facing components, and the demos.
- `Nucleus.Integrations.BlitzRelay/` and `Nucleus.Integrations.Newfarm/` — the other integration layers.
- `Nucleus.Tests/` — the body of runnable, core-only examples. Reading a component or a test here shows working core code with no engine attached.

Where to go next

A quickstart gets one object moving between two peers. It does not tell you which of the thirty other pages matter for the game you're actually building. This page does.

If you're building on Unity

- **A player-controlled character.** Start with control and `NucleusBehaviour`: how a system decides who may drive it, and `IsController` as the check your scripts gate client state and movement on. `NetworkPlayerSpawner` hands each authenticated connection its own object automatically; you still write what a script does with it once `IsController` says this peer drives it.
- **A world with more objects than one peer can hold.** Read interest next: how `InterestManager` decides which spawns, despawns, and state updates a connection actually receives.
- **A game with several rooms or matches.** Read up on scenes and per-room worlds — each room opening its own copy of a scene, and how a system finds the copy it belongs to rather than the first one loaded.
- **A dedicated server.** Read the server/headless setup page, then control — a dedicated server is the clearest case of a peer that drives everything and controls nothing itself.

If you're building against the core library

- **A player-controlled character.** Read control at the `NetworkSystem` level — `IsController(ControllerType)`, without the Unity component layer (`NucleusBehaviour`, `EnsureIsController`) on top.
- **A world with more objects than one peer can hold.** Same interest page as above; `InterestManager` is core, not Unity-specific.
- **A game with several rooms or matches.** Read the scene management page for how a server opens and closes per-room scene instances and how systems get scoped to one.
- **A dedicated server.** Read the transport and connection setup pages for running a `CoreManager` headless, then control.

What everyone needs by week two

Regardless of route, three pages stop being optional once you're past a single moving cube:

- **Control** — who may write what, and why a write from the wrong peer is silently wrong rather than an error.
- **Interest** — why a connection sees some objects and not others, and what changes that.
- **Diagnostics** — when an object exists on one peer and never appears on another, this is the page that walks the checklist: is it in the sender's interest set, did the spawn actually replicate, is the receiving system still waiting on a handshake step. Read it before assuming the bug is yours.

The two guided tutorials

If you'd rather build one whole small game than read features in isolation, work through both tutorials in order. Each is a sequence of small, working steps rather than a single walkthrough.

Courier Run starts from a bare connecting world and adds, step by step: a per-player controlled object, prefab claiming, keeping a claim honest against a race, carrying and scoring an item, replacing `Instantiate` with pooled spawning, measuring what each kind of object costs to replicate, and moving the

whole arena into a scene of its own that every player joins. By the end it also covers a scoreboard that reaches every player regardless of which scene they're standing in.

Lobby and Worlds starts from an empty lobby and builds up room membership: taking a name, creating and listing a room, joining with its four refusal cases, leaving and host handoff, kicking, and ready-checks. It then opens a scene per room — the trap of placing an object before that scene is the one you're in, and how a system finds the copy it was just opened into — and finishes with per-room physics and a per-room win message.

Neither tutorial is a substitute for the reference pages above; each explains a feature only as far as that game needs it.

Runnable examples, no engine required

`Nucleus.Tests` is the actual body of core-only examples. `Integrations/Bridge/BridgeSyncTests.cs` wires two real `CoreManager` instances together over an in-memory transport and drives them by hand — message round-trips in both directions, server-to-client system spawn, and delta convergence — with no Unity, no sockets, and no flaky timing. `Harness/LoopHarness.cs` is the setup code those tests share: building a hand-driven `CoreManager`, and ticking both ends forward until a condition holds. If you learn an API fastest from working code rather than prose, start there.

When a page doesn't answer it

If you've read the relevant reference page and the tutorial chapter that covers the same feature and you're still stuck, that's worth asking about directly rather than guessing from partial documentation — file the question against the Nucleus repository rather than assuming the gap is intentional.

Your first .NET host

Using Unity? See Your first networked scene.

A session starts with a CoreManager and a loop provider

A Nucleus session starts with a `CoreManager`, given the loop provider that will step it:

```
using Nucleus.Managers.Core;

MainThreadStepProvider loop = new();
CoreManager coreManager = new(networkLoopStepProvider: loop);
```

That constructor is the whole bootstrap. It builds every manager the session needs and, on its last line, installs and starts the loop provider you pass it. There is no `Initialize` or `Start` call to make afterward. `MainThreadStepProvider` is a small class of your own, shown under "Driving the network loop" below: it steps the loop only when your code tells it to, so everything you set up before then runs against a loop that is standing still.

The constructor takes two optional arguments:

```
public CoreManager(uint tickRate = NetworkLoopManager.DefaultTickRate, INetworkLoopStepPro
vider? networkLoopStepProvider = null)
```

`tickRate` fixes the session's ticks per second for its lifetime. `networkLoopStepProvider` is covered below.

The managers it builds

`CoreManager` exposes each subsystem as a public field, all constructed together:

Field	Owns
`NetworkLoopManager`	The tick loop itself.
`TransportManager`	Transports and connections.
`ServerManager`	The server role: approving, kicking, messaging clients.
`ClientManager`	The client role.
`SystemManager`	Replicated systems.
`InterestManager`	What each connection observes.
`SceneManager`	Scene load/unload and completion.
`PacketManager`	Packet framing.
`MessageManager`	`IMessage` registration and dispatch.
`RpcManager`	Remote calls.
`ViolationManager`	Misbehaving-peer tracking.

A Pro build adds two more fields the same way: `BundleManager` (content delivered outside the build — Addressables, a patcher, a CDN) and `WorldPersistenceManager` (saving and rebuilding a world on disk). A Free build has neither field at all.

Driving the network loop

Nothing ticks until something steps the loop. The provider from the bootstrap is the smallest one that works: each time your code calls `Step`, it hands the time since the previous call to `NetworkLoopStepDriver`, which decides when a tick runs and invokes the loop's steps in order. Put it in its own file:

```
using Nucleus.Managers.NetworkLoop;

/// <summary>Steps the network loop only when called, on the calling thread.</summary>
sealed class MainThreadStepProvider : INetworkLoopStepProvider
{
    private readonly NetworkLoopStepDriver _driver = new();

    public bool IsStarted { get; private set; }

    public bool IsTickFrame => _driver.IsTickFrame;

    public void Initialize(NetworkLoopManager networkLoopManager) => _driver.Initialize(ne
tworkLoopManager, 1000f / networkLoopManager.TickRate);

    public void Start() => IsStarted = true;

    public void Stop() => IsStarted = false;

    public void Return() { }

    public void Step(float frameDeltaMilliseconds)
    {
        if (!IsStarted)
            return;

        _driver.AdvanceEarly(frameDeltaMilliseconds, isVariableUpdateAllowed: true);
        _driver.AdvanceLate(frameDeltaMilliseconds, isVariableUpdateAllowed: true);
    }
}
```

If your host already runs its own loop, such as a game engine's update or a custom simulation tick, call `Step` from there instead of from `Main`.

Pass the provider at construction, not after. Nothing steps the loop until the constructor's last line, so naming a provider there means the default never runs at all. Pass no provider and the constructor installs a `SystemNetworkLoopStepProvider`, which runs the loop off a `System.Timers.Timer` on a thread-pool thread from the moment the constructor returns. Everything you do after that, adding transports and subscribing to events included, races a loop that is already stepping, and swapping a provider in later leaves the default driving it until the swap. See Threading and lifetime.

Bringing up a transport

A single-process host

The simplest way to get a server and a client talking is one process playing both roles over an in-memory transport, `Yak`:

```
using Nucleus.Connections;
using Nucleus.Transports.Yak;

TransportManager transportManager = coreManager.TransportManager;

Yak yak = new();
await transportManager.TryAddTransportAsync(yak);

await yak.ConnectAsync(Invoker.Server);
await yak.ConnectAsync(Invoker.Client);
```

Make these `ConnectAsync` calls after you have subscribed to events and registered handlers, as the next two sections show. On `Yak` the connection comes up inside `ConnectAsync` itself, so a handler added afterwards never sees it.

`ConnectAsync` lives on the `Transport`, not on `TransportManager`. `Invoker` just names which role the call acts for — it says nothing about whether that role is actually started yet.

Two real processes

For an actual server and client across a socket, add `Synapse` and configure where it connects:

```
using System.Net;
using Nucleus.Connections;
using Nucleus.Transports.Synapse;

Synapse synapse = (Synapse)await transportManager.AddTransportAsync<Synapse>();
synapse.Configuration.Port = 7777;
synapse.RemoteHost = IPAddress.Loopback;

// On the server process:
await synapse.ConnectAsync(Invoker.Server);

// On the client process:
await synapse.ConnectAsync(Invoker.Client);
```

`Configuration.Port` is the port the server listens on and the client connects to. `RemoteHost` is the address the client dials — on the core `Synapse` transport it's an `IPAddress`, not a string.

Watching it connect

Subscribe to `TransportManager.ConnectionRemoteStateChanged` to see connections come up, and read the role flags off `TransportManager` once they do:

```
transportManager.ConnectionRemoteStateChanged += change =>
{
    Console.WriteLine(change.AsString());

    Console.WriteLine($"Server started: {transportManager.IsServerStarted}");
    Console.WriteLine($"Client started: {transportManager.IsClientStarted}");
    Console.WriteLine($"Host started: {transportManager.IsHostStarted}");
};
```

`IsHostStarted` is `IsServerStarted && IsClientStarted` — true once a single process is acting as both.

Sending something across

Define a message type by implementing `IMessage`:

```
using Nucleus.Managers.Messages;
using Nucleus.Serializers;

[NetworkType]
public struct HelloMessage : IMessage
{
    public string Text;
}
```

`IsAuthenticationRequired` decides whether the sender must be authenticated for the message to be delivered; it defaults to `true`, and by default Nucleus authenticates a connecting client the moment its transport connects, so this needs no attention here. Override it to `false` only for messages that must reach an unauthenticated peer, such as a handshake of your own.

Register a handler on `MessageManager` and send with `Connection.SendMessage`:

```
using Nucleus.Transports;

coreManager.MessageManager.RegisterMessageHandler<HelloMessage>((in MessageContext context
, HelloMessage message) =>
{
    Console.WriteLine($"Received: {message.Text}");
});

transportManager.ConnectionRemoteStateChanged += change =>
{
    if (change.Connection.RemoteState is RemoteConnectionState.Connected && change.Connect
ion.IsClient)
        change.Connection.SendMessage(Channel.Reliable, new HelloMessage { Text = "hello f
rom the server" });
};
```

That's the whole round trip: the server sends the moment the client's connection comes up remotely, and the handler registered above prints it on receipt. See the Messaging pages for anything beyond a single struct — batching, per-connection sends, and the authentication handshake itself.

Running the loop, and shutting down

With everything wired up, `Main` becomes the loop. Step the provider until you want to stop, then tear the session down:

```
using System.Diagnostics;

CancellationTokenSource stopSource = new();
Console.CancelKeyPress += (_, eventArgs) =>
{
    eventArgs.Cancel = true;
    stopSource.Cancel();
};

Stopwatch frameClock = Stopwatch.StartNew();
while (!stopSource.IsCancellationRequested)
{
    float frameDeltaMilliseconds = (float)frameClock.Elapsed.TotalMilliseconds;
    frameClock.Restart();

    loop.Step(frameDeltaMilliseconds);
    Thread.Sleep(1);
}

coreManager.Deinitialize();
```

Ctrl+C ends the loop, and `coreManager.Deinitialize()` tears the session down. It's idempotent: calling it more than once, or from two places that both think they own shutdown, is safe.

One thing to know before the second hour

With this provider, your message handlers, connection-state callbacks, and anything else Nucleus calls run on the thread that calls `loop.Step`, which here is `Main`'s own thread. With the default provider they run on the network loop's own thread-pool thread instead, not the thread that constructed the `CoreManager`. If that's your UI thread or anything else that isn't thread-safe, marshal back to it yourself before touching it.

Your first networked scene

> **Driving the core API directly?** See Your first .NET host.

Every Unity scene that uses Nucleus needs exactly one `UnityCoreManager`. Everything else in this tutorial follows from that single component.

Add the core manager

Create an empty GameObject at the root of your scene and add `UnityCoreManager` to it. That is the only component you have to add: when the scene starts, it adds whichever of the other Unity manager components are missing. You can also add any of them to the same GameObject yourself, and that is how you set their Inspector fields, such as the tick rate and start mode below, before pressing Play. Keep them on that one GameObject, because `UnityCoreManager` only looks there before adding its own.

`UnityCoreManager` is marked `[DisallowMultipleComponent]`, so a scene can only ever have one on a given GameObject, and `[DefaultExecutionOrder(-10000)]`, so its `Awake` runs before every other script's. That ordering matters: other components resolve a ready `CoreManager` from their own `Awake`, and that only works if `UnityCoreManager` has already built one.

In its `Awake`, `UnityCoreManager`:

> Adds a `UnityNetworkLoopStepProvider`, so the network loop is driven from Unity's own update loop instead of the engine's default background-timer provider.
>
> Constructs the `CoreManager` from the tick rate and that step provider.
>
> Calls `EnsureManagers`, which adds any of the following components that isn't already present on the GameObject:
>
> - `UnityNetworkLoopManager`
> - `UnityTransportManager`
> - `UnityServerManager`
> - `UnityClientManager`
> - `UnitySystemManager`
> - `UnityInterestManager`
> - `UnityPacketManager`
> - `UnityMessageManager`
> - `UnitySceneManager`
> - `UnityPhysicsManager`

You author one component and end up with ten, each mirroring a manager on the core `CoreManager`. You never pick a network loop step provider yourself — `UnityCoreManager` already added one before the `CoreManager` existed to need it.

Set the tick rate

`UnityNetworkLoopManager` exposes a **Tick Rate** field in the Inspector. `UnityCoreManager.Awake` reads it before constructing the `CoreManager`, because the `CoreManager` bakes every rate-derived value in at construction and exposes no setter afterward. Set it before you press Play; changing it during play has no effect on the running session.

If no `UnityNetworkLoopManager` exists yet when `UnityCoreManager` wakes, the `CoreManager` falls back to the engine's default tick rate.

Choose how the connection starts

`UnityTransportManager` has an **Automatic Start Mode** field, backed by:

```
public enum NetworkStartMode
{
    None,
    Server,
    Client,
    Host,
}
```

It defaults to `None`, so nothing connects on its own — you choose Server, Client, or Host in the Inspector, and `UnityTransportManager.Start` calls the matching method for you when play begins.

To start the connection from your own code instead, call one of:

```
await unityTransportManager.StartServerAsync();
await unityTransportManager.StartClientAsync();
await unityTransportManager.StartHostAsync();
```

Each of these ensures a transport is added to the core manager first, then connects it in the matching role. `StartHostAsync` connects both a server and a client on the same transport, in the same process.

The default transport

`UnityTransportManager` needs at least one `NetworkTransport` component to connect. If you haven't added one yourself, it adds a `SynapseTransport` to the GameObject the first time a transport is needed, so the manager always has something to work with. Its defaults:

Field	Default
Port	`7777`
Remote Host	`127.0.0.1`
Maximum Transmission Unit	`1200`
Idle Timeout Seconds	`15`
Connecting Timeout Seconds	`10`
Host Pairing	`Local`

Host Pairing controls how a host reaches its own client half. `Local` pairs the two in process instead of routing that traffic over the socket, so a host's own connection comes up immediately.

Tearing down

When the `UnityCoreManager` GameObject is destroyed — including on leaving Play mode — its `OnDestroy` unwinds the integration binding first, then deinitializes the `CoreManager`. This runs before Unity finishes destroying the rest of the scene's GameObjects, so nothing is left driving objects that no longer exist.

Next steps

At this point you have a scene that can start a server, a client, or a host, and see the connection come up.

Building a dedicated server with no Unity in the process at all works differently — see Your first .NET host.

Component Index

Core managers

Component	Description
`UnityCoreManager`	Owns the process's single CoreManager and bootstraps every other `UnityManager` on its GameObject.
`UnityClientManager`	Inspector-side driver for the core client manager.
`UnityServerManager`	Inspector-side driver for the core server manager.
`UnityTransportManager`	Adds and configures the CoreManager's transport, and chooses whether the peer starts as a server, client, or neither.
`UnityNetworkLoopManager`	Carries the authored tick rate and gives the default physics world its driver.
`UnityPacketManager`	Inspector-side driver for the core packet manager.
`UnityMessageManager`	Inspector-side driver for the core message manager.
`UnitySystemManager`	Exposes the system manager's runtime knobs as serialized fields.
`UnityInterestManager`	Carries the world-wide interest rules, re-resolve interval, and per-tick spawn cap.
`UnityPhysicsManager`	Inspector-side seat for the process-global physics execution mode and fixed-step alignment.
`UnitySceneManager`	Exposes the scene manager's placement policy and supplies a scene loader if none is placed.

State

Component	Description
`NetworkTransform`	Replicates a GameObject's position, rotation, and scale, in local or world space.
`NetworkAnimator`	Replicates an `Animator`'s parameters, layer weights, and speed.
`NetworkNavMeshAgent`	Replicates a `NavMeshAgent` as the path leg it is walking rather than as its raw pose.

Scenes

Component	Description

`NetworkSceneLoader`	Base behaviour for loading networked scenes; registers itself as the peer's scene loader.
`UnitySceneLoader`	Loads networked scenes additively through Unity's own scene loading.
`NetworkSceneManifest` (asset)	Maps the scene identifiers named on the wire to the scenes this peer loads and the content bundle each ships in.

Physics

Component	Description
`PhysicsSimulationDriver`	Aligns one scene's physics world with the network tick instead of Unity's own FixedUpdate.
`ProjectedRigidbody`	Declares a rigidbody's replicated physics component and drives its sync, adopting, interpolating, or converging as needed.

Lag compensation

Component	Description
`UnityLagCompensationManager`	Inspector-side seat for lag compensation (Pro): the rewind window, whether 2D queries reach 3D hitboxes, whether queries start in hitboxes, and the spawn compensation cap.
`NetworkHitboxRoot`	Gathers every `NetworkHitbox` on a networked object and registers them with the server, so shots can be judged against where they were; disabling it makes the object unhittable.
`NetworkHitbox`	One lag-compensated volume, a sphere, box or capsule or a 2D circle, box or capsule, copied from a collider beside it or typed in.

Transports

Component	Description
`NetworkTransport`	Base for an Inspector-configurable transport component.
`SynapseTransport`	Inspector-configurable component for the Synapse UDP transport: port, remote host, and timeouts.
`BlitzRelayTransport`	Inspector-configurable component for the Blitz Relay transport, which carries a session through a relay.

Bundles

Component	Description
`NetworkBundleLoader`	Base behaviour for loading content bundles; registers itself as the peer's bundle loader.
`UnityAssetBundleLoader`	Loads content bundles as Unity asset bundles and registers the prefabs they ship.
`NetworkBundleManifest` (asset)	Maps the bundle identifiers named on the wire to the asset bundle files this client loads them from.

Systems and spawning

Component	Description
`NetworkSystemObject`	Marks the root of a networked prefab or scene object; the single link between the GameObject and its NetworkSystem.
`NetworkPrefabCollection` (asset)	A serialized, enumerable container of networked prefabs shipped with a content unit.
`NetworkPlayerSpawner`	Spawns one prefab per authenticated client and hands that client control of it.
`NetworkInterestObject`	Authors an object's interest rules once, for every NetworkSystem it carries.
`NetworkHostVisibility`	Hides an object's renderers on a host while its own client half would not have been holding the object.
`NucleusBehaviour<TComponent0>`	Base for a script whose lifetime is one networked system; exposes the typed component it carries and this peer's role and control.

Developer tools

Component	Description
`NetworkCorrectionVisualizer`	Flashes a line from where a predicted position was shown to where a reconcile corrected it, on the client that predicts the object.
`NetworkInterestOverlay`	While selected on a server or host, draws a line to every networked object, colored by what the object's controller holds of it.

Control in a Unity script

Driving the core API directly? See Reading who controls a system.

A `NucleusBehaviour<TComponent0>` runs on every peer that has the object in scene, not just the one that controls it. A script that assumes it is the controller breaks the moment a client owns the object instead of the server, or the moment it runs on a peer that only observes. Every method below exists to make that distinction explicit instead of accidental.

Checking control

Inherit `NucleusBehaviour<TComponent0>` (or the non-generic `NucleusBehaviour` for a script that needs the role and control checks but not a typed component, on an object where another component such as a `NetworkTransform` declares the system) and call `IsController(ControllerType)`:

```
public class Health : NucleusBehaviour<HealthComponent>
{
    private void ApplyDamage(int amount)
    {
        if (!IsController(ControllerType.AnyController))
            return;

        Component.Health.Value -= amount;
    }
}
```

`ControllerType` is a flags enum: `ControllerType.Server` means the local peer is the server and no client controls the system, `ControllerType.Client` means the local peer is the client that controls it, and `ControllerType.AnyController` (`Server | Client`) means this peer is the controller either way. Use `Server` or `Client` alone when the logic only makes sense for one of those, and `AnyController` when it only cares that this peer is the one in charge.

`IsController` returns false while no system is linked, so it is always safe to call, including before `OnSystemLinked` has fired.

The loud guards

`EnsureIsController(ControllerType)` and `EnsureIsStarted(Invoker)` answer the same question as `IsController` and `IsStarted`, but log a warning when the answer is false instead of letting the caller silently do nothing:

```
private void RequestFire()
{
    if (!EnsureIsController(ControllerType.Client))
        return;

    // fire
}
```

The warning names the calling member and is logged once per calling member on each behaviour, so a guard that keeps failing from the same place warns the first time and then stays quiet. `RequestFire` above is a one-off call from an input handler, where a failure is worth logging loudly because it means something is actually wrong. Reach for the loud forms there; reach for the plain `IsController` / `IsStarted` in a per-frame or per-tick hook, where a non-controlling peer failing the check is the normal, expected path rather than a mistake.

Reacting to control changing

Override `OnControllerChanged`:

```
protected override void OnControllerChanged(Connection previousControllerConnection, Conne
ction currentControllerConnection)
{
    // previousControllerConnection and currentControllerConnection are null for the serve
r
}
```

This fires on every peer when control of the linked system moves. It does not fire when a system links: linking adopts whichever roles are already started rather than waiting for an edge, so a peer that is already the controller when the object appears in its scene never sees a "became controller" transition for that arrival. Read `IsController` in `OnSystemLinked` if the initial state matters, not `OnControllerChanged`.

Role versus control

`IsStarted(Invoker)` reports whether this peer's server or client role is running, once a system is linked. `Invoker` only has `Server` and `Client` values — there is no host member, because a host is both roles running together rather than a third role. `IsHostStarted` covers that case directly.

A role check is not a peer check. A host runs every connected player's copy of the scene, so `IsStarted(Invoker.Server)` is true on the host for every object in the scene, not just the ones it controls. Use `IsController` to ask "does this peer control this object", and `IsStarted` / `IsHostStarted` to ask "is this role running here at all" — for example, gating server-only setup that has nothing to do with which client owns the object.

Reading NetworkSystem safely

`NetworkSystem` is null before the system links and after it unlinks. `OnSystemLinked` and `OnSystemUnlinked` are the only places guaranteed to see it non-null and stable: `OnSystemLinked` runs right after `NetworkSystem` is set, and `OnSystemUnlinked` runs while it is still readable, just before it is cleared. Reading `NetworkSystem` from `Awake` or from an inspector-triggered callback is reading it before the link has happened.

```
protected override void OnSystemLinked()
{
    // NetworkSystem is non-null here
}

protected override void OnSystemUnlinked()
{
    // NetworkSystem is still readable here, for the duration of this call
}
```

For the members `NetworkSystem` itself exposes, see the API reference.

Keeping a player's objects across a disconnect

> **Driving the core API directly?** See Controller retention.

By default, a disconnecting player's object is left in the world, uncontrolled, and nothing remembers who held it. Retention changes that: the object is also held on a record and handed back to the same player if they reconnect before the record expires. Two components decide whether that happens and how long it lasts, and a third can override it per prefab.

Unity System Manager: the project-wide default

The **Unity System Manager** component exposes three fields, under its **Controller Retention** heading, that configure `SystemManager` when the managers are instantiated. The last two are drawn only while the first is set to `Retain`:

- **On Controller Disconnect**: the retention policy applied to every controlled object whose system, and whose object, does not override it. Defaults to `ControllerRetentionPolicy.Release`, so retention is off project-wide until something turns it on.
- **Retention Seconds**: how long a retained set stays redeemable, in seconds. Defaults to 120. Zero never expires it.
- **Maximum Retained Sets**: how many retained sets may stand at once before the oldest is evicted. Defaults to 64.

These three map straight onto `SystemManager.DefaultControllerRetentionPolicy`, `ControllerRetentionSeconds`, and `MaximumRetainedControllerRecords`.

NetworkSystemObject: the per-prefab override

`NetworkSystemObject` carries its own retention toggle, for a prefab that needs to disagree with the project default. It has two fields:

- **Override On Disconnect** (`_controllerRetentionOverrideEnabled`): off by default. Unity cannot serialize the absence of an enum value, so this toggle is what lets "inherit the manager's default" be told apart from "I have an opinion."
- **On Controller Disconnect** (`_controllerRetentionPolicy`): drawn and read only when the toggle is on, and it defaults to `ControllerRetentionPolicy.Retain` once enabled. That default is deliberate: a script that turns the toggle on is doing so to hold the object, not to restate the manager's `Release` default.

Turning the toggle on writes the policy to the object's system group (`ControllerRetentionPolicyOverride`) the first time the group is rented, so the override rides with every system the object's group carries, not with one system in isolation.

NetworkPlayerSpawner: Disconnect Mode

`NetworkPlayerSpawner` has its own setting, **Disconnect Mode**, of type `NetworkPlayerDisconnectMode`:

- **`Despawn`** (default) — the player's object leaves the world with them. A returning player is always given a new one. Nothing is correlated, so there's nothing to hand to the wrong person.
- **`RetainForReturn`** — the player's object stays in the world, uncontrolled, and is handed back to the same player if they reconnect before the record lapses. The spawner marks every system it spawns `Retain`, and stores the retention token the departure issues against the connection's

`Connection.Identity`. When that identity reconnects, the token is redeemed and the object is handed back through the same spawner.

An object whose record expires, is evicted, or is redeemed by someone else is despawned; nobody is coming back for it.

The NAT warning

`RetainForReturn` is only as trustworthy as the identity behind it. The default `AddressClientAuthenticator` identifies a client by the address it connected from, and two players behind one NAT share that address. Under the default authenticator, `RetainForReturn` can hand the second player the first player's object. Use `RetainForReturn` only with an `IClientAuthenticator` that issues a real per-player identity.

What none of this does on a client

Controller retention is server bookkeeping: the Unity System Manager fields, the `NetworkSystemObject` override, and the `NetworkPlayerSpawner` Disconnect Mode all govern what the server does when a controller disconnects. A plain client can have every one of these fields set — they're serialized on the prefab and the scene components either way — but none of them does anything on a client. A client never decides retention for an object it doesn't control the fate of.

Where to go next

Tokens, the retention events, and reclaiming a retained object by hand are covered on the API page: Controller retention.

This page does not cover host migration or world adoption. For what happens to retained objects when the host itself is lost, see Surviving the loss of a host.

Control, Client State and Prediction

Pages in this category:

- Objects this peer does not control
- What control means
- Sending client state from Unity
- Control in a Unity script
- Giving a client control of an object
- Prediction settings on the Unity manager
- When prediction is worth its cost
- Keeping a player's objects across a disconnect
- Replaying a correction in Unity

Prediction settings on the Unity manager

Driving the core API directly? See Predicted members and prediction history.

The Unity system manager groups its prediction fields under **Prediction**. Each one is assigned onto the core `SystemManager` in `UnitySystemManager.ManagersInstantiated`, before any user script's `Awake` or `Start` can observe `SystemManager`, so script execution order cannot get ahead of them. Set them in the inspector before play.

Prediction History Ticks

The **Prediction History Ticks** field (backed by `_predictionHistoryTicks`) sets `SystemManager.PredictionHistoryTicks`: how many ticks of history `MemberRole.Predict` and `MemberRole.Validate` members keep, so a reconcile can restore any object to a past tick and a replay can read each past tick's client state.

```
NucleusSystemManager.PredictionHistoryTicks = _predictionHistoryTicks;
```

It defaults to `SystemManager.UnsetPredictionHistoryTicks`, zero. Left at zero, every client keeps `SystemManager.DefaultPredictionHistoryTicks` (64 ticks, about a second at 60 ticks per second) for every object it holds, controlled or not, which covers most games. The server, which never rolls back, keeps two ticks.

Sizing it

Set a value only when the default does not fit. It must cover the worst round trip in ticks plus **State Interpolation**, with headroom: a 250 ms round trip at 60 ticks per second is 15 ticks, so with an interpolation of 2 use 24 or more. A history that falls short logs a warning once per system saying the server's value was older than any tick its predicted history still holds; every outcome on that system then reconciles with `MismatchCause.HistoryMissing`.

A member takes the history when it initializes, so changing the field after objects have spawned does not resize the members that already exist. A member served on a paced send interval keeps a longer history on its own, sized from its send cadence, whatever this field says.

Maximum replay length

`SystemManager.MaximumReplayTicks` has no inspector field. Set it from code once the managers exist to cap how many ticks a reconcile replays; a longer one lands the server's values at its start tick without replaying and sets `ReplayReport.IsReplaySkipped`. Zero, the default, sets no limit.

Replay Hold Radius

The **Replay Hold Radius** field (backed by `_replayHoldRadius`) sets `SystemManager.ReplayHoldRadius` as the managers come up. When a reconcile restores the objects this client controls, every object it does not control is held at the reconcile's start tick while they replay; with a radius set, only the ones within it of a restored object are held. Distances are read through the interest position reader, so an object whose position cannot be read is always held, and only Pro reads positions. Zero, the default, holds every object. The inspector corrects a negative value to zero.

Send Result Values

The **Send Result Values** field (backed by `_resultValueSendEnabled`, on by default) sets `SystemManager.IsResultValueSendEnabled`. It is read on the controller: while it is on, the controller

sends its `MemberRole.Validate` result values beside the hash every tick, so a server that finds a result different can name the members and both values through `NetworkSystem.ResultMismatched`. The values cost bandwidth on every tick, so turn it off for a release build to keep the hash alone.

Predicted By Observers

The **Predicted By Observers** toggle sits under Prediction on the `NetworkSystemObject` inspector. Ticked, it sets `NetworkSystem.IsObserverPredictionEnabled` on every system as it links to the object, on every peer, so a client that observes the object predicts it as its controller does. It has an effect only on an observing client that itself controls an object sending client state. The object's simulation must carry missing client state with `TryGetLastState`, and each observing client pays a simulation per tick and a replay per reconcile for it. See Predicting another client's object.

Where to go next

For what a predicted member does with its history, see Predicted members and prediction history; for the replay that follows a correction, see Replaying a correction in Unity.

Objects this peer does not control

Most objects in a session run on peers that don't control them: every client watching another client's character, and the server itself when a client is the controller. Nothing about the network loop skips these objects. Every `NucleusBehaviour` hook still fires, every tick, on every peer — control only changes what the code inside the hook should do.

What still fires

`NucleusBehaviourBase` dispatches its per-step hooks (`OnEarlyVariableUpdate`, `OnEarlyStateUpdate`, `OnLateStateUpdate`, `OnReconcile`, `OnEarlyFixedUpdate`, `OnLateFixedUpdate`, `OnVariableUpdate`, `OnEarlySerialize`, `OnLateSerialize`, `OnTickAdvance`, `OnLateVariableUpdate`) whenever the concrete type overrides them, regardless of whether this peer controls the linked system. `OnControllerChanged` is the same: it runs on every peer the moment control moves, not just the one gaining or losing it.

Control is a question a hook answers for itself, with `IsController`:

```
protected override void OnEarlySerialize(StepDelta stepDelta)
{
    if (!IsController(ControllerType.AnyController))
        return;

    // Write gameplay state here; a non-controlling peer skips this and falls through to n
othing.
}
```

`IsController(ControllerType controllerType)` returns false whenever no system is linked yet, and otherwise the system's own answer. `EnsureIsController` is the loud version of the same check: it returns the same bool but also logs a warning the first time it fails from a given calling member. Neither of them stops the hook from running; a script that doesn't call one of them on a non-controlling peer does its writing work anyway, against state it doesn't own.

`IsStarted(Invoker)` is a separate axis entirely — server-started or client-started — and answers independently of who controls the object. A host can be `IsStarted(Invoker.Server)` and still not be the controller of a client-controlled object.

NetworkTransform on a proxy

`NetworkTransform` is written from wherever the controller is and applied everywhere else. Two inspector fields govern what a proxy does with it, both on by default:

- **Kinematic Management Enabled** (`_kinematicManagementEnabled`, default `true`) holds an attached `Rigidbody` kinematic on every non-controlling peer — including the server while a client controls the system — and returns it to simulation on the controller. The switch is a straight assignment, applied only when the role actually changes: `_rigidbody.isKinematic = !isController;`. Turn it off only when other code already owns that rigidbody's kinematic state.
- **Interpolate Scale Enabled** (`_interpolateScaleEnabled`, default `true`) has a proxy interpolate the replicated scale alongside position and rotation. Disabled, the local scale keeps whatever value it already holds — it's excluded from the sweep, not zeroed.

Both only matter on a peer that isn't the controller. On `VariableUpdate`, a non-controlling, started system calls `_transformComponent.Interpolate(_interpolateScaleEnabled)`; on `EarlySerialize`, a controller calls `_transformComponent.WriteState()`. A given peer runs exactly one side of that split

per tick, decided by the same `IsController(ControllerType.AnyController)` check every other hook uses.

Buffering: State Interpolation

`UnitySystemManager` exposes `_stateInterpolation`, defaulting to `1` in the inspector, and applies it as `NucleusSystemManager.StateInterpolation`. It's how many ticks are allowed to pass before a `StatePacket` can be applied. A proxy isn't drawing the instant a value arrives — it's drawing from a small buffer of recently-received ticks, which is what lets `NetworkTransform` interpolate between two real values instead of snapping to each one the moment it lands. Raising it trades latency for smoothness against jitter and loss; the default of `1` is the smallest buffer the inspector allows.

The subtick fraction

Interpolation doesn't jump from tick to tick — within a tick it sweeps, and the fraction it's swept by is the subtick percentage. It's owned by the engine-shared `NetworkLoopStepDriver`, not by the Unity loop provider:

```
public float SubtickPercentage => _tickIntervalMilliseconds <= 0f ? 0f : _elapsedTickUpdat
eTime / _tickIntervalMilliseconds;
```

`AdvanceEarly` publishes it right before `VariableUpdate` runs:

```
_networkLoopManager.SetSubtickPercentage(SubtickPercentage);

_networkLoopManager.InvokeNetworkLoopStep(NetworkLoopSteps.VariableUpdate, BuildStepDelta(
frameDeltaMilliseconds, hasFixedDelta: false));
```

Anything that reads it, `NetworkTransform`'s sweep included, reads it back through `NetworkLoopManager.SubtickPercentage`. `UnityNetworkLoopStepProvider` does **not** publish this itself; its own remarks are explicit that the subtick fraction for interpolation smoothing is not published from there, because publishing after its `Update` returned would leave `VariableUpdate` smoothing against the previous frame's fraction instead of the current one. What the Unity provider does own is the split around that shared driver: `Update`, at `[DefaultExecutionOrder(-10000)]`, runs the early steps; `LateUpdate` runs the serialize steps, `TickAdvance` and `LateVariableUpdate`.

A pose follower, not a simulation

A proxy isn't simulating anything. It's told where the controller's transform is — position, rotation, and, if enabled, scale — and it follows that pose. Velocity is never sent. That's also why kinematic management matters: while a rigidbody is kinematic, Unity's physics can't push back against a written pose, and nothing local — a collision, a force, a script nudging `transform.position` — should be moving a proxy either, since whatever it moves gets overwritten, or fought, on the very next tick it receives.

See NetworkTransform for the rest of its inspector fields, and NetworkMember for the per-member interpolation snap threshold that cuts a large discontinuity instead of sweeping across it.

Replaying a correction in Unity

Driving the core API directly? See Reconciliation and replay.

When a predicted member disagrees with the server, the engine restores the client's world to the server's values as of one start tick and replays every tick since then through your fixed updates. Put the simulation in `OnEarlyFixedUpdate` and the replay comes with it; there is no replay loop to write.

Simulate in OnEarlyFixedUpdate

`NucleusBehaviourBase` (the base every `NucleusBehaviour` script inherits) runs its two fixed-update virtuals once per simulated tick of the linked system:

```
protected virtual void OnEarlyFixedUpdate(StepDelta stepDelta, SimulationContext simulatio
nContext) { }
protected virtual void OnLateFixedUpdate(StepDelta stepDelta, SimulationContext simulation
Context) { }
```

`simulationContext.Type` says which pass the call belongs to:

- `Forward`: the first time a tick is simulated. On the server, once per server tick on one of the client's ticks for an object a client drives through client state, otherwise once per tick; on the controlling client, once per live tick.
- `Replayed`: a past tick simulated again during a reconcile, on the controlling client (or an observer predicting the object).
- `None`: nothing is simulated, on a client watching an object it does not control. The call still comes once per loop tick.

`stepDelta.Tick` names the local tick being simulated, and a replayed step carries one tick's worth of every delta. On the server that is the server's own tick, even for a step that runs one of the client's ticks. The context also carries the other end's tick:

- `simulationContext.RemoteTick`: on the server, the controller's tick whose client state is being simulated; on a client, the server tick of the state it applied. `NetworkLoopManager.UnsetTick` when there is none.
- `simulationContext.IsRemoteTickEstimated`: true when that tick was inferred rather than received, as on the server's step for a client tick whose client state did not arrive, and on every replayed tick.

Use `RemoteTick` to pair a log line with what the other end did, never for timing: on the server it is a number the client chose. Time cooldowns and everything else by `stepDelta.Tick`.

With the client state as `MemberRole.Predict` members and the result as `MemberRole.Validate` members, the component looks like this:

```
using Nucleus.Components;
using NumericsVector2 = System.Numerics.Vector2;
using NumericsVector3 = System.Numerics.Vector3;

namespace MyGame
{
    public partial class PredictedMoverComponent : NetworkComponent
    {
        // Client state: written by the controller each tick.
        public readonly NetworkMember<NumericsVector2> Direction = new(role: MemberRole.Pr
edict);
        // Computed from client state by the server and, ahead of it, by the controlling c
lient.
        public readonly NetworkMember<NumericsVector3> Position = new(role: MemberRole.Val
idate, accuracy: 0.01f, correctionSmoothingTicks: 6);
    }
}
```

Write the step once and let every pass run it:

```
using Nucleus.Integrations.Unity.Serializers;
using Nucleus.Integrations.Unity.Systems;
using Nucleus.Managers.NetworkLoop;
using Nucleus.Systems;
using UnityEngine;

namespace MyGame
{
    public class PredictedMover : NucleusBehaviour<PredictedMoverComponent>
    {
        [SerializeField] private float _speed = 5f;

        private Vector2 _frameDirection;

        private void Update()
        {
            if (Component is null)
                return;

            // Replace with your project's input system.
            if (IsController(ControllerType.AnyController))
                _frameDirection = Vector2.ClampMagnitude(new Vector2(Input.GetAxisRaw("Hor
izontal"), Input.GetAxisRaw("Vertical")), 1f);

            transform.position = Component.Position.InterpolatedValue.ToUnity();
        }

        protected override void OnEarlyFixedUpdate(StepDelta stepDelta, SimulationContext
simulationContext)
        {
            if (simulationContext.Type is SimulationType.None)
                return; // a client watching an object it does not control: nothing to sim
ulate

            // Live tick on the controller: the keys become this tick's client state. A re
play re-reads the recorded client state instead.
            if (simulationContext.Type is SimulationType.Forward && IsController(Controlle
rType.AnyController))
                Component.Direction.Value = _frameDirection.ToNative();

            // The server's step for a tick whose client state never arrived: carry the la
st tick's instead of the defaults.
            if (stepDelta.IsClientStateMissing)
                Component.TryGetLastState(isEveryMemberCarried: true);

            // Runs for the live tick, on the server once per server tick on one of the cl
ient's ticks, and again for every replayed tick.
            Component.Position.Value = Step(Component.Position.Value.ToUnity(), Component.
Direction.Value.ToUnity()).ToNative();
        }

        private Vector3 Step(Vector3 position, Vector2 direction)
        {
            float seconds = 1f / CoreManager.NetworkLoopManager.TickRate;

            return position + new Vector3(direction.x, 0f, direction.y) * (_speed * second
s);
```

```
        }
    }
}
```

The controller sends a hash of `Position` with each tick of client state, never the value itself. The server simulates the same tick and, when its own result differs, orders a reconcile. Inside a replayed tick the `Validate` members write that tick and the `Predict` members read the client state recorded for it, so the same lines rebuild history instead of moving the live object. A `Validate` member is quantized to its `accuracy` on every write, on both peers, so both simulate from the numbers the wire carries. Step by the fixed tick (`1f / CoreManager.NetworkLoopManager.TickRate`), never `Time.deltaTime`, so every pass of a tick computes the same value. No physics step runs between the early and late fixed update of a replayed tick: `CharacterController.Move` replays, a rigidbody does not (see the Physics category for `ProjectedRigidbody`).

The `ValidatedJump` demo under `Demos/Prediction` is a complete example: its movement lives in `ValidatedJumpMovementComponent.OnEarlyFixedUpdate`, and its behaviour only feeds it the keys and draws.

Play effects once

A replay re-runs every tick since the start tick, so play one-shot effects (sounds, particles, a jump animation) only on `SimulationType.Forward`. The behaviour mirrors the loop's `SimulationType` and `IsPredicting` (true on a client that replays this object) for code outside the two fixed updates.

While a replayed tick runs, the engine refuses calls that would act twice, logging once per kind: `TrySendRpc` returns false, `Connection.SendMessage` sends nothing, predicted spawns report `PredictedSpawnRefusal.Replaying`, collection and reference writes change nothing, and `NetworkAnimator.SetTrigger`, `ResetTrigger`, `CrossFade` and `Play` do nothing.

Place the object around a replay

The engine does not move your Transform. Override the replay hooks to put whatever shows the object at its member values, so replayed raycasts and `CharacterController.Move` calls start from the right place:

```
protected override void OnReplayStarting(ReplayReport replayReport)
{
    // Controlled: the restored start. Another player's object: its pose as of the start t
ick.
    transform.position = Component.Position.Value.ToUnity();
}

protected override void OnReplayCompleted(ReplayReport replayReport)
{
    // Controlled: the corrected present. Another player's object: back on its latest pose
.
    transform.position = Component.Position.Value.ToUnity();
}
```

They are called on a client, on every object the replay covers: each one it controls, restored to `replayReport.StartTick` and then replayed, and each one it does not control, held at the server's start-tick values for the replay and then returned to its latest values. Physics transforms are synced after each hook. A behaviour is subscribed only when its type overrides one of them.

Read what was corrected

`replayReport.Mismatches` lists the predicted members that were wrong at the start tick, each with its `MemberName` and `Cause`, and `replayReport.Corrections` lists the members whose present the replay moved. `NetworkMember<T0>.TryGetMismatchValues` gives both values at the start tick, and `TryGetPreRewindValue` gives the value that was on screen, until the next tick advance:

```
protected override void OnReplayCompleted(ReplayReport replayReport)
{
    foreach (MemberCorrection memberCorrection in replayReport.Corrections)
        Debug.Log($"[{memberCorrection.MemberName}] corrected from tick {replayReport.Star
tTick}.");
}
```

To glide to a correction instead of jumping, construct the member with `correctionSmoothingTicks: 6` and render `InterpolatedValue`: it eases from the value that was on screen while `Value` is already correct.

Hold a crowd, or predict other players

By default every object this client does not control is held at the start tick during a replay. Set **Replay Hold Radius** on the Unity system manager to hold only the ones near a restored object, or tick **Predicted By Observers** on a `NetworkSystemObject` to have observing clients predict and replay it instead of holding it. See Prediction settings on the Unity manager.

Replaying from ReconcileRequired yourself

A game that must own the replay loop can still subscribe to `NetworkSystem.ReconcileRequired` and replay in `OnReconcile`, which runs every tick on the `Reconcile` step. Do this only for an object nothing simulates in `OnEarlyFixedUpdate`/`OnLateFixedUpdate`; otherwise the engine has already replayed it.

```
private uint _pendingRewoundTick = NetworkLoopManager.UnsetTick;

protected override void OnSystemLinked() => NetworkSystem.ReconcileRequired += OnReconcile
Required;

protected override void OnSystemUnlinked() => NetworkSystem.ReconcileRequired -= OnReconci
leRequired;

private void OnReconcileRequired(uint rewoundTick)
{
    if (rewoundTick != NetworkLoopManager.UnsetTick)
        _pendingRewoundTick = rewoundTick;
}

protected override void OnReconcile(StepDelta stepDelta)
{
    if (_pendingRewoundTick == NetworkLoopManager.UnsetTick)
        return;

    uint currentTick = CoreManager.NetworkLoopManager.Tick;

    for (uint tick = _pendingRewoundTick + 1; tick < currentTick; tick++)
    {
        NetworkSystem.BeginReplayTick(tick);

        // Re-run this tick's step: Predict members already read this tick's client state.

        NetworkSystem.EndReplayTick();
    }

    _pendingRewoundTick = NetworkLoopManager.UnsetTick;
}
```

`ReconcileRequired` is raised at the head of the `Reconcile` step, after the restore, so `OnReconcile` runs after it on the same step. `rewoundTick` is the start tick: the server's value there already includes that tick's client state, so replay from the tick after it. It is `NetworkLoopManager.UnsetTick` when a server `SendReconcile` landed with no echo of the processed client state tick; skip the replay then. Bracket every past tick with `BeginReplayTick` / `EndReplayTick`, in tick order: a write to a predicted member outside the bracket lands on the live value and breaks the next comparison.

A server correction of your client state alone does not raise `ReconcileRequired`. The server's value lands in that tick's history and the component raises `OnMembersChanged` with `MemberChangeDirection.Correction` instead.

See also

- The restore, the report and every refusal are described in full in Reconciliation and replay.
- For a controlled rigidbody, physics correction has its own convergence path; see the Physics category for `PhysicsConvergence`.

What control means

Every `NetworkSystem` has three questions attached to it: who is the server, who controls it, and who observes it. They are answered separately, and mixing them up is the most common source of "why did that work on my machine but not over the network" bugs.

The three roles

The server is the server. `NetworkSystem` exposes `IsServerStarted` for this; the server's word on a system's state is final, whatever any client believes.

The controller is the one connection - server or a single client - that drives the system. Control is exclusive: at most one connection controls a given system at a time. `ControllerType` names who that can be:

```
[System.Flags]
public enum ControllerType : byte
{
    None = 0,
    Server = 1 << 0,
    Client = 1 << 1,
    AnyController = Server | Client,
}
```

Ask which one applies to the local peer with `IsController`:

```
if (networkSystem.IsController(ControllerType.Client))
{
    // This peer is the client controlling this system.
}
```

`ControllerType.Server`'s own meaning is exact: "the local peer is the server and no client controls the system." A system is never controller-less from the server's point of view - if no client controls it, the server does.

An observer is any peer the system is replicated to. Observation is separate from both of the above: the server and the controller are single, named connections, while a system can have any number of observers, including none.

Control is exclusive and server-assigned

Only the server assigns a controller, through `SetController`. A client that calls it is rejected outright - the server logs an error and nothing changes. Control replicates from there: every observer is told who the controller is, and `IsController` reads that replicated identity rather than anything a client could assert about itself.

Because control is exclusive, a system is never controlled by two clients at once, and a controller change is a single, well-defined event (`ControllerChanged`) rather than a negotiation.

What control decides

Three things ride on who the controller is, and all three are controller-only regardless of any write-access setting:

- **Who may send client state.** Only the controller's client state is accepted for the system.
- **Who predicts.** The controlling client predicts the system from its own client state, and the server checks

its result and corrects it when they differ. Any other client receives the server's state instead, and during a reconcile holds the system at the server's values while its own systems replay, unless it opts into predicting it as well with `NetworkSystem.IsObserverPredictionEnabled`.

- **Who may write state by default.** `StateWriteAccess.Controller` - "the server, or the single controlling client" - is the default access, and it is also the behavior every system had before write access existed as a setting at all. The controller can always write state; widening that to other observers is a separate, additive Pro setting (`StateWriteAccess.AnyClient`), covered on Letting clients write state.

The same shape holds for RPCs: `RpcSendAccess.Controller` is the default there too, and its own doc comment puts it plainly - "only whoever drives the object may ask it to do anything."

Observation is required under every check

Control and write access decide what a connection is *permitted* to do once it is already being told about the system. Being told about it - observation - is checked first, unconditionally, before either of them is consulted. The server asks the same three questions of a client's state write and of a client's remote call:

> Does the connection observe the system? If not, it is refused.
>
> Is it the system's controller (`ControllerConnectionId`)? If so, it is admitted.
>
> Otherwise it is admitted only when the access in force is the any-client one (`StateWriteAccess.AnyClient` or `RpcSendAccess.AnyClient`).

State writes and remote calls both go through this same check. A client that does not observe a system fails it immediately, regardless of whether it is the controller or the write access has been widened to any client. Widening control or write access never substitutes for being an observer - a connection has to be told about a system before anything it does about that system means anything. Observation itself is decided by the interest system, covered on What area of interest is.

The host case

A host runs the server and a client in the same process. That peer's `IsServerStarted` is true, so it is the server. If it also controls a given system - the ordinary case for a host's own player object - it satisfies `IsController(ControllerType.Client)` as well. And because a host holds every system in its own process, its own peer is exempt from being culled by interest: it observes everything by construction.

So on a host, one process can be server, controller and observer for the same system simultaneously. That collapses three checks that are independent everywhere else into one that always passes, which is exactly what hides the bugs that show up the moment a second machine joins: a missing observer registration, a write-access setting that was never actually exercised, a client state path that assumed the controller and the server were the same connection. None of those fail on a host, because on a host they can't - there is only one peer to be all three roles at once.

Giving a client control of an object

Driving the core API directly? See Assigning and releasing control.

Calling SetController

`SetController` is a member of the networked system, not of the `NucleusBehaviour`, so a component calls it through `NetworkSystem`. It is server-only: calling it from a client is rejected and logged. Guard the call with `EnsureIsStarted(Invoker.Server)` so a misplaced call fails loudly instead of silently doing nothing on a client.

```
if (EnsureIsStarted(Invoker.Server))
    NetworkSystem.SetController(connection);
```

Passing `null` returns control to the server.

The built-in case: NetworkPlayerSpawner

`NetworkPlayerSpawner` is the component to reach for when every authenticated client needs one object of its own. It exposes two inspector fields:

- `_playerPrefab` - the networked prefab spawned for each client. It needs a `NetworkSystemObject` marker.
- `_spawnPoints` - the poses new objects are placed on, taken in turn and wrapping when exhausted. Leave this empty to spawn every player on the spawner's own transform.

On the server, each authenticated client gets exactly one object of this prefab. When a fresh spawn is needed, the spawner calls `Instantiate(_playerPrefab, spawnPose.position, spawnPose.rotation)` and then hands the client control with `SetController`. The spawner only ever creates objects while its peer is the server; a client never spawns one for itself.

What each peer sees

Control changes are announced through `NetworkSystem.ControllerChanged`, and `NucleusBehaviourBase.OnControllerChanged(previousControllerConnection, currentControllerConnection)` forwards it to every peer that has the system linked:

- The previous controller (if any) sees its `OnControllerChanged` fire with the new controller as `currentControllerConnection`, and loses whatever control-gated behavior it was running.
- The new controller sees the same callback, with itself as `currentControllerConnection`, and can now act as this system's controller.
- Every other observer - every peer with the object in interest, controller or not - sees the same callback fire, with both connections filled in as far as that peer can resolve them.

A `null` connection in either parameter means either the server or a peer client this peer holds no `Connection` for: control returning to the server reports `currentControllerConnection` as null, and an object spawned already under a client's control reports `previousControllerConnection` as null on a peer that cannot yet resolve who that was.

Retention policy is set on every object the spawner creates

Before calling `SetController`, the spawner's `AssignController` writes `ControllerRetentionPolicyOverride` on every `NetworkSystem` the object carries:

```
ControllerRetentionPolicy controllerRetentionPolicy = _disconnectMode is NetworkPlayerDisc
onnectMode.RetainForReturn
    ? ControllerRetentionPolicy.Retain
    : ControllerRetentionPolicy.Release;

foreach (NetworkSystem playerNetworkSystem in playerNetworkSystemObject.Systems)
{
    playerNetworkSystem.ControllerRetentionPolicyOverride = controllerRetentionPolicy;
    playerNetworkSystem.SetController(connection);
}
```

This happens for every system the spawned object carries, not just the first, and it happens whether the disconnect mode is `Despawn` or `RetainForReturn`. It overrides whatever the manager's default retention policy is - a world configured to retain by default would otherwise retain a player object the spawner means to despawn on disconnect, and the two would disagree about what happens to it.

The Instantiate trap

Placing a controlled object with `Instantiate(prefab, position, rotation)` works in the ordinary case - that is exactly what `NetworkPlayerSpawner` does. The trap is narrower, and specific to an object whose *pose* the controller itself simulates, such as a `ProjectedRigidbody`. Once that component links to its system, the controller's own simulation asserts its projected pose back onto the object within a tick, overwriting whatever position and rotation `Instantiate` set.

This is easy to miss on a host, because the host's own client is often the same peer whose simulated pose is being asserted - the overwrite lands on the pose the host expected anyway, so nothing looks wrong until a second peer, whose simulation starts from the true placement, shows the object snap.

The spawner's public surface

- `EnsurePlayerObject(Connection connection)` - the only way an object is created for a client. Idempotent: a client that already has one is a no-op.
- `TryGetPlayerObject(uint connectionId, out NetworkSystemObject playerNetworkSystemObject)` and the `TryGetPlayerObject(Connection connection, ...)` overload - look up the object this spawner holds for a client.
- `PlayerObjectCount` - how many clients this spawner currently holds an object for.
- `PlayerObjectSpawned` - raised on the server when a client is given a newly created object, after control has been assigned.
- `PlayerObjectReclaimed` - raised when a client is given an object that already existed, whether redeemed from a retention record or found already under that client's control after a world adoption. The object may carry state from its previous controller, so any per-player reset a game needs must run here as well as on `PlayerObjectSpawned`.
- `PlayerObjectDespawned` - raised when a player object leaves the world, whether its player disconnected under `NetworkPlayerDisconnectMode.Despawn` or a retention record lapsed unredeemed.

Arbitration rules

For how `SetController` resolves conflicting or repeated calls, see Assigning and releasing control.

When prediction is worth its cost

What prediction actually buys

Prediction removes one thing: the wait for a round trip before the controller sees the result of its own client state. Without it, a client presses a key, sends the resulting client state, and only moves once the server's outcome comes back. With it, the client applies that client state locally the moment it is written and the server's outcome arrives later to confirm or correct that guess.

That is the entire benefit. Prediction does not make an object more accurate, more responsive to other peers, or cheaper to replicate. It only hides the round trip from the one peer driving the object.

Prediction costs bandwidth, memory and replay time

Turning prediction on for a `NetworkMember<T0>` is one constructor argument, its role:

```
public NetworkMember(CompressionLevel deltaCompressionLevel = CompressionLevel.Tight, bool
 isAutomaticPoolingEnabled = false, MemberRole role = MemberRole.State, float accuracy = f
loat.NaN, float interpolationSnapThreshold = float.NaN, SendInterval sendInterval = SendIn
terval.Normal, TransmissionMode transmissionMode = TransmissionMode.Interval, bool isExtra
polationEnabled = true, byte correctionSmoothingTicks = 0, bool isCarriedForward = true)
```

A result declared `MemberRole.Validate`, computed from client state declared `MemberRole.Predict`, is predicted by the controller and checked by the server. That does three things, and each one is a real cost:

- **A history, not just a live value.** A predicted member keeps a history deep enough to cover the round trip, so a reconcile can restore the member to a past tick and a replay can read each past tick's client state. Every client keeps `SystemManager.PredictionHistoryTicks` of it, or `SystemManager.DefaultPredictionHistoryTicks` (64) when that is unset, and a member served on a wider interval than the tick rate keeps more. An unpredicted member keeps two slots.
- **A replay whenever a prediction misses.** A matching result costs nothing beyond its hash on the upstream client state, plus the result values beside it while `SystemManager.IsResultValueSendEnabled` is on (the default, so the server can name what differed; turn it off to keep the hash alone). A result the server finds different orders a reconcile, which restores every system the client predicts to the server's values as of the wrong tick, and the engine re-runs the fixed updates (`OnEarlyFixedUpdate` and `OnLateFixedUpdate`, as `SimulationType.Replayed`) for every tick from there to the present. A replay covers about a round trip plus the state interpolation in ticks, all simulated in one frame.
- **Game code that tolerates being re-run.** Anything that happens during a replayed tick happens again, for every tick between the rewind point and the present. That is fine for a movement integration; it recomputes the same answer. It is not fine for a side effect that cannot be undone or repeated safely, so effects belong on `SimulationType.Forward` only. The engine refuses the worst offenders while a replayed tick runs: remote calls, messages, predicted spawns, collection and reference writes and `NetworkAnimator` triggers. A rigidbody cannot be re-stepped per object at all; use `ProjectedRigidbody` for bodies.

The cheaper alternatives

Most objects do not need any of this. In order of cost:

- **An uncontrolled object with interpolation.** No controller, no ring beyond the ordinary two slots, no result check, no replay. The receiver just animates between the values it receives, paced by

`SystemManager.StateInterpolation` (0 by default, capped at `MaximumStateInterpolation`). This is the default for anything nobody is driving.

- **A controller-written value with no prediction.** The controller still writes the member every tick, but its role stays `MemberRole.State`. The controller sees its own writes with the same round-trip delay as everyone else; no local guess, no rewind, no replay. Cheaper than prediction, more responsive than a purely server-driven value the controller doesn't write at all.
- **A purely local visual that is never replicated.** Not a `NetworkMember<T0>` at all. Particle effects, camera shake, UI feedback: anything that only needs to look right on the machine that triggered it costs nothing on the wire and needs no reconciliation, because there is nothing to reconcile.

The rule that disqualifies a candidate

If the value only ever needs prediction because something *reads* it (a position feeding a camera, a resource bar), the reconcile replay problem is contained: worst case, the display corrects itself. Prediction turns dangerous the moment the tick being replayed can trigger a side effect the replay can't take back: spawning an object, awarding score, firing an RPC. The engine refuses remote calls, messages and predicted spawns during a replay, but it cannot know about your own side effects: awarding score or writing to a static from a replayed tick happens again on every replay. Gate side effects on `SimulationType.Forward`, or keep the member driving them unpredicted.

Why most of a world has nothing to predict

Prediction only matters to the peer that controls the object, because it's the only peer waiting on its own round trip. `NetworkMember<T0>` checks this directly: a predicted result skips its own upstream serialization on a pure client that predicts its system, which is the controller (`NetworkSystem.IsController(ControllerType.Client)` and not server-started), or an observer with `NetworkSystem.IsObserverPredictionEnabled` set, because its value is computed locally from client state and checked by the server rather than sent. An object nobody controls has no local guess to make in the first place: there's no round trip to hide, because there's no client state driving it locally. Most objects in a networked world fall into that category: static geometry, background actors, anything the server alone moves. That's the uncontrolled-plus-interpolation case above, and it's the default, not the exception.

The choice is per member

`role` is a per-`NetworkMember<T0>` constructor argument, not a project setting or even a per-object one. A single component can predict its position while leaving its health, ammo count, or any other member as ordinary `MemberRole.State`; each one is its own opt-in. No shipped component turns prediction on; the Unity demo `Demos/Prediction/ValidatedJump` shows it, with the walk direction and jump press count of `ValidatedJumpMovementComponent` as `MemberRole.Predict` client state and every movement result as `MemberRole.Validate`. Prediction is available, per member, wherever the predict-and-replay cost is worth paying, but nothing turns it on by default.

Predicted spawning is Pro, predicted motion is not

Predicted members, their client state, and the reconcile replay described above are available in every build. Predicted *spawning*, predicted *despawn*, and spawn compensation are Pro-only: `NetworkSystem.PredictedSpawn.Pro.cs`, `SystemManager.PredictedSpawn.Pro.cs`, `NetworkSystem.SpawnCompensation.Pro.cs`, and `SystemManager.SpawnCompensation.Pro.cs` don't exist in a Free build. Predicting an object's motion or state costs nothing extra to license; predicting the moment an object comes into or out of existence does.

Sending client state from Unity

> **Driving the core API directly?** See Client state

Client state is the one kind of data a client sends upstream every tick: the client that controls an object writes what its player asked for (a move direction, a jump press), and the server simulates the object from it. This page sends a keyboard or gamepad direction from a `NucleusBehaviour` and moves the object on the server.

Declare the client state in the component

Client state is ordinary `NetworkMember<T0>` fields with `role: MemberRole.Predict`, declared in the object's own `NetworkComponent` beside the state the server computes from them. There is no separate class to write, and nothing to add or register when the object links: the members travel because of their role.

```
using Nucleus.Components;
using Nucleus.Connections;
using NumericsVector2 = System.Numerics.Vector2;
using NumericsVector3 = System.Numerics.Vector3;

namespace MyGame
{
    public partial class MoverComponent : NetworkComponent, IStateWriteValidator<NumericsV
ector2>, IStateWriteValidator<NumericsVector3>
    {
        // Client state: written by the controller on its live tick, sent upstream every t
ick.
        public readonly NetworkMember<NumericsVector2> Direction = new(role: MemberRole.Pr
edict);
        // State only the server computes from it.
        public readonly NetworkMember<NumericsVector3> Position = new();

        // Runs on the server for each tick of client state it takes. Write the member and
 decline to keep your own value instead.
        public StateWriteAction ValidateStateWrite(NetworkMember<NumericsVector2> networkM
ember, NumericsVector2 proposedValue, Connection writingClientConnection)
        {
            float length = proposedValue.Length();

            if (length <= 1f)
                return StateWriteAction.Accept;

            Direction.Value = proposedValue / length;

            return StateWriteAction.Decline;
        }

        // Only the server moves this object, so a client write to the position is decline
d.
        public StateWriteAction ValidateStateWrite(NetworkMember<NumericsVector3> networkM
ember, NumericsVector3 proposedValue, Connection writingClientConnection) => StateWriteAct
ion.Decline;
    }
}
```

The validator runs on the server once per Predict member for every client tick it takes: accept the value,

decline it to keep what the member holds, or write the member and decline to keep your own, as the direction clamp does. A validator that throws declines. A press that must not repeat across a lost tick is declared with `isCarriedForward: false` (see below).

Read the device in Update, write it on the tick

Read the keys from Unity's `Update`, into a field: a held key is a property of the frame, so it has to be read every rendered frame. Then copy that field into the Predict member inside `OnEarlyFixedUpdate`, on the controller's live tick only. That tick is `SimulationType.Forward`; a replay reads each past tick's value from the member's history and must never overwrite it.

```
using Nucleus.Connections;
using Nucleus.Integrations.Unity.Serializers;
using Nucleus.Integrations.Unity.Systems;
using Nucleus.Managers.NetworkLoop;
using Nucleus.Systems;
using UnityEngine;

namespace MyGame
{
    public class ServerMover : NucleusBehaviour<MoverComponent>
    {
        [SerializeField] private float _speed = 5f;

        private Vector2 _frameDirection;

        protected override void OnSystemLinked()
        {
            if (IsStarted(Invoker.Server))
                Component.Position.Value = transform.position.ToNative();
            else
                transform.position = Component.Position.Value.ToUnity();
        }

        protected override void OnSystemUnlinked() => _frameDirection = Vector2.zero;

        private void Update()
        {
            // Replace with your project's input system.
            if (IsController(ControllerType.Client))
                _frameDirection = Vector2.ClampMagnitude(new Vector2(Input.GetAxisRaw("Hor
izontal"), Input.GetAxisRaw("Vertical")), 1f);
        }

        protected override void OnEarlyFixedUpdate(StepDelta stepDelta, SimulationContext
simulationContext)
        {
            if (simulationContext.Type is SimulationType.None)
                return;

            // The controller's live tick: the keys become this tick's client state.
            if (simulationContext.Type is SimulationType.Forward && IsController(Controlle
rType.Client))
                Component.Direction.Value = _frameDirection.ToNative();

            if (!IsStarted(Invoker.Server))
                return;

            // A tick whose client state did not arrive: keep the last direction rather th
an stopping.
            if (stepDelta.IsClientStateMissing)
                Component.TryGetLastState(isEveryMemberCarried: true);

            Component.Position.Value = Step(Component.Position.Value.ToUnity(), Component.
Direction.Value.ToUnity()).ToNative();
        }

        protected override void OnVariableUpdate(StepDelta stepDelta) => transform.positio
```

```
n = Component.Position.InterpolatedValue.ToUnity();

        private Vector3 Step(Vector3 position, Vector2 direction)
        {
            float seconds = 1f / CoreManager.NetworkLoopManager.TickRate;

            return position + new Vector3(direction.x, 0f, direction.y) * (_speed * second
s);
        }
    }
}
```

Do not put a `NetworkTransform` on the same object; this script replicates the position itself. Step by the fixed tick (`1f / CoreManager.NetworkLoopManager.TickRate`), never `Time.deltaTime`, so every peer integrates identically.

What the server does with it

A pure client sends its Predict members to the server every tick; a member that did not change costs one bit. The server takes one of the client's ticks per server tick, through **State Interpolation**, runs the validator, then runs the object's fixed updates with the members holding that tick's values. On a host playing its own object the host writes the members and steps with them in the same tick, so nothing travels upstream, but each value still goes through the same validator.

The object is stepped on every server tick, even when that tick's client state never arrived. On that step `stepDelta.IsClientStateMissing` is true and the Predict members read their defaults until the component's `TryGetLastState(isEveryMemberCarried)` carries the previous tick's. Declare a press with `isCarriedForward: false` so `TryGetLastState(false)` leaves it at its default, or send a running press count, which repeats nothing when carried. The overload `TryGetLastState(isEveryMemberCarried, out SimulationContext lastStateSimulationContext)` also reports which tick the carried state came from: on the server its `RemoteTick` is the client tick that state was written on, so `simulationContext.RemoteTick - lastStateSimulationContext.RemoteTick` is how many ticks it has been carried, for example to stop a character walking once the client has been silent too long. Late and dropped ticks are counted on the controller's `Connection` as `ClientStateLateCount` and `ClientStateDroppedCount`.

The server's step runs on the server's own tick (`stepDelta.Tick`). `simulationContext.RemoteTick` names the client's tick it is simulating, for logging and pairing only: never time a cooldown by it, since a client picks its own tick numbers.

React to client state

The server forwards the values it kept, after validation and its own writes, to every observer of the object, the controller included. `OnMembersChanged` on the component reports them on change, not every tick:

- `MemberChangeDirection.Read` on the server when a client's changed value is taken (`TryGetWritingClient` names the sender), and on an observer receiving a changed value. Drive presentation such as footsteps or animation from it.
- `MemberChangeDirection.Correction` on the controller when the server's value for one of its past ticks differs from its own, as when the validator clamped it. The server's value is already in the member's history at that tick.

```
public override void OnMembersChanged(ulong memberFlags, MemberChangeDirection memberChang
eDirection)
{
    if (memberChangeDirection is MemberChangeDirection.Correction && (memberFlags & (ulong
)MoverComponentFlags.Direction) != 0)
        UnityEngine.Debug.Log($"The server corrected the direction to {Direction.Value}.")
;
}
```

Put this override in `MoverComponent`. The simulation reads the members every tick anyway; use the callback for presentation or to react to a correction.

Predict the result as well

Here only the server moves the object, so the controller sees its own movement a round trip late. To have it move at once, declare `Position` with `role: MemberRole.Validate`, run the step on the controller too, and let the engine replay it when the server finds the result different. See Replaying a correction in Unity.

For the server's side in more depth, see Client state.

Host mode

What a host is

A host is one process running both roles at once: it starts a server and connects its own client to that server. `TransportManager.IsServerStarted` and `TransportManager.IsClientStarted` are both true, and `IsHostStarted` is defined as `IsServerStarted && IsClientStarted`. Nothing else distinguishes host mode from a pure server or a pure client: the same `CoreManager`, the same transport, both halves live in one process.

A host holds one Connection for its own player

A host's own client is described by one `Connection`. The server side keeps a roster entry for every connected player, and for the host's own client that roster entry is the client side's local `Connection` itself, under either Synapse pairing. A call that names the host's own player by its local `Connection`, such as `RpcTarget.To` or `RpcTarget.ObserversExcept`, therefore matches the roster entry the server holds.

Compare players by `Connection.Id` all the same:

```
bool isSelf = someConnection.Id == localClientConnection.Id;
```

A `Connection` is pooled and reused once its link ends, so a reference kept past that point names whoever rents the instance next, while the Id is what names a player on the wire.

Or resolve each side explicitly through the `TransportManager` rather than holding a reference across the two halves:

```
transportManager.TryGetLocalClientConnection(out Connection localClient);
transportManager.TryGetServerConnection(out Connection localServer);
```

`TryGetLocalClientConnection` returns this peer's own client `Connection`; it is false on a pure server. `TryGetServerConnection` returns this peer's own server `Connection`; a controlling client uses it to find the upstream observer target. Both resolve to the owner-held instance for that side rather than scanning transports.

IsHostLoopback

Both halves of the pairing above are marked `Connection.IsHostLoopback`, and the client half is also the server's roster entry for its own client. The mark identifies a link whose two ends are the same process. It is neither `IsEmulated` (no socket at all) nor `IsPeerStandIn` (a peer this client cannot reach directly) — it names a real, live link where both ends happen to be this peer.

State addressed across a loopback link is discarded on arrival rather than applied: whatever was sent was written by the process it would be delivered to, so there is nothing to learn from serializing it and reading it back. Neither half owes the other an acknowledgment either; the sending half settles its own outstanding ticks as it serializes them.

A `Transport` answers the same question at the socket level, before a `Connection` can be marked at all, through the abstract `Transport.IsOwnClientConnection(Connection serverSideClientConnection)`. It exists because a `Connection` itself carries no endpoint or handshake to prove the two ends are the same process — only the transport underneath it knows what it's holding. A transport that cannot tell must answer false rather than guess: a false answer costs a host the in-process delivery of its own traffic, but a wrong true answer would hand a stranger connecting over loopback this peer's own server half.

HostLoopbackDelivery

`TransportManager.HostLoopbackDelivery` controls when a host's two halves see what one of them addressed to the other. It is settable only to `HostLoopbackDelivery.Immediate`; setting anything else is refused with a `LogError` rather than silently ignored:

```
// Refused: logs an error and leaves HostLoopbackDelivery at Immediate.
transportManager.HostLoopbackDelivery = HostLoopbackDelivery.Deferred;
```

`Immediate` (the default, and the only value a game may set) runs the receiving half's handlers at the send site, before the send call returns — the same shape every other callback in the engine already uses. `Deferred` would run those handlers a tick later, at the loop step a genuinely remote copy lands on; it exists only so the engine's own tests can measure that alternative, reached through an internal setter game code cannot call.

A message delivered this way is never serialized. Under `Immediate` its handler runs synchronously at the send site, with nothing queued. `Deferred` queues it instead, and the queue is run at `NetworkLoopSteps.EarlyVariableUpdate`, the same step a wire copy would be read on.

Callbacks a host raises for itself

`TransportManager.PeerConnectionDiscovered` and `PeerConnectionDropped` normally hand a client a stand-in Connection (`Connection.IsPeerStandIn` true) for a peer it was told about. On a host these events instead name the server's own live roster entry for a joining or leaving peer — `IsPeerStandIn` reads false, because the Connection carries a real link rather than a bare identity. A host is told through `RaiseHostPeerConnectionDiscovered` / `RaiseHostPeerConnectionDropped` rather than through the roster, because the roster message it writes to its own client is the one discarded on its own loopback.

Neither event is ever raised for the host's own client. No client is told about itself.

The self-delivery hazard

Never pair an addressed send (`RpcTarget.To(connection)`) with immediate self delivery (`RpcSelfDelivery.Immediate`) on a surface the host's own client also listens on. On a host, the process running the server is the same process running that client, so a handler registered for the surface runs once for the send and once more for the host's own copy — the host ends up running every player's copy of a call meant for one. See the RPC page for the detail on `RpcSelfDelivery` and `RpcTarget`.

Starting a host

In Unity, set `UnityTransportManager`'s automatic start mode to `NetworkStartMode.Host` in the inspector, or call `StartHostAsync()` directly:

```
await unityTransportManager.StartHostAsync();
```

`StartHostAsync` connects every added transport as both a server and a client, in that order.

From plain C#, connect both invokers on the transport:

```
await transport.ConnectAsync(Invoker.Server);
await transport.ConnectAsync(Invoker.Client);
```

Core, Managers and the Tick Loop

Pages in this category:

- The Nucleus Core Manager component
- Tick rate and the loop in Unity
- Host mode
- The network loop steps
- Starting and stopping a session in Unity
- Manager components reference
- The manager map
- Reacting to session events in Unity
- Play mode, domain reload and session lifetime

The manager map

The eleven managers

`CoreManager` owns eleven managers as public fields, constructed once and held for the life of the session. Each one is documented in full on its own pages elsewhere; this page only says which manager owns which concern and how to reach it.

Manager	Owns
`NetworkLoopManager`	The tick loop: tick rate and the steps that drive a tick forward.
`TransportManager`	The transports a session sends and receives over.
`ServerManager`	Server-side session state.
`ClientManager`	Client-side session state.
`SystemManager`	Networked systems: spawning, ownership, and their lifecycle.
`InterestManager`	What each peer is allowed to see.
`SceneManager`	Scene load, stacking, and per-scene state.
`PacketManager`	Packet framing and dispatch.
`MessageManager`	Messages sent between peers.
`RpcManager`	Remote procedure calls.
`ViolationManager`	Detecting and acting on protocol violations.

The two Pro-only managers

`BundleManager` and `WorldPersistenceManager` are not part of the base `CoreManager`. They are added by Pro partial classes - `CoreManager.Bundles.Pro.cs` and `CoreManager.Persistence.Pro.cs` - each of which declares the field and fills in a partial hook (`AddBundleManager`, `AddWorldPersistenceManager`) that the base constructor calls unconditionally. In a Free build neither partial exists, the hooks compile away to nothing, and `CoreManager` simply has no `BundleManager` or `WorldPersistenceManager` field at all - not null, absent.

`BundleManager` owns content delivered from outside the build (Addressables, a patcher, a CDN). `WorldPersistenceManager` owns writing a world to disk and building it back.

Reaching a manager

From C#, every manager is a public field directly on `CoreManager`: `coreManager.TransportManager`, `coreManager.SystemManager`, and so on, including the two Pro fields when they're present.

From Unity, nine of the eleven have a matching `Unity*Manager` component with a `Nucleus*Manager` property that returns the same instance:

Component	Property
`UnityNetworkLoopManager`	`NucleusNetworkLoopManager`

`UnityTransportManager`	`NucleusTransportManager`
`UnityServerManager`	`NucleusServerManager`
`UnityClientManager`	`NucleusClientManager`
`UnitySystemManager`	`NucleusSystemManager`
`UnityInterestManager`	`NucleusInterestManager`
`UnitySceneManager`	`NucleusSceneManager`
`UnityPacketManager`	`NucleusPacketManager`
`UnityMessageManager`	`NucleusMessageManager`

Two exceptions have no Unity-side component of their own: `RpcManager` and `ViolationManager`. Reach either one through `UnityCoreManager.CoreManager` (the component that owns the session's `CoreManager`) or through `NucleusUnity.BoundCoreManager`, then take the field as in C#.

The tenth Unity component `EnsureManagers` adds, `UnityPhysicsManager`, is the odd one out in the other direction: it fronts no core manager at all. It's an inspector-side seat for physics execution settings that a physics coordinator reads directly, not a wrapper around one of the eleven fields above.

No engine-wide settings object

There is no single settings type for a Nucleus session. Every knob - tick rate, a transport's connection settings, interest rules, scene load behavior - is a field on the manager that owns that concern. To change something, find its manager first.

Managers are instance-scoped

None of the eleven (or thirteen, with Pro) managers are static. Each belongs to one `CoreManager`, and a `CoreManager` is just an object - a process can construct more than one, each with its own independent set of managers. This is what lets a single process host a server and a client in the same run, or run several test peers side by side, without their state colliding.

The network loop steps

Eleven steps run in a fixed order

Nucleus never lets game code touch networked state at an arbitrary point in a frame. Every frame walks the loop's named steps in one fixed order, and a frame that runs a tick walks all eleven:

`EarlyVariableUpdate`, `EarlyStateUpdate`, `LateStateUpdate`, `Reconcile`, `EarlyFixedUpdate`, `LateFixedUpdate`, `VariableUpdate`, `EarlySerialize`, `LateSerialize`, `TickAdvance`, `LateVariableUpdate`.

Each is one flag of the `NetworkLoopSteps` enum. A callback for a step only ever receives that one flag; combinations are for declaring interest across several steps, never for a single dispatch.

The steps group by how often they run:

- **Variable steps run every frame.** `EarlyVariableUpdate`, `VariableUpdate` and `LateVariableUpdate` run on every frame the platform produces, whatever the frame rate is doing.
- **Tick steps run once per tick, on the frame that ticks.** `EarlyStateUpdate`, `LateStateUpdate`, `Reconcile`, `EarlySerialize`, `LateSerialize` and `TickAdvance` run at the engine's fixed tick rate. They are where replicated state is applied, reconciled and written, and where the tick advances.
- **Fixed steps run once per elapsed tick interval.** `EarlyFixedUpdate` and `LateFixedUpdate` run on the frame that ticks, once for each tick interval that has passed, so after a long frame they can run twice on one frame. A frame still runs at most one tick.

A frame that does not reach the next tick runs only its variable steps. Do not assume a tick step or a fixed step fires on every frame.

Several steps run framework work around your callback

Several steps carry framework work woven in alongside your callbacks, and whether that work runs before or after your callback matters:

- **`EarlyVariableUpdate` receives packets before your callback runs.** After your callback, received messages are dispatched to their handlers. If you need to react to something a message just delivered, do it downstream of this step, not inside it.
- **`LateStateUpdate` applies received state before your callback runs.** First `PresentedTick` catches up to `Tick`, then received state is applied, then received RPCs are dispatched, so a callback on this step sees the state and calls that arrived this tick already applied.
- **`EarlyFixedUpdate` advances a catching-up object before your callback runs.** In Pro, a newly spawned object that is still catching up advances one step first, so its step for the tick has already happened before anything simulates against it.
- **`LateSerialize` serializes changed state before your callback runs.** After your callback, the frame's messages, RPCs and kicks are sent. A write made inside this step's own callback is too late for this tick's state, but a message or call sent from it still leaves this frame.
- **`TickAdvance` ends the tick after your callback runs.** Systems whose stop was pending this tick finish stopping, clients that did not finish authenticating in time are dropped, `Tick` increments, and then systems whose start was deferred go live. It is the last step of a tick, so `LateVariableUpdate`, the frames between ticks and every message handler already read the tick their writes will ride.

The remaining steps (`EarlyStateUpdate`, `Reconcile`, `LateFixedUpdate`, `VariableUpdate`,

`EarlySerialize` and `LateVariableUpdate`) carry no framework work of their own; your callback is the only thing that runs on them.

Messages, RPCs and kicks are sent on every frame, not only on frames that tick. On a frame that does not tick, the engine still sends them at `LateSerialize`'s place in the order; that send is internal and raises no callback. A message, call or kick queued from a `TickAdvance` or `LateVariableUpdate` callback leaves on the next frame.

Where a step sits decides what your code may safely do

- **Read replicated state after it has been applied**, not before. `LateStateUpdate` and later is where incoming values are current; a read during `EarlyStateUpdate` or earlier can still see last tick's value.
- **Write before it is serialized.** `EarlySerialize` is the step for writing values you want sent this tick. `LateSerialize` serializes whatever changed before its callbacks run, so a write made in `LateSerialize` or later rides the next tick.
- **Do not expect a tick step every frame.** The tick steps and the fixed steps run at the tick rate, not the frame rate. Code that must run every frame regardless belongs on `VariableUpdate` or one of the other variable steps, and anything it moves should move by `StepDelta.NormalizedDelta` rather than the raw frame delta, so each tick sees an even distance.

Message and RPC dispatch

Two dispatch points fall directly out of this ordering: typed messages resolve on `EarlyVariableUpdate`, and system RPCs resolve on `LateStateUpdate`. The difference, and why each lands where it does, is covered in When a Message or Call Reaches Its Handler rather than repeated here.

Hooking into a step

In plain C#, implement `INetworkLoopStepCallback` and register it with the `NetworkLoopManager`:

```
public class ScoreTracker : INetworkLoopStepCallback
{
    public NetworkLoopSteps GetNetworkLoopSteps() =>
        NetworkLoopSteps.LateStateUpdate | NetworkLoopSteps.EarlySerialize;

    public void OnNetworkLoopStep(NetworkLoopSteps networkLoopStep, StepDelta stepDelta)
    {
        if (networkLoopStep == NetworkLoopSteps.LateStateUpdate)
        {
            // Read applied state here.
        }
    }
}

coreManager.NetworkLoopManager.RegisterNetworkLoopStepCallbacks(scoreTracker);
```

`GetNetworkLoopSteps` can combine any number of flags; `OnNetworkLoopStep` is called once per step it declared, each time with just that one flag. Unregister with `UnregisterNetworkLoopStepCallbacks` when the object is torn down.

In Unity, a `NucleusBehaviour` exposes the same eleven steps as per-step virtuals instead: `OnEarlyVariableUpdate`, `OnEarlyStateUpdate`, `OnLateStateUpdate`, `OnReconcile`, `OnEarlyFixedUpdate`, `OnLateFixedUpdate`, `OnVariableUpdate`, `OnEarlySerialize`, `OnLateSerialize`, `OnTickAdvance`, `OnLateVariableUpdate`. Override the ones you need; the base class only registers for the steps your concrete type actually overrides. See Tick rate and the loop in Unity

for the component side of this.

Testing a step in a hot path

`NetworkLoopStepsExtensions.FastContains` checks one `NetworkLoopSteps` value for another without allocating, useful when a hot path needs to branch on the current step:

```
if (networkLoopStep.FastContains(NetworkLoopSteps.EarlyFixedUpdate))
{
    // ...
}
```

The Nucleus Core Manager component

Driving the core API directly? See Creating a CoreManager.

`UnityCoreManager` is the one component a scene needs to bring up a Nucleus session. Drop it on a root GameObject and its `Awake` builds the `CoreManager`, wires the Unity integration to it, and fills in every other manager component the session needs.

Adding it to a scene

Add `UnityCoreManager` to a single GameObject. Its `Awake` constructs the session's `CoreManager` and exposes it as the `CoreManager` property. Nothing else in the scene has to wire anything by hand.

What Awake builds

`EnsureManagers()` adds any manager component missing from the GameObject, so the inspector always shows the full set:

- `UnityNetworkLoopManager`
- `UnityTransportManager`
- `UnityServerManager`
- `UnityClientManager`
- `UnitySystemManager`
- `UnityInterestManager`
- `UnityPacketManager`
- `UnityMessageManager`
- `UnitySceneManager`
- `UnityPhysicsManager`

`EnsureManagers` only adds to the `UnityCoreManager`'s own GameObject, but the notify pass afterward uses `GetComponentsInChildren<UnityManager>()`, so a manager authored on a child GameObject is still found and bound. `EnsureManagers` checks the children too, inactive ones included, before adding a default, so a manager you placed on a child is used as it is and never gains a default twin on the root. Each one is notified through `ManagersInstantiated`.

Execution order

`UnityCoreManager` carries `[DefaultExecutionOrder(-10000)]` and `[DisallowMultipleComponent]`. The execution order runs it before every other script in the scene, so a context-aware rent made from another script's `Awake` always finds a ready `CoreManager` instead of racing its construction. `DisallowMultipleComponent` keeps a GameObject from ending up with two, which would mean two `CoreManager` instances competing for the same manager components.

Order inside Awake

Before constructing anything, `Awake` reads `UnityNetworkLoopManager.TickRate` from a `UnityNetworkLoopManager` found with `GetComponentInChildren`. This has to happen first because `CoreManager` bakes every rate-derived value into itself at construction and exposes no setter afterward; the tick rate can only be supplied at the point of construction. If no `UnityNetworkLoopManager` exists yet in the scene, the tick rate falls back to `NetworkLoopManager.DefaultTickRate`.

`Awake` also adds a `UnityNetworkLoopStepProvider` component before constructing the `CoreManager`, and hands it in as the step provider. This is deliberate: the core installs its own background-timer provider when given none, and that provider would start driving the loop from the thread pool immediately, racing every loop callback the rest of `Awake` still has to register. Supplying the Unity step provider up front means the loop only ever advances from Unity's own update, and stays inert until `CoreManager` starts it.

With the tick rate and step provider in hand, `Awake`:

- Constructs `CoreManager` = `new(tickRate, unityNetworkLoopStepProvider)`.
- Calls `NucleusUnity.Initialize(CoreManager)`.
- Calls `EnsureManagers()`.
- Calls `ManagersInstantiated` on every `UnityManager` found via `GetComponentsInChildren<UnityManager>()`.

Reaching the session from elsewhere

`NucleusUnity.Initialize` binds the integration to the `CoreManager` that was just built. A script that is not on the `UnityCoreManager`'s own GameObject reaches the session through `NucleusUnity.BoundCoreManager`, a public property set by `Initialize`.

Teardown

`OnDestroy` unwinds `Awake` in reverse: it calls `NucleusUnity.Deinitialize(CoreManager)` first, then `CoreManager.Deinitialize()`. Deinitializing releases the integration's bindings before the manager graph underneath them is torn down. Calling `CoreManager.Deinitialize()` a second time is safe; it is idempotent, so a `CoreManager` that another teardown path already deinitialized simply absorbs the call.

Manager components reference

UnityManager, the base

Every manager component on the manager GameObject derives from `UnityManager`. It gives each component two things:

- `UnityCoreManager` — the `UnityCoreManager` component that owns the session's `CoreManager`.
- `ManagersInstantiated(UnityCoreManager)` — a virtual method called once, after the `CoreManager` and every one of its managers exist. Override it, call `base.ManagersInstantiated(unityCoreManager)` first, then bind your core manager reference and push any inspector fields into it.

Binding happens in this callback rather than in `Awake` or `OnEnable` because Unity gives no ordering guarantee between components on the same GameObject. `ManagersInstantiated` runs only after the whole manager graph is built, so no component can read a core manager that hasn't been constructed yet regardless of script execution order.

Components that front a core manager

Each of these exposes the `Nucleus*` core manager it drives as a public property, set inside its own `ManagersInstantiated` override:

Component	Core manager property
`UnityNetworkLoopManager`	`NucleusNetworkLoopManager` (`Nucleus.Managers.NetworkLoop.NetworkLoopManager`)
`UnityTransportManager`	`NucleusTransportManager` (`Nucleus.Managers.Transports.TransportManager`)
`UnityServerManager`	`NucleusServerManager` (`Nucleus.Managers.Server.ServerManager`)
`UnityClientManager`	`NucleusClientManager` (`Nucleus.Managers.Client.ClientManager`)
`UnitySystemManager`	`NucleusSystemManager` (`Nucleus.Managers.Systems.SystemManager`)
`UnityInterestManager`	`NucleusInterestManager` (`Nucleus.Managers.Interest.InterestManager`)
`UnitySceneManager`	`NucleusSceneManager` (`Nucleus.Managers.Scenes.SceneManager`)

UnityPhysicsManager

`UnityPhysicsManager` does not front a core manager the way the components above do — there is no core `PhysicsManager` and no `Nucleus*` property on it. It is the inspector seat for process-global physics settings, and feeds `PhysicsSimulationCoordinator` directly: `ExecutionMode` (a `PhysicsExecutionMode`) and a toggle that matches Unity's fixed delta time to the network tick rate. See the physics pages for what `ExecutionMode` controls and how the fixed-delta-time match works.

Components with no inspector fields

`UnityServerManager`, `UnityClientManager`, `UnityPacketManager`, and `UnityMessageManager` carry no serialized fields. Each still exists to bind its core manager — `ServerManager`, `ClientManager`, `PacketManager`, `MessageManager` — into `UnityCoreManager`'s manager graph and expose it as a `Nucleus*` property, so the manager GameObject is a complete, discoverable front for every core manager even where there's nothing yet to configure from the Inspector.

Tuning covered elsewhere

Some components own inspector fields that are documented on their own pages rather than repeated here:

- **UnitySystemManager** — redundancy, state interpolation, partial state apply, host migration, and controller retention settings.
- **UnityInterestManager** — evaluation cadence, host interest, and interest condition settings.
- **UnitySceneManager** — join placement, automatic scene requests, and load timeout settings.

Reaching a manager from a script

Three ways to get from your own script to a core manager, depending on what reference you already hold:

- **The component's `Nucleus` property**, when you have a reference to the manager component, e.g.* unitySystemManager.NucleusSystemManager`.
- **`UnityCoreManager.CoreManager`** , when you have a reference to the `UnityCoreManager` but not the specific component, e.g. `unityCoreManager.CoreManager.InterestManager`.
- **`NucleusUnity.BoundCoreManager`** , when you have no reference at all. It's the `CoreManager` the Unity integration bound at startup.

Runtime changes don't travel

Every inspector field covered here is pushed into its core manager once, inside `ManagersInstantiated`, when the manager graph is built. Editing a field in the Inspector after startup changes the serialized value on the component but does not push it into the core manager again — the core manager already has its own copy. To change a setting at runtime, set it directly on the `Nucleus*` property instead.

Play mode, domain reload and session lifetime

Driving the core API directly? See Shutting a session down.

Static state outlives a play session

`CoreManager` keeps a static registry of every live manager in the process (`LiveCoreManagers`, behind `CoreManager.Instance`), and the Unity integration keeps its own static session state alongside it (`NucleusUnity.BoundCoreManager`, `NucleusUnity.SpawnHandler`, the prefab pool, the network prefab registry). None of that is scoped to a play session. Stopping play mode does not clear static fields on its own, and with Unity's domain reload disabled it especially does not: the same AppDomain, and the same static fields, carry straight into the next Play button press. Without an explicit reset, a second session inherits the first one's `CoreManager`, a `SpawnHandler` bound to GameObjects Unity already destroyed, and a prefab registry that thinks its content is already loaded.

`NucleusUnity` closes that gap with five resets that run at the head of every session, before any of your code executes.

The five session-start resets

Three run as `[RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.SubsystemRegistration)]`, the earliest hook Unity offers, ahead of every `Awake`:

- **`CoreManager.ResetRegistry()`** clears the static `LiveCoreManagers` list and nulls `CoreManager.Instance`. It only drops references, it does not call `Deinitialize()` on anything still in the list — a manager left over from an ended session may already be holding GameObjects Unity destroyed, so nothing in the registry is safe to call into. The orderly unwind is `Deinitialize()`, called while the session is still live (see below); this reset is the backstop for when that never ran.
- **`NetworkPrefabRegistry.Clear()`** clears the static prefab registry. Without it, the next session's `NetworkPrefabCollection` finds every bundle already marked registered by a collection that Unity destroyed without ever unregistering, and loads nothing.
- **Session binding slots**: `NucleusUnity.SpawnHandler` and `NucleusUnity.BoundCoreManager` are set to `null`, the prefab pool is reset to a fresh `DefaultNetworkPrefabPool()`, and the internal host-visibility binder list is cleared. This is the same backstop reasoning as above, applied to the integration's own static fields rather than the engine's registry: a session that stopped without its `Deinitialize` call would otherwise leave the next session pointing at a dead `CoreManager` and a pool full of destroyed instances.

Two more run at `RuntimeInitializeLoadType.BeforeSceneLoad`, inside `RegisterServices`, just after the SubsystemRegistration hooks and still ahead of your own bootstrap:

- **The logger service** is (re)installed with `LoggingService.UseLogger(new Logging.Logger())`.
- **The application-state service** is (re)installed with `ApplicationStateService.UseApplicationState(new Environment.ApplicationState())`.

Both are re-registered every session rather than assumed to already be set, for the same static-state reason as everything else here.

Fast enter play mode and IsApplicationQuitting

With domain reload disabled, none of the engine's static state is rebuilt by Unity itself — it is entirely up to these `RuntimeInitializeOnLoadMethod` hooks, which Unity still re-runs on every play session regardless of the domain reload setting. That's what makes the five resets load-bearing rather than defensive: without them, fast enter play mode is the case where the previous session's `CoreManager`, `SpawnHandler` and prefab registry would otherwise still be sitting there when the next one starts.

`RegisterServices` also resets `NucleusUnity.IsApplicationQuitting` to `false` and re-subscribes `OnApplicationQuitting` to `Application.quitting`. The flag latches to `true` when the application begins tearing down, and a marker destroyed during that teardown uses it to know its transport is already going down with it rather than treating the destroy as a normal despawn. Resetting it per session (rather than leaving it `true` from the previous run) is what keeps a second play session from starting in an already-quitting state.

Teardown on stop

`UnityCoreManager` is the component that owns the session's `CoreManager`. Its `OnDestroy` runs the unwind:

```
private void OnDestroy()
{
    if (CoreManager is null)
        return;

    NucleusUnity.Deinitialize(CoreManager);
    CoreManager.Deinitialize();
}
```

The `CoreManager is null` check covers an `Awake` that threw before `CoreManager = new(...)` ran — there is nothing to unwind, so the method does nothing. `NucleusUnity.Deinitialize` runs first, releasing the `SpawnHandler` binding while the manager graph is still whole; `CoreManager.Deinitialize()` then releases the managers themselves, stops the network loop, and shuts down transports.

`CoreManager.Deinitialize()` is idempotent: `CoreManager.IsDeinitialized` is set at the start of the call and checked on entry, so a second call is a no-op. That flag is public specifically so a second teardown path — a destroy callback that fires after an explicit `Deinitialize()` already ran — can tell an already-torn-down manager from a live one without keeping its own tracking.

Symptoms of a session that leaked into the next one

A handler fires twice, or answers for an object that no longer exists. Something is still subscribed from the previous session. Check whether the `CoreManager` that owned it actually reached `OnDestroy` — a `DisallowMultipleComponent` conflict, a disabled GameObject, or a domain-reload-disabled run that skipped teardown entirely (stopped from the editor toolbar mid-frame, or killed) will all skip `UnityCoreManager.OnDestroy`, leaving that session's subscriptions live into the next one.

A manager keeps responding after play has stopped, or the next session's `CoreManager.Instance` isn't the one you expect. The `SubsystemRegistration` resets run before every session, but only clear static state — they do not call into a manager that was never torn down. If `Deinitialize()` never ran on the previous `CoreManager`, its manager objects still exist and still hold whatever GameObjects, transports, or callbacks they had, even though `ResetRegistry()` has already dropped them from the registry and `Instance`. Look for a missing or thrown `Awake`/`OnDestroy` on `UnityCoreManager` in the previous session's log before assuming the reset itself is at fault.

Content that loaded in the first session doesn't load in the second. Usually

`NetworkPrefabRegistry.Clear()` ran (it always does, at `SubsystemRegistration`) but the session's `NetworkPrefabCollection` never re-registered afterward — check that the collection load actually runs on the new session rather than assuming it was skipped by the reset.

A torn-down CoreManager is not reusable

Once `Deinitialize()` has run, `IsDeinitialized` stays `true` for the life of that object; there is no re-initialize path. A `UnityCoreManager` whose `Awake` wants to start a new session builds a new `CoreManager` — it does not, and cannot, revive the one `OnDestroy` just tore down.

Reacting to session events in Unity

> **Driving the core API directly?** See Session events.

A scene script usually needs to know when a client authenticates, when one drops, or when the local client's own authentication comes back. Those moments arrive as events on `ServerManager` and `ClientManager`, reached through the `UnityServerManager` and `UnityClientManager` components on the CoreManager's GameObject.

Subscribing at the right time

`UnityCoreManager` builds the CoreManager and then calls `ManagersInstantiated(UnityCoreManager)` on every `UnityManager` on its GameObject, so that's the first point the managers are guaranteed to exist. A component driving one of those managers overrides it directly:

```
public class SessionLogger : UnityManager
{
    public override void ManagersInstantiated(UnityCoreManager unityCoreManager)
    {
        base.ManagersInstantiated(unityCoreManager);

        // Managers exist now; safe to bind.
    }
}
```

An ordinary scene script that isn't a `UnityManager` doesn't get that callback. `UnityCoreManager` carries `[DefaultExecutionOrder(-10000)]` specifically so its `Awake` runs before an ordinary script's own `Awake`; subscribing from `Start` instead is still the simpler default to reach for, since it needs no dependence on execution order at all.

```
public class LobbyUI : MonoBehaviour
{
    private void Start()
    {
        // Bind here, never in Awake.
    }
}
```

The four events

Reached through the manager components once bound:

- `NucleusServerManager.ClientAuthenticated` — a `ServerManager.ClientAuthenticatedHandler(Connection connection)`, raised after the client is marked authenticated.
- `NucleusServerManager.ClientDisconnecting` — a `ServerManager.ClientDisconnectingHandler(Connection connection)`, raised while the objects the departing client controlled are still readable through it.
- `NucleusServerManager.ClientAuthenticationDenied`: a `ServerManager.AuthenticationDeniedHandler(Connection connection)`, raised for a client refused or timed out, before it disconnects.
- `NucleusClientManager.LocalClientAuthenticated` — a

`ClientManager.LocalClientAuthenticatedHandler(Connection localClientConnection)`, for the local client's own authentication.

```
public class SessionLogger : UnityManager
{
    private UnityServerManager _unityServerManager;

    public override void ManagersInstantiated(UnityCoreManager unityCoreManager)
    {
        base.ManagersInstantiated(unityCoreManager);

        _unityServerManager = GetComponent<UnityServerManager>();
        _unityServerManager.NucleusServerManager.ClientAuthenticated += OnClientAuthentica
ted;
        _unityServerManager.NucleusServerManager.ClientDisconnecting += OnClientDisconnect
ing;
        _unityServerManager.NucleusServerManager.ClientAuthenticationDenied += OnClientAut
henticationDenied;
    }

    private void OnClientAuthenticated(Connection connection) { }
    private void OnClientDisconnecting(Connection connection) { }
    private void OnClientAuthenticationDenied(Connection connection) { }

    private void OnDestroy()
    {
        if (_unityServerManager == null)
            return;

        _unityServerManager.NucleusServerManager.ClientAuthenticated -= OnClientAuthentica
ted;
        _unityServerManager.NucleusServerManager.ClientDisconnecting -= OnClientDisconnect
ing;
        _unityServerManager.NucleusServerManager.ClientAuthenticationDenied -= OnClientAut
henticationDenied;
    }
}
```

Unsubscribe in OnDestroy

Every subscription made here needs a matching removal in `OnDestroy`. This matters more in the editor than it sounds: fast enter play mode skips the domain reload, so a static handler left subscribed from a previous play session survives into the next one, still firing against a `Connection` from a session that's already gone. A handler on an instance method is only as safe as its `OnDestroy` unsubscribe; a `static` handler that never unsubscribes is the leak fast enter play mode makes visible first.

Resolving the managers without a serialized reference

A script that isn't sitting next to `UnityServerManager` in the Inspector can still reach the session through `NucleusUnity.BoundCoreManager`, which is the CoreManager the integration is bound to:

```
CoreManager coreManager = NucleusUnity.BoundCoreManager;
if (coreManager != null)
    coreManager.ServerManager.ClientAuthenticated += OnClientAuthenticated;
```

During shutdown, check `NucleusUnity.IsApplicationQuitting` before touching anything session-related — it's set once `Application.quitting` fires and re-armed per session, so it reliably tells

a teardown path whether the transport is already coming down around it.

Scripts on a networked object

A `NucleusBehaviour` attached to a networked object doesn't need any of the above — it has its own role hooks, `OnServerStarted`, `OnClientStarted` and `OnControllerChanged`, documented alongside systems rather than here.

Starting and stopping a session in Unity

Driving the core API directly? See Server, client and host roles.

The Automatic Start Mode field

`UnityTransportManager` has an **Automatic Start Mode** field backed by `NetworkStartMode`:

- `None` (the default) - does nothing on `Start`.
- `Server` - starts every transport as a server.
- `Client` - starts every transport as a client, connecting to its configured remote host.
- `Host` - starts every transport as both a server and a client in one process.

Set it in the Inspector and press Play. It is the fastest way to get two editors talking: build one editor's `UnityTransportManager` with `Server` (or `Host`) and the other with `Client`, and both come up connected without a line of code.

Starting from code

Call the async methods on `UnityTransportManager` directly instead:

```
await unityTransportManager.StartServerAsync();
await unityTransportManager.StartClientAsync();
await unityTransportManager.StartHostAsync();
```

Each of these calls `EnsureAddedAsync()` first, so the transports are added before they connect. Call `EnsureAddedAsync()` on its own when you need the transports added to the core manager without starting anything yet:

```
await unityTransportManager.EnsureAddedAsync();
```

`EnsureAddedAsync()` only adds once - a later call returns immediately if transports are already present.

Shutting a session down

```
unityTransportManager.ShutdownAll();
```

`ShutdownAll()` removes and shuts down every transport the manager added. It leaves the `CoreManager` itself standing; only `CoreManager`'s own teardown (driven by `UnityCoreManager.OnDestroy`) tears that down.

`UnityTransportManager.OnDestroy` calls `ShutdownAll()` itself, unless the `CoreManager` is already deinitialized (checked via `CoreManager.IsDeinitialized`). That means an ordinary scene unload does not leave sockets open behind it - you do not have to call `ShutdownAll()` manually before a scene change.

Wiring a menu button

`UnityCoreManager.Awake` runs at execution order `-10000` and binds every `UnityManager`, `UnityTransportManager` included, before any other script's `Awake` runs. An ordinary menu-button script can call `StartServerAsync()`, `StartClientAsync()`, `StartHostAsync()`, or `ShutdownAll()` from its own `Awake` and find `NucleusTransportManager` already set. Only a script whose own execution order sits at or below `-10000` needs to wait until `Start` instead.

A minimal button hookup:

```
public UnityTransportManager TransportManager;

public async void OnHostButtonPressed()
{
    await TransportManager.StartHostAsync();
}

public void OnStopButtonPressed()
{
    TransportManager.ShutdownAll();
}
```

How the manager finds its transports

`EnsureAddedAsync()` collects `NetworkTransport` components with `GetComponentsInChildren<NetworkTransport>()` - on the `UnityTransportManager`'s own GameObject and its children - and adds each to the core manager in the order they are found. When none are present, it adds a `SynapseTransport` component instead, so the manager always has a usable transport. Configure the transport's own settings on that component; they are not repeated here.

Reading the result

`NucleusTransportManager` (the core `TransportManager` the `UnityTransportManager` drives) exposes three booleans:

- `IsServerStarted` - true when a server connection is established on any added transport.
- `IsClientStarted` - true when a client connection is established **and authenticated**. A socket-level connect is not enough: until the local client finishes authenticating, `IsClientStarted` stays false.
- `IsHostStarted` - true when both of the above are true at once.

```
if (unityTransportManager.NucleusTransportManager.IsClientStarted)
{
    // The local client is connected and authenticated.
}
```

Tick rate and the loop in Unity

Driving the core API directly? See Ticks, tick rate and time.

Setting the tick rate

`UnityNetworkLoopManager` has a single inspector field, **Tick Rate**. It defaults to `NetworkLoopManager.DefaultTickRate` (30) and is clamped by the engine between `NetworkLoopManager.MinimumTickRate` (5) and `NetworkLoopManager.MaximumTickRate` (128). The inspector draws it as a slider bounded to that same range, so an unsupported value cannot be authored in the first place, and shows a note that the rate is fixed once the session starts.

If a scene has no `UnityNetworkLoopManager` at all, the session falls back to `NetworkLoopManager.DefaultTickRate`.

Why the rate is fixed for the session

`UnityCoreManager` reads the tick rate off `UnityNetworkLoopManager` in its own `Awake`, before it constructs the `CoreManager`. The rate is baked into every value derived from it at that point, and `NetworkLoopManager.TickRate` exposes no setter afterward. Changing the inspector field in play mode does nothing: the running session already built its derived values from the rate it started with, and the inspector disables the slider while playing to say so.

To run at a different rate, stop play mode, change the value, and start again.

How Unity drives the loop

Rather than the engine's default background-timer provider, a Unity session is driven by loop-stepping behaviour that `UnityCoreManager` adds to the scene at runtime, alongside `UnityNetworkLoopManager` (or the `UnityCoreManager`'s own object, when no `UnityNetworkLoopManager` is present). You never add or reference this behaviour yourself.

It splits the loop across Unity's own update methods, both driven from `Time.unscaledDeltaTime`:

- **`Update`** advances the early half of the loop.
- **`LateUpdate`** advances the late half of the loop.

Because the loop runs on unscaled time, `Time.timeScale` does not touch networking. Pausing the game with `Time.timeScale = 0` keeps receiving, ticking and sending, so a paused client stays connected and keeps up with the server, and slow motion does not drift the tick away from the other peers. If a pause should freeze your networked gameplay, stop your own simulation in your callbacks (for example, skip your movement code while paused) rather than relying on the time scale.

It runs at a negative execution order, so its `Update` executes before your own scripts' `Update` for the frame. A tick's early steps, and anything they trigger, have already happened by the time your gameplay code runs.

What this means for your code

Because the loop is driven from Unity's own `Update`/`LateUpdate`, every network loop callback in a Unity session runs on Unity's main thread. You can read and write Unity objects (transforms, components, `GameObjects`) directly from a loop step callback. This is unlike the engine's default, non-Unity provider, which steps the loop from a background timer and does not give you the main thread for free.

Reading the clock from a script

`UnityNetworkLoopManager.NucleusNetworkLoopManager` is the underlying `NetworkLoopManager` for the session. From it:

- **`Tick`** — the current local tick.
- **`TickRate`** — the ticks-per-second the session is running at, fixed for the session.
- **`SubtickPercentage`** — how far the loop is into the current tick, as a 0–1 fraction.

The Unity provider updates it every frame, through the same loop driver the default .NET provider uses, so `SubtickPercentage` is meaningful in a Unity session. It's what render smoothing between ticks rides on.

Move objects every frame by the normalized delta, not Time.deltaTime

If your authority moves an object every frame, move it in `OnVariableUpdate` by `stepDelta.NormalizedDelta` rather than in Unity's `Update` by `Time.deltaTime`:

```
protected override void OnVariableUpdate(StepDelta stepDelta)
{
    if (!IsController(ControllerType.AnyController))
        return;

    float seconds = (float)(stepDelta.NormalizedDelta / 1000.0);
    transform.position += _velocity * seconds;
}
```

Frames never line up with ticks, so an object moved by `Time.deltaTime` covers a different distance in every tick, and that unevenness costs bandwidth to replicate. `NormalizedDelta` is the frame's delta adjusted so the frames between two ticks add up to exactly one tick interval. The object still moves every frame at the same speed, but every tick sees the same distance moved.

The difference is large. In the Belted Asteroid benchmark, moving with `Time.deltaTime` cost 1.52 bytes per object per send, and moving with `NormalizedDelta` cost 0.44, against 0.41 for moving once per tick. In the roaming character benchmark the same three figures were 3.22, 0.52 and 0.48.

`NormalizedDelta` is in milliseconds, like the other `StepDelta` fields. See Ticks and time for exactly how it is worked out.

See also

- The network loop steps — what actually runs at each step, and in what order.
- Driving the loop yourself — writing your own `INetworkLoopStepProvider` instead of the one Unity supplies.

Common errors and what causes them

Paste the text of a log line into this page's index to find what produced it and what to change. Every line below is quoted verbatim from the engine.

Index

- ["ran on thread [X] while a network loop step is executing on thread [Y]..."](#off-loop-writer-work) — off-loop writer work
- ["touched the loop-owned collection [...] on thread [X] while a network loop step is executing on thread [Y]..."](#off-loop-collection-access) — off-loop collection access
- ["Step [X] was turned away on thread [...] because this manager is already executing a step on thread [...]..."](#second-driver) — a second driver stepping the loop
- ["Violation [X] from Connection [...]"](#violation-lines) — a violation raised against a peer
- ["A Recovery entry from Connection [...] references unknown NetworkSystem Id [...]; that entry is skipped."](#recovery-and-retention) — recovery named a system the local world no longer has
- ["...acknowledges [an impossible tick]..." (InvalidStateAckViolation)](#recovery-and-retention) — an ack above the local tick
- "...retention window was exceeded..." (RetentionExceededViolation) — acked tick fell out of the retention window
- ["A StatePacket subpacket from Connection [...] declares [X] body bits but only [Y] remain; discarding the rest of the tick."](#serializer-and-framing-faults) — declared-body-bits mismatch
- ["State subpacket [X] from Connection [...] left the reader at [...] bits but its framed boundary was [...]..."](#serializer-and-framing-faults) — framed-boundary mismatch
- ["Connection [...] sent unrecognised PacketType [X]; the packet is discarded."](#version-skew) — version skew, not a kick

Off-loop writer work

`"[Caller<MessageType>] ran on thread [X] while a network loop step is executing on thread [Y]. This work must not run concurrently with the loop; it races the outbound WriterPool and can corrupt sent packets."`

Cause. Something serialized or accumulated a message from a thread other than the one currently running a network loop step. The outbound `WriterPool` is rented from the loop during serialization, so a concurrent caller can be handed a `Writer` that is already in use.

Fix. Move the work onto the loop. If it's resuming after an `await`, route it through the loop rather than sending inline from wherever the continuation happened to land.

This guard is compiled only in `DEBUG`. A Release build has no such check and fails silently: the race still happens, it just never gets a log line naming it. Reproduce and diagnose the fault in Debug, then confirm the fix in Release.

Off-loop collection access

`"[Caller] touched the loop-owned collection [Name] on thread [X] while a network loop step is executing on thread [Y]. This work must run on the network loop thread; touching the collection off the loop races the loop and can silently corrupt it."`

Cause. An async continuation resumed off the loop thread and then touched a collection the loop owns without locking — chiefly the system routing table or the server's open-scene table. The loop mutates and enumerates these with no lock of its own, so a foreign thread touching them races the loop's reads and writes and can tear the collection or drop an entry.

Fix. Switch back onto the loop before touching loop-owned state, rather than acting from wherever the continuation resumed.

Also `DEBUG`-only. Same consequence in Release: the guard disappears, the race doesn't.

Two threads are driving one manager

`"Step [X] was turned away on thread [...] because this manager is already executing a step on thread [...]. One loop must drive one manager, and a second driver races every collection the loop owns. A manager whose owner never supplied a step provider, leaving the default one driving the loop from the thread pool, is the usual cause. At least [N] steps have been turned away since the previous report."`

Cause. Two threads are stepping the same manager's loop. The usual case is an owner that never supplied its own `INetworkLoopStepProvider`, so the default background-timer provider is driving the loop from the thread pool at the same time as whatever else is stepping it.

Fix. Make sure exactly one driver steps the manager — supply the owner's own step provider (a Unity integration's loop provider, a test harness driving ticks by hand) rather than letting two sources call in.

This line is throttled to at most once per minute per manager, because the incident it was built for produced hundreds of thousands of identical lines in minutes. Every turned-away step is still counted even when not logged, and the reported count (`At least [N] steps have been turned away`) is what tells you the real severity — a report you see once with a count of 1 is very different from one with a count in the thousands, even though the line text looks the same either way.

Violation lines

`"Violation [TypeName] from Connection [...]"`, sometimes followed by `"...settled on a kick, but this peer is not a server; the kick is not enforced."` or `"Disconnecting Connection [...] for violation [TypeName]."`

A violation writes one of these lines only when its settled action is `Log` or `Kick`. Most types default to `Ignore` and write nothing, so register an `IViolationObserver` to see those. Look up `TypeName` in the violation reference for the payload fields it carries and what triggered it:

- `EmptyCollectionDeltaViolation` — a collection member delta declaring no operations; a compliant sender can't produce one.
- `InvalidStateAckViolation` — a Connection acknowledged a state tick above the local tick.
- `MalformedDataViolation`: reading data a client sent threw on the server; kicks in a release build, logs the exception in a debug build.
- `PredictedStateChangeViolation` — a controller sent state for a member that should only ride upstream as client state.
- `RecoveryUnconfirmedViolation` — a served recovery went unconfirmed for the full patience window and had to be re-served.
- `RetentionExceededViolation` — an acked tick fell beyond the retention window, forcing a full-world recovery.
- `RpcFloodViolation` — more remote calls arrived in one drain than the receiver admits.

- `RpcSendPermissionViolation` — a remote call was sent without permission (often a well-behaved client racing a permission revocation).
- `UnauthorizedSpawnViolation` — a client sent a full serialize for a system Id the server doesn't hold.
- `UncontrolledStateChangeViolation` — a Connection sent state for a system it doesn't control.
- `UnexpectedBundleRequestViolation` / `UnexpectedSceneRequestViolation` — a client sent a request only the server ever sends.
- `UnsolicitedBundleReportViolation` / `UnsolicitedSceneReportViolation` — a client reported a load or unload it was never asked to make.

Each violation has a default `ViolationAction` (`Ignore` or `Kick`) and a registered handler can override it per Connection. The action only governs the Connection consequence; the offending operation itself is always rejected regardless of what the action does. Logging is separate: it rides on `ViolationContext<T0>.Log`, which the raise site or the handler can set.

Recovery and retention

`"A Recovery entry from Connection [...] references unknown NetworkSystem Id [X]; that entry is skipped."`

The sender believed the peer still held a system that the local world no longer has. The entry is skipped and the rest of the recovery continues.

`InvalidStateAckViolation`, `RetentionExceededViolation`, and `RecoveryUnconfirmedViolation` (see the violation index above) round out recovery diagnosis. All three default to `Ignore` rather than a kick: a single raise of any of them is ordinary behavior for a peer on a bad link — heavy loss, a retention window that's too short for the peer's latency, or a recovery that got lost along with everything else. What separates a lossy link from an actual bug is repetition: an occasional raise is the cost of an imperfect network, while raises that keep recurring for the same Connection mean its latency genuinely exceeds the retention window, or something is wrong with how it acknowledges.

Serializer and framing faults

`"A StatePacket subpacket from Connection [...] declares [X] body bits but only [Y] remain; discarding the rest of the tick."`

`"State subpacket [Type] from Connection [...] left the reader at [...] bits but its framed boundary was [...] (declared [X] body bits, off by [Y]). The stream is misaligned; discarding the rest of the tick."`

Both mean a declared bit count didn't match what was actually read or what remained in the buffer. This is a serializer question, not a gameplay one — it points at a reader/writer pair that disagree about a type's shape, not at anything the game did wrong. The rest of the tick is discarded to avoid reading garbage as if it were the next subpacket.

Version skew

`"Connection [...] sent unrecognised PacketType [X]; the packet is discarded."`

The header was read, but nothing on this side knows how to interpret that `PacketType`. This is what version skew between peers looks like on the wire: an older or newer build sent a packet type the receiver has never heard of. The packet is silently discarded rather than treated as a violation — an unrecognised type on its own isn't proof of a hostile peer, only of a version mismatch, so it's not a kick.

Developer tools in the editor

Four tools help you see what a session is doing without writing a readout of your own. Two are components you add to an object while you chase a problem, one is an editor window, and one is a menu command that checks your prefabs before you build.

Tool	Where it lives	What it answers
`NetworkCorrectionVisualizer`	Component, added under a networked object	Where and how far did prediction correct this object?
`NetworkInterestOverlay`	Component, added under a networked object	What is this object's controller being sent?
Network Inspector	**Window > Nucleus > Network Inspector**	What role, tick, connections and objects does this session have right now?
Validate Network Prefabs	**Nucleus > Validate Network Prefabs**	Will my networked prefabs spawn correctly?

None of them change the session they look at, so they are safe to leave in a scene while you test. Take the two components off, or switch them off, before you ship: they exist to answer a question, not to run in a player's build.

Seeing prediction corrections

Add `NetworkCorrectionVisualizer` to a predicted object, either on the `NetworkSystemObject` itself or on a child of it. It finds the nearest `NetworkSystemObject` at or above it in `Awake` and watches every system linked to that object, including systems that link later. Without a `NetworkSystemObject` above it, it logs an error and disables itself.

Each time a reconcile's replay finishes on a system, the visualizer looks at the members the replay corrected. For every correction larger than its threshold it flashes: a cross where the member was shown before the correction, a cross where the replay put it, and a line between them. The flash is drawn with `Debug.DrawLine`, so it shows in the Scene view, and in the Game view while the Game view's Gizmos toggle is on.

Corrections only happen on a client that predicts the object, so that is the only peer where the visualizer ever draws. On the server, and on a client that only observes the object, it stays quiet.

Field	Default	What it does
Threshold	`0.05`	The largest correction ignored, in the member's own units (metres for a position). Only larger corrections flash.
Flash Interval	`0`	The shortest time between two flashes, in seconds of unscaled time. Zero flashes on every correction past the threshold.

Flash Duration	`0.5`	How long each flash stays drawn, in seconds.
Color	magenta	The color of the line and both crosses.
Marker Size	`0.15`	The half-size of the cross drawn at each end, in metres.
Log Enabled	off	Also log each flashed correction, naming the member, its size and the tick.

While playing, the inspector adds two readouts: **Flashes**, the number of corrections drawn or logged since the component woke, and **Last Correction (m)**, the size of the most recent position correction. Both are also public properties (`FlashCount` and `LastCorrectionDistance`), so a script or a test can read them.

What gets drawn

Only a `NetworkMember<System.Numerics.Vector3>` is drawn as a line, because only a position has a place in the world to draw it. A position replicated in a parent's local space is drawn where it would sit unparented. A correction to any other member type (a rotation, a float, a custom struct) still counts toward **Flashes**, and with **Log Enabled** it is logged, but nothing is drawn for it.

The threshold is measured by the engine's own `NetworkMemberBase.IsCorrectedBeyond`, with the same tolerance comparer the engine uses when it records corrections. That means the number you type is compared in the member type's units, the same way the engine compares it, so a threshold that matches your prediction tolerance shows exactly the corrections the engine treats as real.

Reading what you see

- **Short lines on every tick** mean the client's prediction disagrees with the server by a small, steady amount. Look for something the client simulates differently: a value read from `Time.deltaTime` instead of the network step, a random number, or a physics setting that differs between builds.
- **Long lines now and then** usually line up with packet loss or a collision the client did not predict. Turn on **Log Enabled** to see which tick each one happened on.
- **No lines at all** while the object visibly snaps means the snap is not a prediction correction. Check smoothing and interpolation instead (see Proxies and smoothing).

Raise **Flash Interval** when corrections come so often that the lines turn into a smear.

Seeing what an object's controller is sent

Add `NetworkInterestOverlay` to a networked object, on the `NetworkSystemObject` or a child of it, and switch on **Overlay Enabled**. It is off by default, and it costs nothing until it is switched on and the object is selected, because the drawing happens in `OnDrawGizmosSelected`.

Enter play mode as the server or host and select the object. The overlay reads who controls each system on the object, and for each controlling client it draws a line from this object to every other networked object, colored by what that client's connection holds of it:

Color	Meaning
Green	Streamed: the client has the object and is sent its updates.

Yellow	Stopped: the client still has the object, but is no longer sent updates for it.
Faint red	Unspawned: the client does not have the object at all. Drawn only with **Unspawned Draw Enabled**.

An object with several systems is colored by the most the connection holds of any of them, so an object with one streamed part reads as streamed. When different clients control different systems on the selected object, each client's set of lines is lifted a little higher than the last, by **Controller Spacing**, so they stay apart. **Marker Radius** sets the size of the wire sphere drawn on each object a line reaches.

Interest is resolved only on the authority, so the overlay draws only on a server or host. Selecting the object in a client's editor draws nothing. An object nobody controls (a server-owned object) also draws nothing, because there is no client whose view to show.

To make the same check from code on the server, call `NetworkSystem.GetInterestMembership(connection)`, which returns the `InterestMembership` that system currently resolves toward a connection. Game code that needs to react when that changes should subscribe to the system's `InterestChanged` event instead of polling. Interest troubleshooting walks through what to do once you can see the problem.

The Network Inspector window

Open **Window > Nucleus > Network Inspector**. Outside play mode it only tells you to enter play mode with a `UnityCoreManager` in the scene; while playing, it shows the `CoreManager` the integration bound (`NucleusUnity.BoundCoreManager`) in three sections:

- **Session**: the role (Host, Server, Client or Stopped) and the current tick.
- **Connections**: one row per active connection, with its id, whether it is the server side or a client, its state, round trip time (a dash until the first measurement), packet loss, and how many reconciles the server has sent to it.
- **Objects**: every `NetworkSystemObject` in the open scenes, sorted by name. Each row lists the object's systems with their id, controller (the server or a client id), whether the system has started, and how many reconciles it has received. Click an object's name to select and ping it in the hierarchy; type in **Filter** to narrow the list by name.

The window reads only public state and changes nothing. It repaints about ten times a second, and gathers the object list from the scene once a second rather than on every repaint, so an object you spawn appears within a second.

The reconcile counts are each counted on one side only. A connection's reconciles-sent count is kept on the server, so it reads zero in a client's window. A system's reconcile count is kept on the client that receives the corrections, so it reads zero on the server. Run the window in both editors (see Testing with two editors) to see both halves.

Validating your networked prefabs

Run **Nucleus > Validate Network Prefabs** before a build, or whenever a spawn goes wrong in a way you cannot explain. It loads every prefab in the project, logs each problem it finds to the Console as an error against the prefab it names (click the entry to ping the prefab), and ends with a dialog counting the problems. A clean project gets a dialog saying every networked prefab is set up correctly.

It checks two kinds of problem.

Identity problems are the ones a player build refuses (see Prefab identity and the prefab collection): no

base `NetworkPrefabCollection`, two collections resolving to the same prefab bundle, a networked prefab missing from its collection, a prefab with no prefab id, two prefabs sharing one id, or a prefab stamped with a different bundle than its collection. This is the same check the build runs, so the validator and the build never disagree. Every identity problem is fixed by **Nucleus > Rebuild Network Prefab Collection**, and the dialog says so when it finds one.

Structure problems are hierarchies a spawn would mishandle:

- A `NetworkSystemObject` on a child of the prefab. Only a prefab's root marker is spawned, so remove the child's marker; components on the child belong to the root's marker.
- A `NucleusBehaviour` or `NetworkTransform` with no `NetworkSystemObject` at or above it. It is never given a system, so it never replicates. Add a marker to the prefab's root.
- A `NetworkHitboxRoot` that would record nothing. The validator reports the errors the hitbox root's own inspector shows; its warnings and notes stay in the inspector.

The validator runs only when you ask, never on save or on entering play mode, because loading every prefab gets slower as the project grows. It changes nothing in your project.

Diagnosing desync

Work the layers in order

"It looks wrong on the other peer" is four different bugs wearing one symptom. Check them in this order, because each rung rules out the ones below it:

Is the object there at all? If the receiver never spawned it, or the object vanished when it shouldn't have, this is interest: an `IInterestCondition` culled the pair (`InterestEffect.Spawn`, resolving to `InterestMembership.Unspawned`), or a despawn genuinely happened.

Is it there but frozen? The object exists on the receiver but stopped updating a while ago. This is `InterestEffect.Stop`: the pair resolved to `InterestMembership.Stopped`, which retains the receiver's object and drops it from the delta stream rather than despawning it. It looks identical to a dead connection from the outside; it isn't.

Is it there and moving, but wrong? Position, rotation, or some other member holds a value that doesn't match the sender. This is a state question: either the value never changed on the receiver, or it changed to something incorrect.

Is it there, correct, but late? The value is right eventually, just visibly behind. This is interpolation and the retention window, not a data bug.

Don't skip ahead. A value that "looks wrong" because the object was actually culled five seconds ago and you're staring at a stale render is not a serializer bug, and chasing it as one wastes time.

Never arrived vs. arrived and wrong

Once you're at rung 3, split it again before touching a serializer:

- **A value that never changes on the receiver** is an observation or access question. Is the Connection actually registered as an observer of that `NetworkSystem`? Did an interest condition stop it (rung 2, revisit it)? Does the sender even own or control the member it's trying to change? A member that never updates usually means the write never left the sender, or never reached this receiver, not that the receiver decoded it wrong.
- **A value that changes, but to the wrong thing**, is a serializer or prediction question. The receiver is getting deltas and applying them, so the pipeline is working; the value it lands on is what's wrong. That points at how the member is being written or projected (`TransmissionMode.Divine` vs `TransmissionMode.Interval`), or, for a controlled member, at prediction/reconciliation on the controlling side.

These two failure modes live in different subsystems. Don't debug a serializer because a value never showed up, and don't debug interest because a value arrived and was simply incorrect.

Read the violation stream first

Most desyncs that are actually protocol faults - not just late or stale data - raise a violation before you'd notice anything visually. `ViolationManager` is the framework-wide pipeline for this: register a `ViolationHandler<T0>` to decide the `ViolationAction` for one violation type, or an `IViolationObserver` to hear every type raised, or subscribe the matching per-type event (for example `UncontrolledStateChangeViolationDetected`) to just watch.

The types worth knowing when chasing desync:

- `UncontrolledStateChangeViolation` - a peer sent a state change for a `NetworkSystem` it

doesn't control.

- `PredictedStateChangeViolation` - a controller sent state for a predicted member, which should only ride upstream as client state.
- `EmptyCollectionDeltaViolation` - a collection delta declared no operations, which a compliant sender can't produce; usually a crafted packet or a diverged serializer pair.
- `InvalidStateAckViolation` - a peer acknowledged a tick above anything ever sent to it.
- `RetentionExceededViolation` - a peer's acknowledged tick fell beyond the retention window and had to be re-served the whole world. A single raise is ordinary heavy loss; chronic raises mean the peer's latency exceeds `SystemManager.StateRetentionMilliseconds`, or it's deliberately withholding acknowledgments.
- `RecoveryUnconfirmedViolation` - consecutive recoveries were served to a Connection without it ever confirming one.
- `UnauthorizedSpawnViolation` - a client's full serialize named a system identifier the server doesn't hold, which would have spawned an object on the server.

If a symptom in rungs 1-3 has a matching raise, that's your subsystem, directly. **If nothing is raised at all**, the fault sits in something the violation pipeline doesn't police: a legitimate interest decision, ordinary loss inside the retention window, an interpolation lag, or a prediction mismatch that's still a well-formed packet. Absence of a violation is evidence, not a dead end.

The host special case

A host still serializes its own client's packets normally on the send side - nothing is skipped there. What changes is receipt: the arriving loopback copy is discarded unread rather than deserialized, because whatever it carries either duplicates what the sending half already applied in-process, or is a stale re-apply of the same object the host itself authored.

This means a bug on the receive-and-apply path - deserialization, violation checks that only run on an incoming packet, interest resolution for an arriving state, anything downstream of "a packet was read" - never runs for a host's own client at all. If a repro only shows up between two separate processes (two real clients, or a client against a dedicated server) and never on a host testing against its own client, look at the receive path first. Host loopback simply never exercises it.

Timing faults

Some symptoms point at the tick loop, not at data. The loop runs its steps in a fixed order - network reads happen early, state is read and applied at the start of a step, serialized and sent at the end of one - and a callback only sees what's already been applied by the point it runs on. Signs you're looking at a step-ordering problem rather than a data problem:

- A value read in one callback is stale compared to the same value read in a different one on the same tick - check which steps each callback runs on, not the value itself.
- A change looks like it "doesn't take" until the following tick, or takes twice - it's being written on the wrong side of a state-apply boundary, or read before the write it depends on has happened.
- Symptoms vary with frame rate or with multiple fixed updates landing inside one variable update - that's the loop's own cadence, not the network.

Fixing these means moving the callback to the correct step relative to when state applies and when it's written, not touching serialization at all.

Narrowing to a reproduction

Once you know which rung and which of the two failure modes, shrink the repro before reading a single packet:

Cut to one object and one member. A whole scene desyncing is the same bug as one member desyncing, just noisier to look at.

Turn projection off - set the member's `TransmissionMode` to `Interval`. If the bug disappears, it's in the projection path; if it doesn't, projection was never the cause.

Raise loss deliberately (a `NetworkConditions` with a `PacketLossChance`, passed to `TransportManager.SetNetworkConditions`) to see whether the symptom is loss-shaped - arrives late, arrives out of order, never arrives - versus present on every send.

Only then read a packet. By this point you know which member, which tick range, and which of "never arrived" or "arrived wrong" you're chasing, so there's an actual question to answer instead of a blind capture.

Frequently asked questions

Short answers to the questions that come up before anyone has finished reading the rest of the docs. Each one links to the page with the full explanation.

Visibility and spawning

Why did an object stop moving instead of disappearing

Interest resolves to one of a few verdicts per connection, and "stopped" is not "unspawned." A stopped object is kept by the receiver and simply sent nothing further, which reads as frozen; an unspawned object is torn down. They are different outcomes of the same evaluation, not two names for one thing. See the interest and area-of-interest pages for what drives each verdict.

Why is nothing replicating at all

Check that the Nucleus source generator is wired into the project as an analyzer, not just referenced as an ordinary assembly. Without it no serializers are produced for your networked types, so there is nothing for the wire to send or read. See the source generator setup page.

Control

Why can I see my object but not drive it

Observing and controlling are separate questions. Interest decides whether a connection is sent an object's state at all; control is a separate assignment made by the server over who is allowed to drive it. Being registered as an observer never implies being the controller. See the control assignment page.

Timing

Why does it work in host mode and break with two processes

A host's own client and its server share one process, and packets between them still go through the transport's loopback path. The receiving half never reads that loopback copy back: it discards the arriving packet unread rather than deserializing it, so nothing on the deserialize-and-apply path runs for a host talking to itself. Testing exclusively in host mode never exercises that path at all, which is why a bug there only shows up once a second process is involved. See the diagnosing-desync page for the isolation steps.

Host-mode surprises

Why does the host see my object differently than my client does

The same loopback exemption above applies more broadly: a host's server half never goes through its own client's receive path, so anything gated behind deserialization, including any host-side reporting of what its own client would have received, has to be resolved separately rather than inferred from a packet the host never actually reads.

Bandwidth surprises

Why is my bandwidth number unchanged after I turned a feature off

The transport's byte and bit counters are running totals, accumulated since the connection started, with no per-object or per-feature attribution. Turning a feature off changes the rate at which the totals climb, not the totals themselves, so a snapshot taken once won't show it. Sample the counters over time (per second) and compare rates before and after the change, rather than reading a single cumulative value.

Editions

Why does this type not exist in my build

Free and Pro are not a runtime switch. A Pro-only source file is excluded from the compilation entirely when building Free, so a Pro-only type is a compile error in a Free project, not something that fails or falls back at runtime. If your code references a type that isn't there, you're either building Free against Pro-only API, or your project is pulling in the wrong edition's files.

Filing a bug report

What to include

State the edition: Free or Pro. The edition is a compile-time choice, not a runtime flag — a Free build is produced with `-p:NucleusEdition=Free`, which drops every `*.Pro.cs` file from the compilation, so a report that says "Free" or "Pro" is telling us which files exist in the build, not a setting that could be misread.

List the DLL set you deployed, and confirm every one of those DLLs came from the same build. A mismatched set (an old engine DLL next to a new source-generator DLL, for example) produces failures that look like engine bugs but are actually version skew. If Unity is involved, give the Unity version. If it's a plain host, give the target framework you're building against (the engine itself targets `netstandard2.1`).

Before you file

Reproduce it outside host mode first. A host runs both the server and client sides of the connection in the same process, and that dual role masks or manufactures behavior that a real client/server split does not show. If a repro only fails when hosting, it usually isn't the bug it looks like — narrow it to a genuine two-process reproduction before filing.

Reproduce it again with logging set to `LoggerLevel.Information` and capture the output. Information is verbose enough to show the sequence of events leading into the failure without needing a rebuild.

Say which violations were raised, if any. The engine reports rule breaks through `IViolation` payloads dispatched by the violation system; if your repro triggers one, name it and include its context.

Building a minimal repro

Minimal means: one networked object, one member on it, and the smallest scene (Unity) or the smallest `Main` (plain host) that reproduces the failure. Strip everything not required to trigger it — other systems, other objects, other members — before filing.

A repro that only fails in host mode is usually not the bug you think it is. Confirm it reproduces with the server and client as separate processes before treating it as real.

Attaching evidence

Include:

- The log lines, full text, not paraphrased or truncated.
- Counter readings before and after the failure, if the behavior involves a count (received/sent packets, retries, spawned objects, etc.).
- The exact tick or sequence number at which the failure occurs, if you can identify it.

Evidence in this shape is what lets a report be reproduced from the outside, rather than argued about.

Handling violations in Unity

Driving the core API directly? See Violations

No violation manager component

The Unity integration ships no manager component for violations. There is nothing to drag onto a GameObject and no inspector to configure. `ViolationManager` lives on the `CoreManager`, and you reach it from C# through whichever reference your bootstrap gives you:

```
CoreManager coreManager = GetComponent<UnityCoreManager>().CoreManager;
ViolationManager violationManager = coreManager.ViolationManager;
```

or, from anywhere that isn't the manager root itself:

```
CoreManager coreManager = NucleusUnity.BoundCoreManager;
```

`NucleusUnity.BoundCoreManager` is set once `UnityCoreManager` has bound the integration, and cleared on teardown. Registration is always a C# call against `ViolationManager` - `RegisterViolationHandler<T0>`, `RegisterViolationDetectedHandler<T0>`, or `RegisterViolationObserver`.

Registering at the right time

`UnityCoreManager` builds the `CoreManager` in its own `Awake`, which runs at `[DefaultExecutionOrder(-10000)]` so the rest of the scene sees a constructed manager. That also means a `MonoBehaviour` sitting on the same GameObject as `UnityCoreManager` cannot register from its own `Awake` - execution order puts `UnityCoreManager.Awake` first only because of that attribute, and a component without one is not guaranteed to run after it. Register from `Start`, or from any callback that is documented to run after the manager root's `Awake` (`ManagersInstantiated` on a `UnityManager`, for instance).

```
private CoreManager _coreManager;
private ViolationManager.ViolationDetectedHandler<UncontrolledStateChangeViolation> _handl
er;

private void Start()
{
    _coreManager = NucleusUnity.BoundCoreManager;
    _handler = OnUncontrolledStateChange;
    _coreManager.ViolationManager.UncontrolledStateChangeViolationDetected += _handler;
}

private void OnUncontrolledStateChange(in ViolationContext<UncontrolledStateChangeViolatio
n> violationContext)
{
    // inspect violationContext.Connection, violationContext.Violation, violationContext.A
ction
}
```

Unregistering in OnDestroy

Unregister every handler and subscriber in `OnDestroy`, symmetrically with where you registered:

```
private void OnDestroy()
{
    if (_coreManager is not null)
        _coreManager.ViolationManager.UncontrolledStateChangeViolationDetected -= _handler
;
}
```

A registration is not scoped to the scene or to the GameObject that made it - it lives for the lifetime of the `ViolationManager`, which outlives a scene load and, with fast enter-play-mode (no domain reload), can outlive a whole play session. A handler you never unregister keeps a reference to a destroyed `MonoBehaviour` alive in the manager's dictionary, and on the next raise the manager invokes a delegate whose target no longer exists. That is a leak, not a convenience - unregister it every time, including on an object a pool recycles rather than destroys.

If a handler, observer or subscriber does throw (a `MissingReferenceException` from a destroyed target is the usual one), the manager contains it: a throwing handler falls back to the violation's default action, a throwing observer does not stop the ones after it, and a throwing subscriber does not stop enforcement. The exception is logged to the console as an error, so the bug is visible, but the offending client is still dealt with.

Only a server enforces a kick

`ViolationManager` settles an action for each violation and then enforces it. A `ViolationAction.Kick` is only ever carried out when the local peer has a server started; on a client-only peer the manager logs that the kick settled but was not enforced, and the connection is left alone. A deciding handler registered on a client-only build therefore never actually kicks anyone - it can still log or observe, but nothing it returns changes what happens to the offending connection. Put the handler that decides `Kick` on the build that runs the server; a client-only handler that only observes or logs is fine anywhere.

Reading a violation from the console

With no handler registered, most violations are raised with `ViolationAction.Ignore` and write nothing to the console. The scene- and bundle-protocol faults default to `Kick`, and only `PacketTransformRejectedViolation` (Pro) falls back to `ViolationManager.DefaultAction` (`ViolationAction.Ignore`), which always logs a warning. Logging no longer rides on the action: it is carried on `violationContext.Log`, a `LogEntry` the raise site may seed and the deciding handler may change or clear. To see the silent ones, register an `IViolationObserver` or a per-type subscriber as above and log them yourself. A set `Log` writes a line of the form:

```
Violation [<ViolationTypeName>] from Connection [<connection>]: <message>
```

A `Kick` action logs a similar line before disconnecting; a `Kick` settled on a non-server peer logs that it was not enforced.

Several violation payloads carry a `SystemId` (for example `UncontrolledStateChangeViolation.SystemId`) rather than a GameObject reference. To find the GameObject behind that id, resolve the system first and then the marker:

```
if (_coreManager.SystemManager.TryGetSystemReference(violationContext.Violation.SystemId,
out NetworkSystem networkSystem)
    && networkSystem is not null
    && NetworkSystemObject.TryGetLinked(networkSystem, out NetworkSystemObject networkSyst
emObject))
{
    GameObject offendingGameObject = networkSystemObject.gameObject;
}
```

`SystemManager.TryGetSystemReference` turns the wire id into a `NetworkSystem`; `NetworkSystemObject.TryGetLinked` (or the marker's own `System` property, once you already hold the marker) is what ties that `NetworkSystem` back to the GameObject it is linked to.

Diagnostics and Troubleshooting

Pages in this category:

- Violation reference
- What the server trusts
- Common errors and what causes them
- Technical limitations
- Engine logging in the Unity console
- Handling violations in Unity
- Reading network statistics in Unity
- Developer tools in the editor
- Diagnosing desync
- Frequently asked questions
- Filing a bug report

Engine logging in the Unity console

Driving the core API directly? See Logging.

Every message the engine logs already shows up in the Unity console. There is nothing to wire up.

How it gets there

During initialization, `NucleusUnity` registers a Unity-specific logger with the engine's logging service:

```
LoggingService.UseLogger(new Logging.Logger());
```

`Nucleus.Integrations.Unity.Logging.Logger` implements `ILogger` and routes each level to the matching `UnityEngine.Debug` call: `LogInformation` to `Debug.Log`, `LogWarning` to `Debug.LogWarning`, `LogError` to `Debug.LogError`. Because these are the same calls Unity's own code uses, console lines from the engine click through to their call site like any other Unity log.

Lines carry Unity's own stack trace

The Unity `Logger` hands each message to `Debug` unchanged, and Unity attaches its own stack trace as the project's Stack Trace Logging settings allow (Project Settings > Player > Other Settings). It overrides `DisableUnconditionalDevelopmentStacktrace()` to return `true`, so the engine does not add a second trace on top.

Reading a line

A line is prefixed with `[Type::Method]`, built from the calling type and a `CallerMemberName`-based method name - not a manager or subsystem category. The stack trace is appended to the message as plain text after a colon, not attached as one of Unity's own clickable console links, so the click-through you get is Unity's default per-line behavior, not something the stack trace text itself provides.

Setting the volume

`LoggerSetting` declares three levels: `Editor`, `DevelopmentBuilds`, and `ReleaseBuilds`. The Unity logger's defaults are Information for the editor, Information for development builds, and Error for release builds.

Which of the three applies is decided by the Unity application state the integration registers before the first scene loads: the editor uses `Editor`, a development player uses `DevelopmentBuilds`, and a release player uses `ReleaseBuilds`. The level is worked out again whenever the logger or the application state is set, so the order the two are registered in does not matter, and a release build logs only errors by default. To change the volume, register your own logger whose `GetLoggerSetting` returns the levels you want (below).

Replacing the sink

If your project already logs somewhere else - a file, a remote service, a custom console - register your own `ILogger` in place of the Unity one:

```
LoggingService.UseLogger(new MyProjectLogger());
```

Do this after `NucleusUnity` has registered its logger, so your call is the one that sticks. Your implementation controls `GetLoggerSetting()` and `DisableUnconditionalDevelopmentStacktrace()` as well as the three log methods. Giving up the Unity `Logger` means giving up its click-through-to-call-site behavior; your sink is responsible for its own way

of pointing back at source, if you want one.

Reading network statistics in Unity

> **Driving the core API directly?** See Network statistics

Getting to the counters

Every counter lives on `TransportManager`, off the process's `CoreManager`. Three ways to reach it from a Unity script:

- From a component that already sits next to `UnityCoreManager`: read its `CoreManager` property.
- From any plain `MonoBehaviour`: `NucleusUnity.BoundCoreManager`, the CoreManager the integration bound at startup.
- From a script deriving from `NucleusBehaviour` or `NucleusBehaviour<TComponent0>`: the behaviour already exposes a `CoreManager` property, resolved in `Awake`.

Once you have a `CoreManager`, its `TransportManager` property holds the counters and connection lists this page covers.

A worked readout component

`TransportManager.TotalBytesSent` and `TotalBytesReceived` are running totals, not rates - sample them once a second and diff against the previous sample:

```
using Nucleus.Connections;
using Nucleus.Integrations.Unity;
using Nucleus.Managers.Core;
using Nucleus.Managers.NetworkLoop;
using Nucleus.Managers.Transports;
using UnityEngine;

public class NetworkStatsReadout : MonoBehaviour, INetworkLoopStepCallback
{
    private CoreManager _coreManager;
    private long _lastBytesSent;
    private long _lastBytesReceived;
    private long _elapsedMilliseconds;

    private void OnEnable()
    {
        _coreManager = NucleusUnity.BoundCoreManager;
        _coreManager?.NetworkLoopManager.RegisterNetworkLoopStepCallbacks(this);
    }

    private void OnDisable() => _coreManager?.NetworkLoopManager.UnregisterNetworkLoopStep
Callbacks(this);

    public NetworkLoopSteps GetNetworkLoopSteps() => NetworkLoopSteps.LateVariableUpdate;

    public void OnNetworkLoopStep(NetworkLoopSteps networkLoopStep, StepDelta stepDelta)
    {
        _elapsedMilliseconds += stepDelta.Delta;

        if (_elapsedMilliseconds < 1000)
            return;

        _elapsedMilliseconds = 0;

        TransportManager transportManager = _coreManager.TransportManager;

        long bytesSent = transportManager.TotalBytesSent;
        long bytesReceived = transportManager.TotalBytesReceived;

        long sentPerSecond = bytesSent - _lastBytesSent;
        long receivedPerSecond = bytesReceived - _lastBytesReceived;

        _lastBytesSent = bytesSent;
        _lastBytesReceived = bytesReceived;

        Debug.Log($"Up: {sentPerSecond} B/s, Down: {receivedPerSecond} B/s");

        if (transportManager.IsServerStarted)
        {
            // A server holds one live link per client.
            foreach (Connection connection in transportManager.ActiveConnections)
                Debug.Log($"Client {connection.Id} RTT: {connection.RoundTripTimeMilliseco
nds} ms, Loss: {connection.PacketLossPercentage:F1}%");
        }
        else if (transportManager.IsClientStarted && transportManager.TryGetServerConnecti
on(out Connection serverConnection))
        {
```

```
            // A client's link figures live on the connection that stands for the server.
            Debug.Log($"Server RTT: {serverConnection.RoundTripTimeMilliseconds} ms, Loss:
  {serverConnection.PacketLossPercentage:F1}%");
        }
    }
}
```

The readout is a plain `MonoBehaviour` that registers itself with the network loop while it is enabled, not a `NucleusBehaviour`: a `NucleusBehaviour` only receives loop steps while a networked system is linked to it, and a readout for the whole process has no system of its own. `PacketLossPercentage` already runs from 0 to 100, so it prints with `F1` rather than `P1`, which would multiply it by 100 again. A client reads its own link from `TryGetServerConnection` rather than `ActiveConnections`: on a client that collection holds stand-ins for the other players, which have no link of their own to measure.

Sample on a loop step, not Update

Take the reading from a network loop step (`LateVariableUpdate` above, or any other `NetworkLoopSteps` value) rather than Unity's `Update`. The network loop's own steps run serialization and deserialization at defined points in the tick; a plain `Update` callback has no guaranteed order relative to those steps and can read a counter mid-update. A loop-step callback only fires when the framework has finished the work for that step.

The diagnostics HUD is sample code, not shipped API

`Nucleus.Integrations.Unity/Demos/Common/Scripts` ships a full diagnostics HUD - `NetworkDiagnosticsHud`, `NetworkStatisticProvider`, `RollingNetworkStatisticProvider`, and five concrete providers covering role, round-trip time, bandwidth, state-packet bandwidth, and object bandwidth. It is real, working code, worth reading for patterns.

It compiles into the `Nucleus.Integrations.Unity.Demos` assembly, not the `Nucleus.Integrations.Unity` assembly your project actually links against - it is demo content that ships alongside the integration, not part of it. Copying `NetworkDiagnosticsHud` or a provider into your own project is forking that sample, not consuming a shipped API - you own the copy and its upkeep from that point on.

Totals are process-wide

`TotalBytesSent` and `TotalBytesReceived` count every byte the process sends or receives, across every connection and every networked object. There is no per-object bandwidth counter. To estimate what one object costs, disable it (or the behaviour driving its state) and compare the rate before and after - the difference is the object's share, not something the engine reports directly. The shipped `NetworkObjectBandwidthStatisticProvider` uses this same divide-by-count approach, using `SystemManager.StartedSystemCount` as the denominator.

Other routes worth knowing

- `TransportManager.ActiveConnections` lists every connected client on a server, for a per-connection round-trip time and loss readout like the one above. On a client it lists stand-ins for the other players, which carry no link figures.
- `TransportManager.GetLocalConnections(Invoker.Server)` (or `Invoker.Client`) returns this peer's own connection for that role on each added transport. On a client, the `Invoker.Server` entry is the one holding the server link's round-trip time and loss, the same connection `TryGetServerConnection` returns.

- `SystemManager.StartedSystemCount` is the started-object count the shipped per-object provider divides bandwidth by; use it for the same kind of estimate in your own readout.

Reading network statistics in Unity

Driving the core API directly? See Network statistics

Getting to the counters

Every counter lives on `TransportManager`, off the process's `CoreManager`. Three ways to reach it from a Unity script:

- From a component that already sits next to `UnityCoreManager`: read its `CoreManager` property.
- From any plain `MonoBehaviour`: `NucleusUnity.BoundCoreManager`, the CoreManager the integration bound at startup.
- From a script deriving from `NucleusBehaviour` or `NucleusBehaviour<TComponent0>`: the behaviour already exposes a `CoreManager` property, resolved in `Awake`.

Once you have a `CoreManager`, its `TransportManager` property holds the counters and connection lists this page covers.

A worked readout component

`TransportManager.TotalBytesSent` and `TotalBytesReceived` are running totals, not rates - sample them once a second and diff against the previous sample:

```
using Nucleus.Connections;
using Nucleus.Integrations.Unity;
using Nucleus.Managers.Core;
using Nucleus.Managers.NetworkLoop;
using Nucleus.Managers.Transports;
using UnityEngine;

public class NetworkStatsReadout : MonoBehaviour, INetworkLoopStepCallback
{
    private CoreManager _coreManager;
    private long _lastBytesSent;
    private long _lastBytesReceived;
    private long _elapsedMilliseconds;

    private void OnEnable()
    {
        _coreManager = NucleusUnity.BoundCoreManager;
        _coreManager?.NetworkLoopManager.RegisterNetworkLoopStepCallbacks(this);
    }

    private void OnDisable() => _coreManager?.NetworkLoopManager.UnregisterNetworkLoopStep
Callbacks(this);

    public NetworkLoopSteps GetNetworkLoopSteps() => NetworkLoopSteps.LateVariableUpdate;

    public void OnNetworkLoopStep(NetworkLoopSteps networkLoopStep, StepDelta stepDelta)
    {
        _elapsedMilliseconds += stepDelta.Delta;

        if (_elapsedMilliseconds < 1000)
            return;

        _elapsedMilliseconds = 0;

        TransportManager transportManager = _coreManager.TransportManager;

        long bytesSent = transportManager.TotalBytesSent;
        long bytesReceived = transportManager.TotalBytesReceived;

        long sentPerSecond = bytesSent - _lastBytesSent;
        long receivedPerSecond = bytesReceived - _lastBytesReceived;

        _lastBytesSent = bytesSent;
        _lastBytesReceived = bytesReceived;

        Debug.Log($"Up: {sentPerSecond} B/s, Down: {receivedPerSecond} B/s");

        if (transportManager.IsServerStarted)
        {
            // A server holds one live link per client.
            foreach (Connection connection in transportManager.ActiveConnections)
                Debug.Log($"Client {connection.Id} RTT: {connection.RoundTripTimeMilliseco
nds} ms, Loss: {connection.PacketLossPercentage:F1}%");
        }
        else if (transportManager.IsClientStarted && transportManager.TryGetServerConnecti
on(out Connection serverConnection))
        {
```

```
            // A client's link figures live on the connection that stands for the server.
            Debug.Log($"Server RTT: {serverConnection.RoundTripTimeMilliseconds} ms, Loss:
  {serverConnection.PacketLossPercentage:F1}%");
        }
    }
}
```

The readout is a plain `MonoBehaviour` that registers itself with the network loop while it is enabled, not a `NucleusBehaviour`: a `NucleusBehaviour` only receives loop steps while a networked system is linked to it, and a readout for the whole process has no system of its own. `PacketLossPercentage` already runs from 0 to 100, so it prints with `F1` rather than `P1`, which would multiply it by 100 again. A client reads its own link from `TryGetServerConnection` rather than `ActiveConnections`: on a client that collection holds stand-ins for the other players, which have no link of their own to measure.

Sample on a loop step, not Update

Take the reading from a network loop step (`LateVariableUpdate` above, or any other `NetworkLoopSteps` value) rather than Unity's `Update`. The network loop's own steps run serialization and deserialization at defined points in the tick; a plain `Update` callback has no guaranteed order relative to those steps and can read a counter mid-update. A loop-step callback only fires when the framework has finished the work for that step.

Sample content, not shipped API

`Nucleus.Integrations.Unity/Demos/Common/Scripts` ships a full diagnostics HUD - `NetworkDiagnosticsHud`, `NetworkStatisticProvider`, `RollingNetworkStatisticProvider`, and five concrete providers covering role, round-trip time, bandwidth, state-packet bandwidth, and object bandwidth. It is real, working code, worth reading for patterns.

It is sample content, not part of the `Nucleus.Integrations.Unity` assembly: the folder has no runtime asmdef of its own, so it is not something your project references. Copying `NetworkDiagnosticsHud` or a provider into your project is forking that sample, not consuming a shipped API - you own the copy and its upkeep from that point on.

Totals are process-wide

`TotalBytesSent` and `TotalBytesReceived` count every byte the process sends or receives, across every connection and every networked object. There is no per-object bandwidth counter. To estimate what one object costs, disable it (or the behaviour driving its state) and compare the rate before and after - the difference is the object's share, not something the engine reports directly. The shipped `NetworkObjectBandwidthStatisticProvider` uses this same divide-by-count approach, using `SystemManager.StartedSystemCount` as the denominator.

Other routes worth knowing

- `TransportManager.ActiveConnections` lists every connected client on a server, for a per-connection round-trip time and loss readout like the one above. On a client it lists stand-ins for the other players, which carry no link figures.
- `TransportManager.GetLocalConnections(Invoker.Server)` (or `Invoker.Client`) returns this peer's own connection for that role on each added transport. On a client, the `Invoker.Server` entry is the one holding the server link's round-trip time and loss, the same connection `TryGetServerConnection` returns.
- `SystemManager.StartedSystemCount` is the started-object count the shipped per-object provider

divides bandwidth by; use it for the same kind of estimate in your own readout.

Technical limitations

Some names exist in the code but nothing is wired to them

Some names exist in the engine but nothing wires them up yet. Treat them as documentation of intent, not as working settings.

These features were built and then deliberately removed

These were built, shipped for a time, and then deliberately torn out. If you find a doc, comment or stale reference describing one, it is not a bug you rediscovered — it is history.

- **Distance-driven send-interval bands** — a middle interest tier that paced a distant object's sends instead of stopping it outright was removed from the shipped interest types. Per-member send interval and transmission mode still exist, but neither is driven by distance.
- **Full-world rollback physics** — client-side resimulation that restored the whole physics world to a snapshot tick and re-stepped it replaying client state was removed once projected sync proved it could stay pop-free without that cost.

These features are planned but not built yet

Distinct from the permanent limits above: these are absent because the work hasn't happened, not because the design forbids them. They may ship later.

- **Network profiler window** — an editor window attributing traffic to messages, objects and members. The counters it would read already exist; the window itself does not.

A browser/WebGL transport has no register entry either way: it has neither shipped nor been recorded as planned. Server-side rewind, lag-compensated queries and hitbox history are not on this list because they are built, as the Pro feature described in How lag compensation works.

Violation reference

What a violation is

A violation is a struct implementing `IViolation`, raised through `ViolationManager` when a peer sends something the protocol does not permit. The offending operation is always rejected regardless of what happens next; the violation only decides the `ViolationAction` taken against the `Connection` that sent it: `Ignore` or `Kick`. Logging is separate from the action, carried on `ViolationContext<T0>.Log`, a `LogEntry` the raise site may seed and the deciding handler may change or clear. Register a `ViolationHandler<T0>` to override the default action per type, or subscribe a `ViolationDetectedHandler<T0>` (or `IViolationObserver`) to be told about one without changing the outcome.

There are fifteen shipped violation types. Two exist only in Pro.

Peer-behavior violations

These fire from state application, spawning, and RPC dispatch. All default to `Ignore`.

Type	Payload	Check first
`UncontrolledStateChangeViolation`	`SystemId`	Whether the sender actually holds control of that `NetworkSystem` right now; control handoffs racing a state send look identical to this from the receiving side.
`PredictedStateChangeViolation`	`SystemId`	Whether the member is really predicted; a controller never serializes a predicted member's state, so this usually means the member's transmission setup changed without the controller rebuilding.
`EmptyCollectionDeltaViolation`	`SystemId`, `MemberId`	A compliant sender cannot produce a collection delta with no operations, so first suspect a diverged serializer pair between the two builds, not the sender's intent.
`InvalidStateAckViolation`	`AckedTick`	Whether the two peers' tick counters have actually diverged (a restart, a save/load, a manual tick reset) rather than assuming a forged ack.

`RetentionExceededViolation`	`AckedTick`	The peer's latency against `SystemManager.StateRetentionMilliseconds`; a single raise is ordinary heavy loss, not abuse.
`RecoveryUnconfirmedViolation`	`UnconfirmedServeCount`	Whether the peer's connection is dropping the recovery serve itself (packet loss eating the recovery, not just the state it was fixing) before assuming it is stalling on purpose.
`UnauthorizedSpawnViolation`	`SystemId`, `SystemBundleId`, `SystemLocalId`	Whether the server just despawned that system id and the client's own recovery pass re-served it as a full before learning of the despawn; that path raises nothing, so a raise here means the id was never one the server despawned.
`RpcSendPermissionViolation`	`SystemId`	Whether permission was revoked while calls already sent were still in flight; a well-behaved client reaches this on that race alone.
`RpcFloodViolation`	`ReceivedRpcCount`, `AllowedRpcCount`	`RpcManager.MaximumInboundRpcsPerConne against what the client actually needs to send in one drain, before assuming a flood attempt.

Malformed-data violation

`MalformedDataViolation` fires on the server when reading data a client sent throws, so the data is not applied. A well-behaved client never sends data the server cannot read, so it defaults to `Kick` in a release build and logs `Sent data the server could not read.` as a warning. A debug build (`#if DEBUG`) raises it with `Ignore` and no log line, and the read site logs the exception instead, since there the usual cause is a serializer under development. It is never raised on a client, for data with no known sender, for a server Connection (a host that dialled its own address reads from one as a client), or for a host's own client; in those cases the read site logs the exception and nobody is kicked, so a host never drops its own link or its own player over a bad read.

Type	Payload	Check first
`MalformedDataViolation`	`Kind` (`State`, `FullBody`, `Recovery`, `ClientState`, `Message`, `Rpc`), `SystemId` (zero when not for a system)	Whether both builds share the same serializers and member layout for that `Kind`; a version mismatch between client and server builds reads exactly like a forged packet. Run a debug build to see the exception.

Protocol-direction violations

These fire when a client answers about a bundle or scene it was never asked about. The server drives every load and release with a reliable request; nothing in the engine sends these messages upstream, so a well-behaved client cannot produce any of the four. They default to `Kick`.

Type	Payload	Check first
`UnexpectedBundleRequestViolation`	`BundleId`	Nothing on a well-behaved client sends this upstream; if it fires, look for custom code calling the bundle-request path on the wrong side.
`UnsolicitedBundleReportViolation`	`BundleId`, `IsLoadReported`	Whether the server actually requested that bundle load, or actually recorded the client holding it before an unload report; a mismatched build id here looks identical to a forged report.
`UnexpectedSceneRequestViolation`	`SceneHandle`, `IsLoadRequested`	Nothing on a well-behaved client sends this upstream; look for custom code calling the scene-request path on the wrong side.
`UnsolicitedSceneReportViolation`	`SceneHandle`, `IsLoadReported`	Whether the server actually requested that load or release; a scene handle that was never requested and never recorded held (including the unset handle) lands here too.

Pro-only violations

`PacketTransformRejectedViolation` and `UnauthorizedDespawnViolation` do not exist in a Free build; their events live on Pro partials of `ViolationManager`.

Type	Payload	Default	Check first
`PacketTransformRejectedViolation`	`Channel`, `PayloadByteCount`	No explicit default is passed at the raise site, so it takes `ViolationManager.DefaultAction` (`Ignore`), and because it is the plain framework default it always logs a warning.	Whether the peer is running the same `IPacketTransform` configuration; a stray packet from something that is not this game reaches the port too, so this is not evidence of an attack against a legitimate peer.

`UnauthorizedDespawnViolation`	`SystemId`	`Ignore`	Whether the client and server builds actually agree on that prefab's removal policy; a version skew between them reaches this honestly.

Lag compensation violation

`LagCompensationViewClampedViolation` exists in every edition, but only lag compensation (Pro) raises it: a lag-compensated query raises it, at most once per query, when the shooter's view is older than the rewind window or newer than anything the server has recorded. The view is clamped into range and the shot is still judged. See UnityLagCompensationManager.

Type	Payload	Default	Check first
`LagCompensationViewClampedViolation`	`RequestedTick`, `ClampedTick`	`Ignore`	The shooter's latency against `LagCompensationManager.Maximum` a single raise is a link that spiked, while a steady stream is either a very slow link or a client asking to be judged against the oldest world it can get.

Reading a violation you just saw

Most types default to `Ignore` and stay silent unless something sets `Log` because a well-behaved peer can reach them through an ordinary race (a control handoff, a revocation, a recovery re-serve) or ordinary loss (`RetentionExceededViolation`, `RecoveryUnconfirmedViolation`, `PacketTransformRejectedViolation`). Treat a single raise of any of those as lossy-edge noise and do not kick for it.

The four protocol-direction violations are different: no engine path on a compliant client produces them at all, which is why they default to `Kick`. A raise there means either a deliberately crafted client or your own code driving the request/report path from the wrong side - check the latter first.

What the server trusts

Observation gates everything

Every access check the engine makes, for a state write or for a remote call, starts with the same test: is the sending connection an observer of the system being addressed. That test runs first, before the access value and before whether the sender controls the object. A connection that does not observe a system is rejected outright, regardless of `StateWriteAccess` or `RpcSendAccess`.

This means a client cannot reach a system by guessing its id. An identifier that names something the client was never told about, or has since lost interest in, resolves to nothing on the server. Access - `Controller` or `AnyClient` - only decides what an observer may do once it has cleared this gate. It never substitutes for being told about the object in the first place.

The controller is checked next: a system's controller may always write to it or call it, under any access. An observer that is not the controller is admitted only when the access in force is `AnyClient`.

Nothing travels as a grant

The server tells each observer two things about a system: the access value in force, and a one-bit hint of whether that observer personally may act. Neither is a permission a client can spend on its own. The server re-checks every incoming write and every incoming call against its own copy of the access, on every packet, so a client that forges or replays a favorable-looking declaration gains nothing - the check that matters runs on the receiving side, not the sending side.

The capability hint exists only so a well-behaved client avoids sending something the server is going to refuse anyway. A client that ignores the hint and sends regardless is met with the same enforcement as one that never saw it.

Who actually sent it

An identifier a client places in a payload is a claim about itself, nothing more. What a handler is handed as the sender is the `Connection` the transport resolved for that packet - `RpcContext.SenderConnection` for a call, the equivalent resolved connection for a state write. That resolution happens below any code the client controls, so there is no field a client can set to speak as someone else.

For a call the server relays on to other observers, the relayed copy still carries the original sender's connection id, so attribution survives the hop; a client that receives a relayed call is being told who actually said it, not who forwarded it. Only the authority writes that original-sender id. A call arriving from a client is attributed to the client that sent it, whatever originator id its packet names, so a client cannot dress a call up as coming from another client to fool a handler, the relay's sender exclusion, or `RpcContext.SenderConnection`.

Connections are not players

A `Connection`'s `RemoteAddress` is where it connected from, observed by the server rather than asserted by the client, which is what makes it usable as part of an identity check - but it names a location, not a player, since multiple clients behind the same NAT share one address.

A `Connection.Id` is assigned per session and reused once a connection is gone; nothing about it survives a disconnect. Anything that needs to recognize the same player across a reconnect keys on `Connection.Identity` instead - the value an authenticator stamps in on approval, meant to persist across sessions. Correlating a returning `Identity` with what it previously owned is game code's job; the

engine holds no player-to-object mapping of its own.

Arbitration is not a security boundary

`StateWriteAccess.AnyClient` (Pro) lets every observer, not just the controller, write a system's state. A Free build has no `StateWriteAccess` at all: only a system's controller may write its state, so there is never a contested write to arbitrate.

Contested multi-writer arbitration - resolving which of several observers' simultaneous writes to a system wins - is a Pro feature. Where it exists, the resolution is not adversarially safe: it is arbitrary but stable, and a client that keeps reconnecting to change its own connection id can farm a favorable outcome under it. Arbitration decides whose write is shown when two are equally legitimate; it does not decide whether a write should have been allowed, and it is not a defense against a client trying to win by gaming the mechanism. Anything where that matters needs its own arbiter in game code, sitting on top of the access check, not the arbitration itself.

What Nucleus does not do

Nucleus does not encrypt transport traffic. That is not built in, and adding encryption on top is game code's responsibility.

If you need to protect the bytes themselves - encryption, compression, obfuscation, or a custom integrity check - the one seam for it is the packet-transform layer, `TransportManager.PacketTransform` (Pro-only): it rewrites every packet's payload on its way to and from the transport, on both peers, before anything else in the engine sees it. That layer is not otherwise documented on this page.

A shooter's view is bounded, not trusted

Lag compensation (Pro) judges a client's shot against the moment the client says its screen was showing. That view is the client's claim, so the server bounds it: a view older than `MaximumRewindMilliseconds`, or newer than anything the server has recorded, is clamped into range, still judged, and raises `LagCompensationViewClampedViolation`, ignored by default. Within the window a client can still name any moment, so the window itself is the limit on how far a shooter is favored. The ray is the client's claim too: clip it at walls on the server against the present-time scene, and the queries test it only against the hitboxes the server's own history holds, never against anything the client reports about a target. See UnityLagCompensationManager.

Enforcement only happens on the server

Every check above - observation, controller status, access, arbitration - runs on the server's copy of the system. A client build can run the same code and reach the same verdict locally for prediction or UI purposes, but that verdict changes nothing on the wire. A kick decided by a client, or a write a client's own code declines to send, is not enforcement; the server never asked for that answer and does not use it. The only check that matters is the one the server performs when the packet arrives.

Editor menus and authoring tools

The menu commands

Nucleus adds four items under the `Nucleus` menu. Three are build steps over the project's assets, and the fourth checks those assets without changing them; none is a runtime action. A fifth item, **Window > Nucleus > Network Inspector**, opens a play-mode readout of the running session (see Developer tools in the editor).

Nucleus/Rebuild Network Prefab Collection

Scans the project for every prefab whose root carries a `NetworkSystemObject`, groups them by the asset bundle they're assigned to, and writes each group's `NetworkPrefabCollection` asset. The collection with no asset bundle assignment is the base build's shard, stored under `Assets/Resources/NetworkPrefabCollection.asset` so it loads by name at runtime.

Each prefab is stamped with a local prefab id and a bundle id. Ids are assigned once and then preserved: a prefab that already carries a valid, unique id keeps it on every rescan. Only an unset or duplicated id gets the next free number for its shard. Ids restart at one per shard, because the runtime registry resolves a prefab by the `(bundle, local)` pair, not by the local id alone.

A rebuild that renumbers a duplicated id says so: it logs a warning naming the prefab that carried the id, the prefab that already holds it, and the new id. It logs an error when two collections resolve to the same prefab bundle, because the second of them to load is refused at runtime; give each content bundle assembly its own `[NetworkBundle(n)]`.

Nucleus/Validate Network Prefabs

Checks every prefab in the project and logs each problem against the prefab it names, without changing anything: the prefab identity problems a player build refuses (below), a `NetworkSystemObject` on a prefab's child, a networked component with no `NetworkSystemObject` at or above it, and a `NetworkHitboxRoot` that would record nothing. It runs only on request, because it loads every prefab. Developer tools in the editor lists each check.

Nucleus/Rebuild Network Scene Manifest

Scans the scenes enabled in Build Settings (and, with the Addressables package installed, scenes marked Addressable) and records each one's scene id, path and content bundle in the project's `NetworkSceneManifest`, at `Assets/Resources/NetworkSceneManifest.asset`. A scene keeps its id across the rescan; only a scene the manifest has never seen gets a new one, assigned from the highest id already recorded. Removing a scene from the build leaves its manifest entry in place rather than deleting it, so its id is never handed to a different scene later.

Nucleus/Build Content Bundles

Runs the prefab collection rebuild and the scene manifest rebuild first, then builds every asset bundle a `NetworkPrefabCollection` is assigned to into `Assets/StreamingAssets`, and fills in `NetworkBundleManifest` with each built bundle's id and file name. Running the two rebuilds first matters: a scene or prefab records the bundle id it ships under by reading the collection assigned to that bundle, so building from stale identity is what bakes the wrong id into a shipped bundle.

The automatic steps

Three more steps run without a menu command, so an author never has to remember to invoke them.

`NetworkPrefabBuildCheck` runs before every player build. It makes the same identity check as Validate Network Prefabs, and fails the build when any networked prefab does not carry the identity a rebuild would give it: no base collection, two collections sharing a prefab bundle, a prefab missing from its collection, a prefab with no id or with an id another prefab holds, or a prefab stamped with a different bundle than its collection. The build error lists every problem and tells you to run **Nucleus > Rebuild Network Prefab Collection**, then build again. It stops the build rather than repairing the prefabs itself, because a repair made in the middle of a build could number a server and a client that were built apart differently.

`SceneNetworkObjectStamper` runs on scene save. It assigns a `SceneObjectId` to any `NetworkSystemObject` in the saved scene whose id is unset or collides with another object in the same scene, keeping the first occurrence of any value it finds and handing the next free incremental id to the rest. Duplicating an object that already carries a stamped id is exactly what creates a collision, since the copy carries the same serialized id until the next save reassigns it.

`NetworkSceneManifestAutoRebuild` mints a scene id the moment a scene enters the build: when the Build Settings scene list changes, when a scene asset is imported, moved or deleted, when the project's Addressables entries change, and as a backstop right before a player build. It calls the same reconciliation the manual scene manifest rebuild uses, quietly.

When to re-run each command

Change	Command to run
Added, renamed, moved or duplicated a networked prefab	Rebuild Network Prefab Collection
Added or removed a scene from the build	Nothing to run manually — the auto-rebuild mints the scene id on its own; run the manual command only if you want the log line confirming what mapped
Changed an asset bundle assignment (prefab or scene)	Rebuild Network Prefab Collection, then Build Content Bundles
Ready to ship the built asset bundles	Build Content Bundles (this also re-runs the prefab and scene rebuilds first)

Two authoring traps

A copied prefab carries the original's stamped id. Duplicating a prefab asset in the Project window duplicates its `NetworkSystemObject` component, id and all. Until the next `Rebuild Network Prefab Collection`, the copy has the same prefab id as the original. At runtime the registry keeps whichever of the two registered first and logs an error for the other, so a spawn of one can build the other, and a player build fails until you rebuild. Rebuild after every prefab duplication, not just after adding a new prefab. The rebuild keeps the id on the prefab the collection already lists, which is the original, and on a base over its own variant, gives the copy the next free id, and logs a warning for each prefab it renumbers.

A scene object id only has to be unique within its own scene. `SceneNetworkObjectStamper` reassigns collisions inside one scene, but the same id in two different scenes is fine and expected — the scene half of the correlation comes from the scene's own id, not from the object's. Don't assume a scene object id has to be globally unique across the project; it doesn't.

Why every Nucleus inspector still shows the script row

A custom `[CustomEditor]` replaces Unity's whole default inspector, including the greyed-out script row that lets you ping or open the component's source. `NucleusInspectorGUI.DrawScriptField` redraws that row explicitly, and every Nucleus custom editor calls it before drawing anything else, so that row survives even though the rest of the inspector is custom-drawn. `NucleusInspectorGUI` also provides `DrawPropertiesWithoutScript` (the rest of the default inspector, minus the script row already drawn), plus two field-clamping helpers (`DrawClampedIntSlider`, `DrawNonNegativeLongField`) used by inspectors that need to correct an out-of-range serialized value without flattening a multi-object selection.

`NetworkSceneReferenceDrawer` is a property drawer, not a component inspector: it draws a `NetworkSceneReference` field as a scene-asset picker, with the scene id it currently resolves to (from the project's scene manifest) shown read-only beside it. Nothing about the id is stored on the reference itself — only the scene's GUID is — so the readout always reflects the current manifest rather than a baked-in number.

The custom editors

Twenty-seven components have a `[CustomEditor]`, which is where to look for authoring-time validation and inspector-only readouts:

- `NucleusBehaviourBaseEditor` — covers every `NucleusBehaviourBase` subclass (`editorForChildClasses: true`); adds a runtime readout (linked system, started state, id, state, control) while playing.
- `UnitySceneLoaderEditor` — covers every `UnitySceneLoader` subclass; validates the assigned scene manifest and warns when none is set.
- `UnityAssetBundleLoaderEditor` — covers every `UnityAssetBundleLoader` subclass; validates the assigned bundle manifest and warns when none is set.
- `NetworkSystemObjectEditor`
- `NetworkTransformEditor`
- `NetworkAnimatorEditor`
- `NetworkNavMeshAgentEditor`
- `NetworkHostVisibilityEditor`
- `NetworkInterestObjectEditor`
- `NetworkPlayerSpawnerEditor`
- `NetworkCorrectionVisualizerEditor`: while playing, adds the flash count and the size of the last correction.
- `NetworkInterestOverlayEditor`: shows the overlay's options only while it is switched on, with a key to its colors.
- `NetworkHitboxEditor`
- `NetworkHitboxRootEditor`
- `UnityClientManagerEditor`
- `UnityServerManagerEditor`
- `UnityTransportManagerEditor`
- `UnityInterestManagerEditor`
- `UnityLagCompensationManagerEditor`
- `UnityNetworkLoopManagerEditor`

- UnityPhysicsManagerEditor
- UnitySceneManagerEditor
- UnitySystemManagerEditor
- PhysicsSimulationDriverEditor
- ProjectedRigidbodyEditor
- BlitzRelayTransportEditor
- SynapseTransportEditor

Unity

Documentation for the Unity integration, organized by subject. Each category below favors the Unity component surface; the same subject from plain C# lives under Core API.

- Core, Managers and the Tick Loop
- Systems, Objects and Spawning
- Replicated State and Change Callbacks
- Control, Client State and Prediction
- Messaging and Remote Calls
- Interest Management and Scaling
- Networked Scenes, Content Bundles and Persistence
- Physics Synchronization
- Lag Compensation
- Transports, Connectivity and Hosting
- Serialization, Codegen and the Wire
- Diagnostics and Troubleshooting

What area of interest is

Without any condition, every system replicates to every client

Without any interest condition registered, Nucleus replicates every started NetworkSystem to every authenticated client, every tick. This is the broadcast-to-everyone default, and it is not something interest opts you into: it is what the engine does before interest exists at all.

Interest layers restriction over that default. It never grants a connection anything the default did not already give it. Register no condition anywhere and the evaluation pass does nothing at all - it costs one comparison per tick and stops.

Conditions only restrict

An `IInterestCondition` returns an `InterestResult` carrying an `InterestOpinion` for each effect it declared in its `Capabilities`. That opinion is one of three values: `None` (no opinion, leave it to the others), `Unmet` (veto the effect for this pair), or `Met` (satisfied, no restriction).

`None` is the fold identity. The resolver combines every stacked condition's opinion with a fold equivalent to AND, so an abstaining condition never fights another condition's verdict. Only `Unmet` ever restricts anything; `Met` and `None` are indistinguishable to the resolution and differ only in what the condition means to say. Nothing a condition returns can push a connection past the streamed default - the whole vocabulary is restriction.

Conditions can be registered globally on the `InterestManager`, covering every system that starts from then on, or per-system through the system's own registration. Both stack, and the most restrictive verdict per effect wins across all of them.

Resolution is per pair, inside serialization

Interest resolves one `(NetworkSystem, Connection)` pair at a time. The evaluation runs inside the serialization pass, ahead of the state passes, so a verdict that changes a pair's membership takes effect on the same tick it was decided - a system that just got culled for a connection does not serialize to it that tick.

Because resolution happens on this hot path, an `IInterestCondition` implementation must be a pure read: it must not start or stop systems, mutate observers, or block.

A condition speaks to spawning, to stopping, or to both

`InterestEffect` has two flags, and a condition declares which of them it ever speaks to:

- **Spawn** - whether the system exists for the connection at all. An unmet spawn condition means the system never spawns there, and a system already spawned is despawned exactly as an ordinary despawn releases it.
- **Stop** - whether the system is still replicating for the connection. An unmet stop condition freezes the system for that connection while keeping its object retained, so it can stay visible without costing replication; only a later unmet spawn condition reclaims it.

See stopping-versus-culling for what each looks like from the receiving connection's side.

Cadence and stagger

`InterestManager.EvaluationCadenceTicks` sets how many ticks pass between evaluations of any one system's conditions; the default is 5. A value of 1 evaluates every system every tick. Systems are staggered across the cadence by their `Id`, so on any given tick roughly one cadence-th of the world is

evaluated rather than the whole of it landing on one tick.

The cadence is skipped for two cases that cannot wait for a stagger slot to come around: a connection that just authenticated is evaluated against every started system immediately, and a system that just started is evaluated against every authenticated connection immediately. Both land on the tick they happen, before anything serializes, rather than trickling in as each system's stagger slot arrives.

Evaluation costs one check per system per connection

The evaluation is O(systems ? connections). It is cadenced and staggered to keep that honest, which is fine for demo and mid scale; a world of thousands of systems and connections wants a spatial broadphase feeding it candidate pairs instead, and that is future work rather than something the current pass fakes.

A connection is not registered as an observer the moment it authenticates. Registration - and so interest evaluation - waits for its link to be measured (`Connection.CanReceiveState`), because every timing decision the sender makes about a peer is denominated in its round trip time. A connection authenticated before its link is measured is held back and registered the moment that measurement completes, typically one round trip later, and immediately for a host's own client.

Where to go next

On the API track, see registering-interest-conditions for `InterestManager.AddCondition` and per-system registration. On the Unity track, see the Unity Interest Manager and Network Interest Object components.

Making a host see what its players see

Driving the core API directly? See The host interest report

A host is the server: it holds every object in its own process regardless of what interest rules resolve for anyone. Without anything acting on that report, the person hosting sees every object in the level while their players are only sent a few — the host plays a different game to everyone else.

Turning it on

On the `UnityInterestManager` component, set **Host Interest Enabled**. That is the whole of it. Its acting half, **Host Visibility Enabled**, is already on by default, so switching on the report is what switches on the hiding.

Host Interest Enabled resolves this peer's own client against the world's interest rules too, so the host can be told which objects its client half would not have been sent. It is off by default because the pass costs the host connection an ordinary condition pass per evaluated pair, the same one every remote client already pays.

Host Visibility Enabled is the acting half: it hides an object on a host while the report says its own client half would not have been holding it. Nothing is ever culled from a host by it — the object stays, keeps simulating, keeps replicating. Only its renderers go dark.

What actually hides the object

A `NetworkHostVisibility` component is put on each of a world's networked objects, but only once this peer turns out to be a host, and only then. A client and a dedicated server never attach one and never walk an object for renderers they could not hide anyway.

`UnityInterestManager` sets this up once per `CoreManager` whenever Host Visibility Enabled is set. It watches for the moment this peer first becomes a host with its own client linked. If `InterestManager.HostInterestEnabled` is set at that moment, it sweeps every object already linked and from then on attaches the component to each object as it links. That decision is made once. A peer only becomes a host partway through its own startup, often after its scenes have already spawned a world, so the sweep is what catches those already standing.

NetworkHostVisibility reference

Member	What it tells you
`IsVisibleToHostClient`	True while a host's client half would be holding this object — every moment outside a cull. True on a client, on a dedicated server, and on a host that resolves nothing for itself.
`HiddenRendererCount`	How many renderers this component currently has switched off. Zero on any peer that hides nothing, and zero on an object that is currently shown.
`RefreshRenderers()`	Re-gathers the renderers under this object. Call it after adding or swapping renderers at runtime — a renderer created after this component's `Awake` is otherwise unknown to it.

`HostVisibilityChanged`	Raised as the object crosses into or out of what a host's client half would be holding, after the renderers have already been switched. For a script that has more to hide than renderers.

A renderer the game had already disabled stays disabled: only the renderers this component switched off get switched back on.

Answering the report yourself

Clear **Host Visibility Enabled** to keep the report without the built-in hiding. Nothing is hidden automatically, and it's yours to answer through `NetworkHostVisibility`'s own `HostVisibilityChanged` event, or through `NetworkSystem.HostInterestChanged` directly. That's the path for dimming an object instead of hiding it, or for hiding more than its renderers.

Three things catch people out

- The two inspector switches are copied into the world once, as it comes up, so ticking either while playing changes nothing. Set **Host Interest Enabled** before the host starts.
- The engine's own `InterestManager.HostInterestEnabled` is read on every interest pass, so code can change it mid-session, but the decision to hide is made once, when the peer first becomes a host with its client linked. Turning it on after that gives you the report with nothing hiding on it. Turning it off returns every object's membership to `InterestMembership.Streamed`, which shows the object again. Code that sets it before the host starts must run after `UnityInterestManager` has pushed its settings, or the inspector's copy overwrites it.
- A dedicated server has no client half to answer for. Leave both switches alone.
- Hiding only happens on `InterestMembership.Unspawned`. A system that's merely stopped for a player still keeps its object standing, frozen rather than gone — hiding on a stop would show the host less than that player actually sees.

Interest Management and Scaling

Pages in this category:

- Network Interest Object
- Unity Interest Manager
- What area of interest is
- Limit what a client receives by distance
- An object is missing, or will not go away
- What scales and what does not
- Frozen or gone: what a stop does to the object
- Level of detail in Unity
- Making a host see what its players see

An object is missing, or will not go away

Before working through the cases below, it helps to see what interest resolved. Add a `NetworkInterestOverlay` to the player object, switch it on, and select the object on the server or host while playing: a line goes to every other networked object, colored by whether the player is streamed it, stopped for it, or does not have it at all (see Developer tools in the editor).

A client receives objects from a scene it never loaded

No rule gates scenes by default in a way you can see failing: an ungated spawn is not an error anywhere on the server, it simply lands in the wrong scene on the client. The Unity inspector for `UnityInterestManager` warns about exactly this when its rule list has no scene rule registered: "No rule gates scenes. A client will receive objects from scenes it never loaded, which puts them in the wrong scene rather than raising an error, and several open instances of one scene cannot be told apart by distance."

Fix: add the scene rule to the world's interest conditions unless you are arbitrating scene membership with a rule of your own.

Nothing is ever culled, however large the stop distance is

A distance condition reads positions through its own `TryReadSourcePosition` (the Unity integration's conditions read the linked GameObject's transform), not through the world reader registered with `InterestManager.SetPositionReader`. When it cannot measure a pair it answers unmeasurable, and an unmeasured pair is never allowed to restrict anything, so no cutoff culls or stops it. That happens when the connection controls nothing yet, so there is no point to measure from, or when `TryReadSourcePosition` returns `false` for the object or for every object the connection controls.

Fix: make sure each player controls the object it plays through, and that the condition's `TryReadSourcePosition` answers for both the evaluated object and the controlled ones. The world reader is used only by level of detail: without one, level of detail logs an error once and every pair reads as Near, but culling is unaffected.

An object disappears at range when it should only freeze, or the reverse

`DistanceInterestLadder.Resolve` returns an unmet spawn past the despawn cutoff and only an unmet stop past the stop cutoff — despawn always outranks stop from the same measured distance. Whether an unmet stop actually removes the object is `NetworkSystem.DespawnWhenStoppedEnabled`: off, the object freezes in place and keeps its last state; on, it is treated as culled once stopped.

On a grouped object (several systems sharing one `NetworkSystemGroup`), a stop is a vote, not a unilateral despawn: a member votes only when its stop is unmet *and* its `DespawnWhenStoppedEnabled` is set, and the object leaves only once every member votes to. A member that freezes instead of voting holds the object in place by abstaining, so one system in the group with `DespawnWhenStoppedEnabled` off is enough to keep a "should have despawned" object sitting there. An ungrouped system has no sibling to defer to, so its own unmet stop with the flag on despawns it directly.

Fix: check `DespawnWhenStoppedEnabled` on every system in the group, not just the one you're staring at.

Objects flicker in and out at a cull boundary

There is no hysteresis on a distance cull. `Resolve` is two bare distance-versus-cutoff comparisons with nothing damping the boundary, so a player oscillating around a cutoff between evaluations flips the verdict

every pass. The only thing standing between a player's movement and a flicker is `EvaluationCadenceTicks`: distance is only re-measured once per that many ticks, so as long as a player cannot close the gap between "just inside" and "just past" the cutoff within one cadence, the flicker never has a chance to fire.

Hysteresis does exist in this codebase, but only on the level-of-detail ladder (a fraction-of-a-rung band a distance has to re-cross before it switches back). It cannot fix this: a level-of-detail band is only a reading for the game. It never changes what is sent or how often, nor whether an object spawns, stops, or despawns, so it has no bearing on a cull boundary.

Fix: widen the gap between the stop/despawn cutoff and how far a player can move in one `EvaluationCadenceTicks`, either by raising the cutoff, lowering the cadence, or both. There is no setting to smooth the boundary itself.

A new client sees nothing for a second or two

Registration as an observer waits for more than authentication. A connection authenticates before its round trip time is known, and it is held back from every started system until `Connection.CanReceiveState` is true — which happens once its link is measured, typically one round trip later, or immediately for a host's own client. A client stuck in this window is authenticated but registered against nothing, so it appears to see no objects at all for a moment.

This is expected and self-resolving; there is nothing to configure. If the gap never closes, check that the transport is actually completing a round trip for that connection.

A join or scene load arrives as one huge tick and drops

Spawn pacing exists to prevent exactly this: `InterestManager.MaximumSpawnsPerTick` caps how many objects may start replicating to one client on a single tick, deferring the rest to later ticks. Left at zero it admits everything at once — a full join or scene load lands as a single, maximally expensive tick, the worst one to lose a packet on.

To check whether the ceiling is actually the bottleneck, read:

- `InterestManager.TotalSpawnsDeferred` — total deferrals so far; a world that never reaches the ceiling stays near zero.
- `Connection.SystemsAwaitingSpawnAdmissionCount` — how many spawns a specific connection is still owed.
- A `LogWarning` fires once a connection has been owed spawns for `InterestManager.SpawnAdmissionStallWarningTicks` (90) ticks without one being admitted, naming the connection and the current `MaximumSpawnsPerTick`.

Fix: raise `MaximumSpawnsPerTick`, or spawn less at once. A `NetworkSystemGroup` is always admitted whole rather than starved, so also check for a single group larger than the ceiling — that alone will trigger the stall warning.

Setting the technique or axes logs an error and does nothing

`InterestManager.SetInterestTechnique` is refused, with a logged error, once any distance condition has already registered in that world — tracked internally the moment a position condition is added, globally or per-system. One technique and one plane govern a world for its whole lifetime because the shared spatial index applies its axis mask when a point is filed and again when it is queried; changing either mid-session would have one pass measuring in two of them.

Fix: call `SetInterestTechnique` during setup, before any system carrying a distance rule can spawn —

never after.

A distance rule with no spawn range, only a stop range

This is specific to the authored Unity rule (`AuthoredDistanceInterestCondition`), not the underlying condition type. The inspector-authored rule only exposes a stop distance; it always builds its condition with the despawn cutoff unset (`DistanceInterestLadder.UnsetDistance`), so an object authored this way never culls by distance at all, only stops.

The underlying type has no such limitation. `DistanceInterestCondition` (Unity's own binding of the engine-neutral `DistanceInterestConditionBase`, not the inspector-authored rule) takes a `stopDistance` and a `spawnDistance` independently in its constructor, and so does the `DistanceInterestLadder` beneath it. Construct one yourself in code (or register a global condition) to get culling by distance instead of authoring through the inspector.

Every distance lever on this page is Pro-only

Distance interest conditions, the spatial grid, level of detail, and spawn pacing are all Pro-only — their implementations live in files suffixed `.Pro.cs`: `InterestManager.Conditions.Pro.cs`, `DistanceInterestLadder.Pro.cs`, `DistanceInterestConditionBase.Pro.cs`, `SpatialInterestGrid.Pro.cs`, `InterestManager.LevelOfDetail.Pro.cs`, `InterestManager.SpawnPacing.Pro.cs`, and the Unity-side `AuthoredDistanceInterestCondition.Pro.cs` / `DistanceInterestCondition.Pro.cs`. On a free build these files are absent, so none of the above levers exist: no distance rule, no spatial grid, no level of detail, no spawn ceiling. The scene rule and the base interest system are not gated this way and ship in free.

If a build behaves as though one of these settings simply doesn't apply, confirm it's a free build before debugging further.

Level of detail in Unity

> **Driving the core API directly?** See Interest level of detail.

A band changes nothing about what is sent. It neither culls an object nor alters its rate; it is a measurement your game reads and acts on.

Enabling it

Level of detail is set on the **Unity Interest Manager** component:

- **Level Of Detail Enabled** — turns the pass on for this world.
- **Near Distance** — the distance past which an object bands as Medium.
- **Far Distance** — the distance past which an object bands as Far.
- **Hysteresis** — the fraction of a rung an object must travel back inside before it returns to the nearer band. Default `0.1`.
- **Level Of Detail Cadence Ticks** — how many ticks pass between one resolution and the next. Default `10`.

The far distance must be larger than the near one, and both above zero. Equal rungs make a band unreachable, and until that is fixed nothing is banded at all.

Why two rungs, not more

The ladder has exactly two boundaries: near/medium and medium/far. That gives three bands — Near, Medium, Far — and nothing finer. A band is a measurement your game reacts to, not a rate the engine drives, so more rungs only add more cases for your code to branch on without changing what gets sent.

Hysteresis exists for the player who walks back and forth across a boundary. Without it, a position sitting exactly on a rung flips bands every resolution. The hysteresis fraction holds the object in its farther band until it has come back far enough inside the nearer rung, so a boundary a player straddles doesn't chatter between two bands tick after tick.

Opting an object in

An object is measured only where it asked to be. Call `SetLevelOfDetailEnabled` on its `NetworkSystem` from a `NucleusBehaviour` or any other script:

```
networkSystem.SetLevelOfDetailEnabled(true);
```

Measuring a distance for an object nobody will ask the band of is the one cost this feature has, so it's opt-in per system rather than on for the world. Turning it off drops every band the system held; a later read answers Near again rather than whatever was last measured.

Reading the band for a player

```
InterestLevelOfDetail level = networkSystem.GetInterestLevelOfDetail(connection);
```

- **Near** — inside the near rung, which is no level of detail at all. This is also what an unconfigured ladder, an unmeasurable position, and a peer's own controlled object all resolve to.
- **Medium** — between the near rung and the far one.
- **Far** — beyond the far rung.

Level of detail ships only in Pro

Level of detail is Pro. The whole file set behind it is `*.Pro.cs`; a free build has no bands at all.

Level of detail in Unity

> **Driving the core API directly?** See Interest level of detail.

A band changes nothing about what is sent. It neither culls an object nor alters its rate; it is a measurement your game reads and acts on.

Enabling it

Level of detail is set on the **Unity Interest Manager** component:

- **Level Of Detail Enabled** — turns the pass on for this world.
- **Near Distance** — the distance past which an object bands as Medium.
- **Far Distance** — the distance past which an object bands as Far.
- **Hysteresis** — the fraction of a rung an object must travel back inside before it returns to the nearer band. Default `0.1`.
- **Level Of Detail Cadence Ticks** — how many ticks pass between one resolution and the next. Default `10`.

The far distance must be larger than the near one, and both above zero. Equal rungs make a band unreachable, and until that is fixed nothing is banded at all.

Why two rungs, not more

The ladder has exactly two boundaries: near/medium and medium/far. That gives three bands — Near, Medium, Far — and nothing finer. A band is a measurement your game reacts to, not a rate the engine drives, so more rungs only add more cases for your code to branch on without changing what gets sent.

Hysteresis exists for the player who walks back and forth across a boundary. Without it, a position sitting exactly on a rung flips bands every resolution. The hysteresis fraction holds the object in its farther band until it has come back far enough inside the nearer rung, so a boundary a player straddles doesn't chatter between two bands tick after tick.

Opting an object in

An object is measured only where it asked to be. Call `SetLevelOfDetailEnabled` on its `NetworkSystem` from a `NucleusBehaviour` or any other script:

```
networkSystem.SetLevelOfDetailEnabled(true);
```

Measuring a distance for an object nobody will ask the band of is the one cost this feature has, so it's opt-in per system rather than on for the world. Turning it off drops every band the system held; a later read answers Near again rather than whatever was last measured.

Reading the band for a player

```
InterestLevelOfDetail level = networkSystem.GetInterestLevelOfDetail(connection);
```

- **Near** — inside the near rung, which is no level of detail at all. This is also what an unconfigured ladder, an unmeasurable position, and a peer's own controlled object all resolve to.
- **Medium** — between the near rung and the far one.
- **Far** — beyond the far rung.

Level of detail ships only in Pro

Level of detail is Pro. The whole file set behind it is `*.Pro.cs`; a free build has no bands at all.

Limit what a client receives by distance

Driving the core API directly? See Distance interest conditions

A Distance rule stops an object's systems replicating to a player once that player is too far away. It is an inspector-only rule with two places to add it, one setting to tune, and one warning to know about.

Where a rule lives

A Distance rule can be added in two places, and they mean different things:

- **World Rules**, on `UnityInterestManager` — applies to every object in the world.
- **Rules**, on `NetworkInterestObject` — applies to this prefab only, and stacks on top of whatever the world rules already say.

Both lists take the same authored rule types. Add the rule to the manager when the cutoff should be a world-wide policy; add it to the prefab when only that object should stop being sent at distance.

Adding the rule

Add a Distance rule to either list, then set its **Stop Distance**: the distance past which the object stops replicating to a player.

Leaving Stop Distance at `0` disables the rule entirely — it builds nothing, so an untouched entry costs nothing.

What the distance is measured from

The distance is measured from the `NetworkSystem`s the client controls, not from a camera and not from a "player" the engine has no concept of. A client controlling nothing measurable is never restricted by the rule.

Interest Axes

`UnityInterestManager` has an **Interest Axes** setting that governs how every distance in the world is measured: the full volume (`Xyz`), or one of the three planes that drops a single axis. A top-down world sets this to **Xz** so height nobody travels is ignored, and a player standing on a floor above something is not treated as far away from it.

Interest Technique

Interest Technique on `UnityInterestManager` chooses how those distances are reached: **Pairwise** compares each object against each client's own objects directly, and is cheapest until there are many of both; the shared spatial grid narrows to nearby objects first. Both give the same answers.

Leave it on Pairwise until the cutoff excludes most pairs. Settle the choice before anything carrying a distance rule spawns — it is refused afterward.

One distance rule per system

A `NetworkSystem` accepts only one rule that decides by distance. A second one is refused, and logged, as the object spawns. The `NetworkInterestObject` inspector counts the rules in its **Rules** list and warns when more than one decides by distance, so the mistake is visible before it ships rather than only in a log line at runtime.

Distance rules ship only in Pro

Distance is a Pro feature. A free build ships the interest system and the world-wide Scene rule, but no distance rule anywhere — on a free build, a `NetworkInterestObject`'s **Rules** list has no shipped entry to add at all.

Limit what a client receives by distance

Driving the core API directly? See Distance interest conditions

A Distance rule stops an object's systems replicating to a player once that player is too far away. It is an inspector-only rule with two places to add it, one setting to tune, and one warning to know about.

Where a rule lives

A Distance rule can be added in two places, and they mean different things:

- **World Rules**, on `UnityInterestManager` — applies to every object in the world.
- **Rules**, on `NetworkInterestObject` — applies to this prefab only, and stacks on top of whatever the world rules already say.

Both lists take the same authored rule types. Add the rule to the manager when the cutoff should be a world-wide policy; add it to the prefab when only that object should stop being sent at distance.

Adding the rule

Add a Distance rule to either list, then set its **Stop Distance**: the distance past which the object stops replicating to a player.

Leaving Stop Distance at `0` disables the rule entirely — it builds nothing, so an untouched entry costs nothing.

What the distance is measured from

The distance is measured from the `NetworkSystem`s the client controls, not from a camera and not from a "player" the engine has no concept of. A client controlling nothing measurable is never restricted by the rule.

Interest Axes

`UnityInterestManager` has an **Interest Axes** setting that governs how every distance in the world is measured: the full volume (`Xyz`), or one of the three planes that drops a single axis. A top-down world sets this to **Xz** so height nobody travels is ignored, and a player standing on a floor above something is not treated as far away from it.

Interest Technique

Interest Technique on `UnityInterestManager` chooses how those distances are reached: **Pairwise** compares each object against each client's own objects directly, and is cheapest until there are many of both; the shared spatial grid narrows to nearby objects first. Both give the same answers.

Leave it on Pairwise until the cutoff excludes most pairs. Settle the choice before anything carrying a distance rule spawns — it is refused afterward.

One distance rule per system

A `NetworkSystem` accepts only one rule that decides by distance. A second one is refused, and logged, as the object spawns. The `NetworkInterestObject` inspector counts the rules in its **Rules** list and warns when more than one decides by distance, so the mistake is visible before it ships rather than only in a log line at runtime.

Distance rules ship only in Pro

Distance is a Pro feature. A free build ships the interest system and the world-wide Scene rule, but no distance rule anywhere — on a free build, a `NetworkInterestObject`'s **Rules** list has no shipped entry to add at all.

Network Interest Object

Driving the core API directly? See Registering interest conditions.

One component authors interest rules for the whole prefab

`NetworkInterestObject` authors interest rules once for a prefab, and those rules apply to every `NetworkSystem` on the object. An object built from several systems is filtered as the one thing a player sees, not as separate components each judged on their own.

It requires a `NetworkSystemObject` on the same object, and it does nothing on a client: only the server resolves interest, so the component registers no conditions and hooks nothing there.

Rules

The `Rules` list holds `[SerializeReference]` entries of type `AuthoredInterestCondition`, drawn by an add-button list rather than the default inspector, because a `[SerializeReference]` list is not something the default inspector can author. Each entry builds its own `IInterestCondition` for every system on the object via `AuthoredInterestCondition.CreateCondition()`.

In a free build, the list offers one rule: **Controller Only** (`AuthoredControllerOnlyInterestCondition`), which shows the object only to the client that controls it, for a player's inventory or any other private state. It authors nothing, because the controller is the whole of the question. Add it to the prefabs that need it rather than to the Unity Interest Manager's **World Rules**: there it applies to every object, and an object no client controls is then shown to no client. The Scene rule (`AuthoredSceneInterestCondition`) is world-scope only (`IsObjectScope` is `false` and `CreateCondition()` returns `null`), so it is offered on the Unity Interest Manager instead of on a prefab. A game can add its own rule by deriving from `AuthoredInterestCondition` and marking the derived type `[Serializable]`; it sits in the list beside whatever ships.

An empty `Rules` list filters nothing. It does not mean "hide from everyone" — rules only ever restrict what would otherwise replicate.

Despawn When All Stopped

`_despawnWhenAllSystemsStoppedEnabled`, labeled **Despawn When All Stopped** in the inspector, defaults to `true`. When on, the object is reclaimed from a player once every system on it has stopped for that player, rather than being left standing there frozen.

Turn it off for something a player should keep seeing after it stops updating — a landmark or a corpse. The editor's own warning is explicit about the tradeoff: with the toggle off, nothing on the component will ever take the object away again; something else has to.

InterestStopResolving

The component re-surfaces `NetworkSystem.InterestStopResolving` as its own `InterestStopResolving` event. A handler tests the event's `InterestStopContext.NetworkSystem` against the system it owns and leaves the context alone otherwise; it can waive a stop the object's rules resolved, or stop a player the rules did not.

Subscribing to this event is itself the reason the evaluation pass walks the object: gaining the first listener hooks `InterestStopResolving` on every currently linked system, and losing the last listener unhooks it again, so an object that authors rules and subscribes to none of this pays for none of it.

Rules are re-registered from Awake so a pooled object recovers them

Rules are re-registered from `Awake`, not from a subscription that could be missed. A system linked before this component woke — the case for a context-aware rent that resolves inside the prefab's own `Awake` — raises nothing for this component to subscribe to. So `Awake` walks every system already linked (`_networkSystemObject.Systems`) and adopts each one, which is how a pooled object's next life re-registers everything its previous life's pool return dropped.

One distance rule per system

At most one authored entry on an object may report `IsPositionCondition => true`. A second is refused where it registers, because it would resolve the same pairs again to reach an answer it cannot change. The inspector counts position-condition entries and warns at authoring time when more than one is present, rather than leaving the mistake to a log line on the first spawn.

Network Interest Object

Driving the core API directly? See Registering interest conditions.

One component authors interest rules for the whole prefab

`NetworkInterestObject` authors interest rules once for a prefab, and those rules apply to every `NetworkSystem` on the object. An object built from several systems is filtered as the one thing a player sees, not as separate components each judged on their own.

It requires a `NetworkSystemObject` on the same object, and it does nothing on a client: only the server resolves interest, so the component registers no conditions and hooks nothing there.

Rules

The `Rules` list holds `[SerializeReference]` entries of type `AuthoredInterestCondition`, drawn by an add-button list rather than the default inspector, because a `[SerializeReference]` list is not something the default inspector can author. Each entry builds its own `IInterestCondition` for every system on the object via `AuthoredInterestCondition.CreateCondition()`.

In a free build, the list ships with no entry at all. The Scene rule (`AuthoredSceneInterestCondition`) is world-scope only — `IsObjectScope` is `false` and `CreateCondition()` returns `null` — so it is offered on the Unity Interest Manager instead of on a prefab. A game can add its own rule by deriving from `AuthoredInterestCondition` and marking the derived type `[Serializable]`; it sits in the list beside whatever ships.

An empty `Rules` list filters nothing. It does not mean "hide from everyone" — rules only ever restrict what would otherwise replicate.

Despawn When All Stopped

`_despawnWhenAllSystemsStoppedEnabled`, labeled **Despawn When All Stopped** in the inspector, defaults to `true`. When on, the object is reclaimed from a player once every system on it has stopped for that player, rather than being left standing there frozen.

Turn it off for something a player should keep seeing after it stops updating — a landmark or a corpse. The editor's own warning is explicit about the tradeoff: with the toggle off, nothing on the component will ever take the object away again; something else has to.

InterestStopResolving

The component re-surfaces `NetworkSystem.InterestStopResolving` as its own `InterestStopResolving` event. A handler tests the event's `InterestStopContext.NetworkSystem` against the system it owns and leaves the context alone otherwise; it can waive a stop the object's rules resolved, or stop a player the rules did not.

Subscribing to this event is itself the reason the evaluation pass walks the object: gaining the first listener hooks `InterestStopResolving` on every currently linked system, and losing the last listener unhooks it again, so an object that authors rules and subscribes to none of this pays for none of it.

Rules are re-registered from Awake so a pooled object recovers them

Rules are re-registered from `Awake`, not from a subscription that could be missed. A system linked before

this component woke — the case for a context-aware rent that resolves inside the prefab's own `Awake` — raises nothing for this component to subscribe to. So `Awake` walks every system already linked (`_networkSystemObject.Systems`) and adopts each one, which is how a pooled object's next life re-registers everything its previous life's pool return dropped.

One distance rule per system

At most one authored entry on an object may report `IsPositionCondition => true`. A second is refused where it registers, because it would resolve the same pairs again to reach an answer it cannot change. The inspector counts position-condition entries and warns at authoring time when more than one is present, rather than leaving the mistake to a log line on the first spawn.

Frozen or gone: what a stop does to the object

Driving the core API directly? See Stopping versus culling.

The two outcomes

When a `NetworkSystem` stops for a connection, that stop resolves to one of two things. A stopped system keeps its object on the receiver: the object stays where it was, frozen, and is cheap to undo, because bringing it back is just resuming the delta stream rather than spawning anything. A reclaimed system is despawned instead, and released exactly as an ordinary despawn releases it.

Which one happens is set with `NetworkSystem.DespawnWhenStoppedEnabled`. It is independent of whatever stopped the system in the first place: conditions and `InterestStopResolving` decide *whether* a system stops, and this field decides what that stop then costs the receiver.

Despawn When All Stopped

On `NetworkInterestObject` this is the **Despawn When All Stopped** checkbox. It is on by default, and it sets `DespawnWhenStoppedEnabled` to the same value on every `NetworkSystem` linked to the object, because the object leaves whole or not at all.

For an object carrying a single system, the checkbox is a straightforward decision. For an object carrying several systems, it is a vote and not a decision: the object is reclaimed only once *every* member is stopped and asking to be. One member that still wants to be heard — because its own conditions haven't stopped it, or because a script handling `InterestStopResolving` waived the stop — keeps the whole object where it is for that player.

When to turn it off

Turn off **Despawn When All Stopped** for something a player should go on seeing after it stops updating: a landmark, a corpse, anything whose value is in still being there rather than in still updating. Left frozen, it stays standing until something else takes it away.

The inspector warns about this once the field is cleared: nothing in interest will ever take the object away again. Despawning it is then a job for other game logic entirely — a lifetime timer, an explicit despawn call, whatever fits the object.

Handing a system back: InterestStopResolving

Authored rules can only restrict — a condition can stop a system but never un-stop one. `NetworkInterestObject.InterestStopResolving` is the seam for a decision an authored rule can't express: waiving a stop the rules resolved, or stopping a pair the rules didn't.

```
public event NetworkSystem.InterestStopResolvingHandler InterestStopResolving
```

It fires once per (system, connection) pair on the interest cadence, carrying one `InterestStopContext` for the whole object rather than one subscription per member. A handler tests `InterestStopContext.NetworkSystem` against the system it cares about and leaves the context alone otherwise:

```
private void OnInterestStopResolving(ref InterestStopContext context)
{
    if (context.NetworkSystem != _mySystem)
        return;

    context.IsStopped = false; // keep this member alive for the player
}
```

Declining a stop this way keeps the whole object with that player while **Despawn When All Stopped** is on — that's the lever for letting something cheap outlive the rest of the object it sits on.

What a subscriber is held to

`InterestStopResolving` runs on the network loop thread, inside the serialization pass, once per (system, connection) pair. Handlers are held to the same contract as an interest condition: pure, cheap reads that neither start nor stop systems, mutate observers, nor block.

Gotcha: subscribing costs a walk

A `NetworkSystem` is only visited by the interest pass if it has something to evaluate — either registered conditions or a subscriber to `InterestStopResolving`. Subscribing is on its own a reason for the pass to walk the object: an otherwise rule-free prefab that hooks `InterestStopResolving` starts paying for the evaluation pass even though it authored no conditions at all. Only subscribe where the decision is actually needed.

Frozen or gone: what a stop does to the object

Driving the core API directly? See Stopping versus culling.

The two outcomes

When a `NetworkSystem` stops for a connection, that stop resolves to one of two things. A stopped system keeps its object on the receiver: the object stays where it was, frozen, and is cheap to undo, because bringing it back is just resuming the delta stream rather than spawning anything. A reclaimed system is despawned instead, and released exactly as an ordinary despawn releases it.

Which one happens is set with `NetworkSystem.DespawnWhenStoppedEnabled`. It is independent of whatever stopped the system in the first place: conditions and `InterestStopResolving` decide *whether* a system stops, and this field decides what that stop then costs the receiver.

Despawn When All Stopped

On `NetworkInterestObject` this is the **Despawn When All Stopped** checkbox. It is on by default, and it sets `DespawnWhenStoppedEnabled` to the same value on every `NetworkSystem` linked to the object, because the object leaves whole or not at all.

For an object carrying a single system, the checkbox is a straightforward decision. For an object carrying several systems, it is a vote and not a decision: the object is reclaimed only once *every* member is stopped and asking to be. One member that still wants to be heard — because its own conditions haven't stopped it, or because a script handling `InterestStopResolving` waived the stop — keeps the whole object where it is for that player.

When to turn it off

Turn off **Despawn When All Stopped** for something a player should go on seeing after it stops updating: a landmark, a corpse, anything whose value is in still being there rather than in still updating. Left frozen, it stays standing until something else takes it away.

The inspector warns about this once the field is cleared: nothing in interest will ever take the object away again. Despawning it is then a job for other game logic entirely — a lifetime timer, an explicit despawn call, whatever fits the object.

Handing a system back: InterestStopResolving

Authored rules can only restrict — a condition can stop a system but never un-stop one. `NetworkInterestObject.InterestStopResolving` is the seam for a decision an authored rule can't express: waiving a stop the rules resolved, or stopping a pair the rules didn't.

```
public event NetworkSystem.InterestStopResolvingHandler InterestStopResolving
```

It fires once per (system, connection) pair on the interest cadence, carrying one `InterestStopContext` for the whole object rather than one subscription per member. A handler tests `InterestStopContext.NetworkSystem` against the system it cares about and leaves the context alone otherwise:

```
private void OnInterestStopResolving(ref InterestStopContext context)
{
    if (context.NetworkSystem != _mySystem)
        return;

    context.IsStopped = false; // keep this member alive for the player
}
```

Declining a stop this way keeps the whole object with that player while **Despawn When All Stopped** is on — that's the lever for letting something cheap outlive the rest of the object it sits on.

What a subscriber is held to

`InterestStopResolving` runs on the network loop thread, inside the serialization pass, once per (system, connection) pair. Handlers are held to the same contract as an interest condition: pure, cheap reads that neither start nor stop systems, mutate observers, nor block.

Gotcha: subscribing costs a walk

A `NetworkSystem` is only visited by the interest pass if it has something to evaluate — either registered conditions or a subscriber to `InterestStopResolving`. Subscribing is on its own a reason for the pass to walk the object: an otherwise rule-free prefab that hooks `InterestStopResolving` starts paying for the evaluation pass even though it authored no conditions at all. Only subscribe where the decision is actually needed.

Unity Interest Manager

This component is the inspector-side driver for the core InterestManager

`UnityInterestManager` is the inspector-side driver for the core `InterestManager`. It carries the world-wide interest rules this peer is authored with, controls how often interest is re-resolved, and, in Pro, how spawns are paced and how distance is measured. Every field is pushed onto the core `InterestManager` in `ManagersInstantiated`, so script execution order cannot lose a setting. Both host switches are copied at that point, so ticking either in the inspector mid-session changes nothing. The engine reads its own `InterestManager.HostInterestEnabled` on every pass, but whether a host hides objects is decided once, when the peer first becomes a host with its own client linked.

World Rules

A `[SerializeReference]` list of `AuthoredInterestCondition` entries. Each one registers against every `NetworkSystem` that starts, for the whole world — not one object. On `Reset`, the list is seeded with a single entry: the Scene rule (`AuthoredSceneInterestCondition`).

Conditions can only restrict. Anything registered here narrows what would otherwise replicate to everybody. If no entry in the list gates scenes, the inspector raises a warning: a client will receive objects from scenes it never loaded, several open instances of one scene cannot be told apart by distance, and nothing at runtime reports the fault — it just lands objects in the wrong scene.

Evaluation Cadence Ticks

Default: 5 (`InterestManager.DefaultEvaluationCadenceTicks`).

How many ticks pass between one interest evaluation pass and the next for any one system. A joining client and a newly started system are always resolved immediately, off the cadence. Systems are staggered by Id, so a given tick evaluates roughly one cadence-th of the world rather than all of it at once.

Host Interest Enabled / Host Visibility Enabled

- **Host Interest Enabled**: resolves this peer's own client against the interest rules too, so a host can be told which objects its client half would not have been sent. Off by default: the pass costs the host connection an ordinary condition pass per evaluated pair, and a world that never reads the answer would be paying for nothing. Nothing is culled from a host by this switch alone.
- **Host Visibility Enabled**: hides an object while the report above says the host's own client half would not have been holding it. On by default. Setting **Host Interest Enabled** is the whole of turning host hiding on, since this switch is already set (see Making a host see what its players see). Clearing it keeps the report without acting on it, letting a world hide more than renderers, dim instead, or drive its own behavior from `NetworkSystem.HostInterestChanged`.

When enabled, a `NetworkHostVisibility` component is put on each networked object as it links, only once the peer turns out to be a host.

A free build draws only some of these fields

In a free build the inspector draws only **World Rules**, **Evaluation Cadence Ticks**, and the two host switches. **Maximum Spawns Per Tick** and the entire **Distance Measurement** and **Level Of Detail** sections belong to Pro-only partial files (`UnityInterestManager.SpawnPacing.Pro.cs`,

`UnityInterestManager.LevelOfDetail.Pro.cs`). Those fields do not exist in a free build, so the editor skips drawing them rather than disabling them.

Maximum Spawns Per Tick — Pro

Default: `500`; `0` means unpaced.

How many objects may start replicating to any one client per tick; the rest wait for the ticks after. When set to `0` (`InterestManager.UnlimitedSpawnsPerTick`), the inspector shows an info note: spawns are not paced, and a join or scene load arrives as one tick however large it is.

Distance Measurement — Pro

- **Interest Technique**: how every distance in this world is measured (pairwise comparison, or a spatial grid).
- **Interest Axes**: which axes distances are measured across (full volume, or height dropped for surface play).

Both fields must be settled before anything carrying a distance rule spawns. Changing either after a distance condition has registered is refused, and the engine logs an error.

Level Of Detail — Pro

- **Level Of Detail Enabled**: measures each opted-in object's distance from each player into a Near, Medium or Far band your game can read. It changes nothing about what is sent or how often, and nothing is culled by it.
- **Near Distance**: inside this distance an object reads as Near (no level of detail).
- **Far Distance**: past this distance an object reads as Far, the coarsest band.
- **Hysteresis**: how far inside a rung an object must travel before it returns to the nearer band, as a fraction of that rung; stops an object sitting on a boundary from switching band every pass.
- **Cadence Ticks**: how many ticks pass between one distance measurement of an object and the next.

The rungs must be positive and strictly ascending (far greater than near). If they are equal or zero, the inspector shows an error: nothing is banded until this is fixed.

What scales and what does not

The four levers

There are four ways to spend less on replication, and they save different amounts because they cut at different points in the pipeline.

Don't send it at all. `InterestEffect.Spawn` governs whether a Connection is ever made an observer of a system in the first place. A condition that vetoes spawn keeps a peer from ever holding an object it has no reason to see. This is the cheapest lever there is: nothing is serialized, nothing is queued, nothing rides the wire.

Stop sending it. `InterestEffect.Stop` pauses replication to a peer that already holds the object without reclaiming it client-side. Cheaper than a fresh spawn later (no re-priming full state), but the object still exists on the peer and still occupies whatever bookkeeping a held object costs.

Send it less often. `SendInterval` paces a member's deltas: `SendInterval.Normal` sends every changed tick, a wider interval accumulates changes and rides at most one delta per interval. This only applies under `TransmissionMode.Interval`; a member on `TransmissionMode.Divine` governs its own rate through projection instead. A full serialization (spawn, resync, recovery, reconcile) always carries the current value regardless of the interval, so this lever never affects the moments that matter most.

Send it smaller. `TransmissionMode` and per-value compression level shrink each individual write without changing how often it happens. This is the smallest saving of the four, because it doesn't remove a send, a tick, or a peer's membership — it just makes the send that was already happening cost fewer bits.

Bit-level detail — exactly which encoding a value gets — is a measurement that feeds a choice among these four, not a fifth lever. You don't budget "smaller encoding" as its own line item; you use it to decide whether a lever above is worth pulling.

Three different bottlenecks

"Replication is too expensive" is not one problem. It's usually one of three, and each has its own fix.

Per-client egress. How many bytes one peer's connection pushes per second. This is what `TotalBytesSent` and the per-object/state-only bandwidth stat providers measure. The fix is fewer or smaller sends to that peer: spawn/stop culling first, interval and compression second.

Per-tick burst size. How much of that egress lands in a single tick rather than spread across several. A join, a scene load, a bundle release, or a peer walking back into range can put every system's priming full into one combined stream on one tick. A wide tick is all-or-nothing on an unreliable channel: one lost segment discards the whole tick, and a retained set resends at `EffectiveGlobalRedundancy` on top of that. `InterestManager.MaximumSpawnsPerTick` (default 500, 0 for unlimited) is the lever here — it spreads a burst of admissions across consecutive ticks instead of fixing the total bytes sent. It does nothing for steady-state egress; a world under the ceiling never touches it.

The O(systems ? connections) resolution pass. How much work the server itself does deciding who sees what, independent of what gets sent. Every registered `IInterestCondition` is evaluated per (system, Connection) pair on `InterestManager.EvaluationCadenceTicks`, and that cost scales with systems times connections regardless of how few of those pairs end up replicating. A world with no condition registered skips evaluation entirely and replicates broadcast-to-everyone. A world with conditions pays the evaluation cost even for pairs that resolve to "don't spawn." This is a CPU-time bottleneck, not a bandwidth one, and cadence/staggering is the only lever the engine currently has for it.

Fixing the wrong one does nothing: pacing spawns doesn't lower steady-state egress, and shrinking a delta's encoding doesn't stop a burst from landing in one tick.

Reading the cost you have

The bandwidth counters (`TotalBytesSent`, `TotalBytesReceived`, `TotalStatePacketPayloadBitsSent`, `TotalStatePacketPayloadBitsReceived`) and the per-object/state-only stat provider rows are how you read what a world is actually costing, before deciding which lever to pull. Every one of these figures covers the whole peer — total traffic for that connection, not a single object's line item — so a per-object number is a derived average, not something the wire counts directly. Divide by object count yourself, or use the shipped per-object stat provider row, which already does it. How those counters are wired up and sampled is covered under Diagnostics; this page is about which lever their numbers point you toward.

Burst versus steady state

A spike on a join and a steady per-tick overspend are different failures with different fixes. A join spike is bounded and temporary: it costs frames and, if unpaced, risks a lost-segment discard on a wide tick — the fix is `MaximumSpawnsPerTick`, and it doesn't touch what the world costs once everyone has finished joining. A steady overspend is what egress and resolution-pass measurements are for, and its fix is spawn/stop culling or interval/compression tuning, not pacing.

Redundancy is why a lost tick during a burst is worse than it looks. A retained set is resent `EffectiveGlobalRedundancy` times on the ticks that follow, so bytes a burst already spent get paid again on top of that window's ordinary traffic. A burst that loses a segment doesn't just retry once — it pays the full width of the burst again, redundancy times over, while steady-state traffic is running alongside it. This is the concrete reason pacing a burst is worth doing even though it doesn't move the steady-state number: a wide, unpaced tick multiplies its own cost on loss, a paced one doesn't.

What the engine does not have

Don't design a world around any of these — none of them exist:

- **No per-connection byte-budget scheduler.** There's no cap on how many bytes a peer's connection may spend per tick, and no priority or staleness scheduling deciding what gets sent first when a peer is behind. The spatial grid and per-connection spawn pacing exist; a bandwidth cap does not.
- **No per-object send priority.** Nothing lets you mark one object as more urgent than another when a peer is bandwidth-constrained. Every eligible object is served the same way.
- **No interest enter/exit callbacks.** Game code isn't notified when a peer gains or loses interest in an object. If you need to react to that transition, you have to infer it yourself.
- **No broadphase seam for interest.** There's no hook letting an integration hand the interest pass a pre-narrowed candidate set; evaluation is the O(systems ? connections) pass described above for every world. The spatial grid narrowing that exists today is internal to distance/grid conditions, not an integration seam.

Edition effects on the budget

Several of the levers above only exist in Pro: distance and spatial-grid interest culling (the conditions that drive `InterestEffect.Spawn`/`Stop`), level-of-detail interest, and per-connection spawn pacing (`MaximumSpawnsPerTick`) are all Pro-only. Free still has the four-lever framework and the broadcast-to-everyone default, but the conditions that make spawn/stop culling and burst pacing possible are Pro.

Compression is the other edition split. Free writes every value at its exact length; Pro adds bit-packing brackets that pack values more tightly. This is the one gate that forks the wire format itself — a Free and Pro peer serialize differently at this layer — so it is not something you can toggle per-world; it is fixed by which edition built the peer. This page states that the brackets exist and are Pro-gated; how the encoding works is out of scope here.

A worked budget for a small world

Say a world has 50 objects, each with one `NetworkMember` sending a full-precision value every tick at `SendInterval.Normal`, replicated to 20 connections.

Steady-state egress, ignoring redundancy and headers: 50 objects ? 20 connections = 1,000 per-peer-per-tick sends. If each delta costs roughly the same regardless of which peer receives it, the total is 20 connections' worth of "50 objects' deltas per tick" — the per-connection egress is what scales with object count, not the total across all connections, because each connection only ever carries its own 50-object slice.

Resolution-pass cost, if you add one distance condition: 50 systems ? 20 connections = 1,000 pair evaluations every `EvaluationCadenceTicks` ticks. At a cadence of 1 that's 1,000 evaluations/tick; staggering it across a cadence of, say, 10 spreads that to roughly 100 evaluations on any given tick, at the cost of resolving each pair up to 10 ticks stale.

Burst size on a join: a joining peer is registered against every started system at once — all 50 objects' priming fulls land on one combined stream unless spawn pacing caps it. At the default `MaximumSpawnsPerTick` of 500, a 50-object world never touches the ceiling; the burst is whatever 50 fulls cost, in one tick, with no pacing needed. That ceiling only starts mattering once system count times joining-peer count starts approaching it — this world is nowhere close.

The arithmetic is the budget: egress scales with objects ? connections that can see them, the resolution pass scales with objects ? connections regardless of visibility, and burst size scales with objects per joining peer against the pacing ceiling. Three different multiplications, three different levers, and a 50-object/20-connection world doesn't need any of them pulled yet.

Clipping shots at walls

Driving the core API directly? See Walls are not in the history.

Walls are judged in the present

Lag compensation records hitboxes, not level geometry, so a rewound query sees no walls. A shot is therefore judged in two steps on the server: clip it against the present-time scene to find how far it can travel, then pass that distance as the range of the lag-compensated query, so the past is judged only within what is left.

`LagCompensationWalls` is the clip:

```
float range = LagCompensationWalls.GetDistance(shot.Origin.ToUnity(), shot.Direction.ToUni
ty(), maximumRange, wallLayerMask);

bool isHit = lagCompensationManager.TryRaycast(shooterConnection, shot.View, shot.Origin,
shot.Direction, range, hitboxLayerMask, out LagCompensatedHit lagCompensatedHit);
```

It returns the distance from the origin to the nearest wall along the shot, or the full range when no wall is in reach. Pass it as the range of `TryRaycast` or any other 3D query from the same origin, and `GetDistance2D`'s result to `TryRaycast2D` or any other 2D query.

Why not a plain Physics.Raycast

A plain `Physics.Raycast` stops on the colliders of networked objects too. An automatic `NetworkHitbox` needs a live collider beside it, so the raycast stops on the target's present-time collider. When the target runs toward the shooter, that collider stands nearer than its rewound hitbox, and a clean hit is judged a miss. A ray from a third-person camera stops on the shooter's own character controller at almost no distance at all.

`LagCompensationWalls` reports the nearest collider on the wall layers that belongs to no networked hitbox object. A collider is passed over when its GameObject has a `NetworkHitbox`, or when it sits on or under a GameObject carrying a `NetworkHitboxRoot`, enabled or not. A disabled root's colliders are passed over too, so making an object unhittable never turns it into cover.

Triggers never stop a shot

A trigger never stops a shot, whatever `Physics.queriesHitTriggers` and `Physics2D.queriesHitTriggers` say: the casts never report one. A shooter standing inside a pickup or checkpoint volume still fires, and a trigger lying flush with a wall never carries a shot past it. `GetDistance2D` turns `Physics2D.queriesHitTriggers` off for its casts and puts it back before it returns.

A direction it cannot follow returns zero

The direction does not need to be of unit length. A direction with no length, or one too long to measure (infinite or not a number), returns zero, which stops the shot where it starts, so a client can never clear a shot through walls by sending a direction the check cannot follow.

The 2D form

`GetDistance2D(Vector2 origin, Vector2 direction, float maximumDistance, LayerMask wallLayerMask)` does the same in the default 2D physics world. A 2D shot fired from the shooter's own center starts inside its own collider, which a 2D raycast hits at distance zero while `Physics2D.queriesStartInColliders` is on. That collider belongs to a networked hitbox object, so it

is passed over like any other, and the shot is still clipped only by a wall. `GetDistance2D` keeps `queriesStartInColliders` for its first cast, turns it off for the casts after, and puts it back before it returns. A wall the shot itself starts inside stops it at distance zero while the setting is on, as an ordinary 2D raycast would.

Scenes in their own physics world

`GetDistance(origin, direction, maximumDistance, wallLayerMask)` looks for walls in `Physics.defaultPhysicsScene`, which holds none of the walls of a scene loaded into its own local physics world. A networked scene loads that way when `UnitySceneLoader.LocalPhysicsEnabled` is on, which the scene manager's stacked-scene simulation setting turns on; see Local physics scenes. Where that can happen, pass the shooter's physics world first:

```
float range = LagCompensationWalls.GetDistance(shooterScene.GetPhysicsScene(), shot.Origin
.ToUnity(), shot.Direction.ToUnity(), maximumRange, wallLayerMask);
```

Passing it is always safe, because for a scene without a world of its own that is the default world. `GetDistance2D` always looks in the default 2D world, because Nucleus only gives a scene a local world for 3D physics.

How it walks past hitbox colliders

Nothing is kept between calls and nothing is allocated. Rather than gathering every hit into a buffer, each method casts again from just past a collider it passes over, a thousandth of a unit past it or a millionth of the distance travelled, whichever is longer. Call it on the main thread, as with every Unity physics query; a message handler in a Unity session already runs there, because Unity drives the network loop from `Update` and `LateUpdate`.

That step is a known limit: a wall whose near face lies within it, behind the surface of a hitbox object's collider being passed over, is not seen, because the next cast starts past that face and a raycast does not report a collider it starts inside.

Firing and judging a shot

Driving the core API directly? See Capturing a shot and Judging a shot with rewound queries.

The whole flow in one script

A shot has two halves. The client captures it on the frame it fires and sends it; the server clips it at walls, judges it against the past, and applies the result. One script can hold both, because only the server ever runs the second half:

```
using Nucleus.Connections;
using Nucleus.Integrations.Unity;
using Nucleus.Integrations.Unity.LagCompensation;
using Nucleus.Integrations.Unity.Serializers;
using Nucleus.Managers.Core;
using Nucleus.Managers.LagCompensation;
using Nucleus.Managers.Messages;
using Nucleus.Serializers;
using Nucleus.Systems;
using Nucleus.Transports;
using UnityEngine;

[NetworkType]
public struct ShotMessage : IMessage
{
    public LagCompensatedShot Shot;
}

public class Shooter : MonoBehaviour
{
    [SerializeField]
    private float _range = 100f;
    [SerializeField]
    private LayerMask _hitboxLayers = ~0;
    [SerializeField]
    private LayerMask _wallLayers = ~0;

    private CoreManager _coreManager;
    // Set by your own player code when this peer's avatar spawns, so a host's own shots s
kip it.
    private NetworkSystem _ownAvatarSystem;

    private void Start()
    {
        _coreManager = NucleusUnity.BoundCoreManager;
        _coreManager.MessageManager.RegisterMessageHandler<ShotMessage>(OnShotReceived);
    }

    private void OnDestroy()
    {
        _coreManager?.MessageManager.UnregisterMessageHandler<ShotMessage>(OnShotReceived)
;
    }

    // Call from your input code on the frame the player fires.
    public void Fire(Ray ray)
    {
        LagCompensatedShot shot = _coreManager.LagCompensationManager.CaptureShot(ray.orig
in.ToNative(), ray.direction.ToNative());

        // A host judges its own shot at once. Its view is the present, and a null shooter
 skips nothing, so it names its avatar.
        if (_coreManager.TransportManager.IsServerStarted)
        {
            Judge(shooterConnection: null, _ownAvatarSystem, shot);
            return;
        }
```

```
        if (_coreManager.TransportManager.TryGetServerConnection(out Connection serverConn
ection))
            serverConnection.SendMessage(Channel.Reliable, new ShotMessage { Shot = shot }
);
    }

    private void OnShotReceived(in MessageContext messageContext, ShotMessage shotMessage)
    {
        if (_coreManager.TransportManager.IsServerStarted)
            Judge(messageContext.SenderConnection, ignoredNetworkSystem: null, shotMessage
.Shot);
    }

    private void Judge(Connection shooterConnection, NetworkSystem ignoredNetworkSystem, L
agCompensatedShot shot)
    {
        // Walls are not in the history, so clip the shot against the present scene first.
        float range = LagCompensationWalls.GetDistance(shot.Origin.ToUnity(), shot.Directi
on.ToUnity(), _range, _wallLayers);

        if (!_coreManager.LagCompensationManager.TryRaycast(shooterConnection, ignoredNetw
orkSystem, shot.View, shot.Origin, shot.Direction, range, (uint)_hitboxLayers.value, out L
agCompensatedHit lagCompensatedHit))
            return;

        if (NetworkHitboxRoot.TryGetHitbox(lagCompensatedHit, out NetworkHitbox networkHit
box))
            Debug.Log($"Hit [{networkHitbox.name}] at [{lagCompensatedHit.Point.ToUnity()}
].");
    }
}
```

The rest of this page takes each step in turn.

Capture on the click's own frame

`CaptureShot(origin, direction)` takes `System.Numerics` vectors, so convert with `ToNative()` from `Nucleus.Integrations.Unity.Serializers`. It returns a `LagCompensatedShot` carrying the ray unchanged and the view on screen right now, taken in the same call so the two can never come from different frames. The direction does not need to be of unit length.

Call it from the frame the click happens, never from code that builds the next tick's client state. A tick spans several rendered frames, each drawing a different blend between two server ticks, and the view records which blend was on screen.

Send the shot whole

Put the `LagCompensatedShot` itself in the message, as above, or in a remote call argument or a client state member. Never copy its origin and direction into plain `Vector3` fields: a message rounds each component to a thousandth, which can turn a long shot by about 9 cm at 100 m and lose a headshot with no error. A `LagCompensatedShot` sends each component as its raw 32 bits; an infinity or NaN arrives as zero.

A shot that is not a ray, such as a melee swing whose volume the server builds itself, sends just a view: `CaptureView()` returns the `LagCompensationView` on screen, and the server passes it to an overlap query.

Judge on the server

The server passes the shot's `View`, `Origin` and `Direction` straight into a query. Pass the sender's `Connection` as the shooter: its own objects are skipped, so a shot never hits its shooter, and its view is checked against what the server could have shown it. Pass `null` for a shot the server fires itself.

The 3D queries are `TryRaycast` and `RaycastAll` for rays, `TrySphereCast` and `SphereCastAll` for a thick shot swept along a ray, and `OverlapSphere`, `OverlapCapsule` and `OverlapBox` for explosions, melee swings and area attacks. The `All` forms and the overlaps fill a `List<LagCompensatedHit>` you pass in, nearest first; keep one list and reuse it so nothing is allocated. Every test is exact. The core API page lists each one and what its hit's `Distance`, `Point` and `Normal` mean.

A query's layer mask is a `uint`, one bit per layer; a `LayerMask`'s `value` cast to `uint` is exactly that.

Ignoring an object

Every query has an overload taking a `NetworkSystem ignoredNetworkSystem` straight after the shooter, whose hitboxes it skips. Pass the root of the object to ignore, normally the shooter's own root object, or `null` to ignore nothing. A host firing for its own avatar, which judges with a `null` shooter, passes its avatar, and a turret or NPC the server fires for passes itself. Everything nested under the root is skipped too: naming any member of a grouped object skips the whole object, and every object parented under it with `SystemManager.EnsureSetSystemParent`, such as a carried weapon or shield, is skipped with it at any depth. Naming a nested object instead skips only it and what is under it.

Reading the hit

`LagCompensatedHit` carries the target's `NetworkSystem` (`null` when it has stopped since the judged tick), its `SystemId`, the `HitboxIndex` within its root, the hitbox's `Layer`, and the first contact's `Distance`, `Point` and `Normal`. `NetworkHitboxRoot.TryGetHitbox(lagCompensatedHit, out NetworkHitbox networkHitbox)` leads back to the hitbox's GameObject, where per-region data such as a headshot multiplier can live; see Making an object hittable.

A shot that starts inside a hitbox

With **Queries Start In Hitboxes** on, the default, a ray whose origin lies inside a hitbox, or a sphere or circle cast that starts touching one, hits it at distance zero with its own origin as the point. That is the right answer for a shot fired point-blank. Turn it off on the lag compensation manager when a muzzle can sit inside some other hitbox and the shot should carry on to what lies beyond. Overlaps are not affected.

2D games

A 2D game captures with a Z of zero and judges with the 2D queries, which take `System.Numerics.Vector2` and run in the plane Z = 0:

```
LagCompensatedShot shot = _coreManager.LagCompensationManager.CaptureShot(new(origin.ToNat
ive(), 0f), new(direction.ToNative(), 0f));

float range = LagCompensationWalls.GetDistance2D(shot.Origin2D.ToUnity(), shot.Direction2D
.ToUnity(), _range, _wallLayers);

bool isHit = _coreManager.LagCompensationManager.TryRaycast2D(shooterConnection, ignoredNe
tworkSystem, shot.View, shot.Origin2D, shot.Direction2D, range, (uint)_hitboxLayers.value,
 out LagCompensatedHit lagCompensatedHit);
```

`TryRaycast2D` and `RaycastAll2D` mirror the 3D raycasts. `TryCircleCast2D` and `CircleCastAll2D` mirror the sphere casts, with their arguments in `Physics2D.CircleCast`'s order. `OverlapCircle`,

`OverlapCapsule2D` and `OverlapBox2D` mirror the overlaps; `OverlapBox2D` turns its rectangle by an angle in radians, so multiply a Unity angle by `Mathf.Deg2Rad`. A hit on a 2D hitbox has a `Point` and `Normal` with a Z of zero.

A 3D query never touches a 2D hitbox, so `TryRaycast` in a 2D game finds nothing. A 2D query touches only 2D hitboxes unless **2D Queries Hit 3D Hitboxes** is ticked.

Common mistakes

- **Judging on the client.** The client fires and sends; only the server decides.
- **Expecting `Physics.Raycast` to hit rewound targets.** Nothing moves; only the lag-compensated queries see the past.
- **Clipping at walls with a plain `Physics.Raycast`.** It stops on the target's present-time collider and on the shooter's own controller. Use `LagCompensationWalls`.
- **Capturing the view and the ray separately.** They can land on different frames. `CaptureShot` takes both in one call.

Worked example: the HitscanRewind demo

What the demo shows

The `LagCompensation/HitscanRewind` demo is one idea: the server judges a client's shot against the targets where that client saw them, not where they are by the time the shot arrives. Open `Scenes/HitscanRewindDemo.unity` in two editors (a ParrelSync clone or an MPPM virtual player becomes the client), play the main editor first, then click a moving target in the client's Game view. The demo folder's `README.html` walks through every step; this page reads the code as a template for your own game.

Three blue targets, each a capsule body with a sphere head, sweep side to side across the camera's view, fast enough mid-swing that the delay between server and client puts what you see well behind where they really are. The transport adds 75 milliseconds to every packet each way, so even two editors on one machine see the effect. There are no walls, so every miss is a miss in time rather than a miss behind cover.

The targets: hitboxes copied from colliders

Each target prefab carries a `NetworkSystemObject` and a `NetworkHitboxRoot`, and its body and head each carry a `NetworkHitbox` left on `Automatic`, which copies the shape from the capsule and sphere colliders beside it. Nothing else is needed for the server to record them. Select a part while playing and the hitbox draws the exact shape the server records, in orange, over the collider's own outline.

The colliders stay on the targets. The client raycasts them to find its own hit for the on-screen comparison; the server never touches them, and tests its recorded hitboxes instead.

The message: the shot travels whole

`HitscanShotMessage` carries the shot as a `LagCompensatedShot`, so its ray crosses the wire exactly, plus a few demo-only fields the comparison uses:

```
[NetworkType]
public struct HitscanShotMessage : IMessage
{
    public bool IsAuthenticationRequired => true;

    public LagCompensatedShot Shot;
    public uint ShotNumber;
    public bool IsClientHit;
    public uint ClientTargetSystemId;
    public HitscanExactVector3 ClientHitPoint;
}
```

The client half: two lines

`HitscanShooter.Fire` runs on the frame the click happens. It captures the shot and sends it:

```
LagCompensatedShot lagCompensatedShot = _coreManager.LagCompensationManager.CaptureShot(ra
y.origin.ToNative(), ray.direction.ToNative());

serverConnection.SendMessage(Channel.Reliable, hitscanShotMessage);
```

On a server, the same method skips the message and judges the shot at once with a `null` shooter. A server sees the world as it is, so its own shots are judged against the present.

The server half: one query

`HitscanShooter.Judge` is the whole server side:

```
LagCompensationManager lagCompensationManager = _coreManager.LagCompensationManager;
LagCompensatedShot lagCompensatedShot = hitscanShotMessage.Shot;
uint layerMask = (uint)_hitboxLayers.value;

bool isRewoundHit = lagCompensationManager.TryRaycast(shooterConnection, lagCompensatedSho
t.View, lagCompensatedShot.Origin, lagCompensatedShot.Direction, _range, layerMask, out La
gCompensatedHit lagCompensatedHit);
bool isPresentHit = lagCompensationManager.TryRaycast(shooterConnection, default, lagCompe
nsatedShot.Origin, lagCompensatedShot.Direction, _range, layerMask, out _);
```

The first call is the one a game needs: it rebuilds every hitbox where the shooter saw it and returns the nearest hit, with its system, hitbox index, point and normal. The second call passes an empty view, `default`, which means the present, only so the readout can count **Hits that only rewinding made**: shots that hit the rewound target but would have missed the present one. A real game makes only the first call.

The hit's `SystemId` leads to the target that records it, and the verdict goes back to the shooter in a reliable message.

The comparison: how close the rebuild lands

Every hit leaves a grey copy of the target where the client saw it, with a cyan sphere where the client's own raycast hit and a yellow sphere where the server's rewound raycast hit. The client sends its own hit point with the shot, and when both hit the same target the server answers with the difference between the two points. They usually land within a few hundredths of a metre: the server has rebuilt the target where the client's screen showed it.

The orbiting spheres: what a hitbox needs from replication

Between the first and second targets, two spheres orbit and swap between a close and a far orbit together. The left one carries its own `NetworkTransform`; the right one is posed by a `NetworkAnimator` with **Precise Playback** on, so the client poses its animator at the playback position the server sent. Both land where the server had them, to within a millimetre, so shots at either are judged against what the client saw. With Precise Playback off, the client runs its own copy of the animation from replicated parameters, a few centimetres from the server's pose: fine for looks, not for hitboxes. A hitbox is only as accurate as the replication that places the thing the player aims at.

What to add in a real game

- **Walls.** The demo has none. A game with cover clips the shot first with `LagCompensationWalls.GetDistance` and passes the result as the query's range.
- **Immediate feedback.** The demo's marker waits for the verdict, a round trip away. A game plays its own effect at once and lets the verdict confirm the damage.
- **Per-region damage.** Read a component on the hit hitbox's GameObject through `NetworkHitboxRoot.TryGetHitbox`; see Making an object hittable.
- **A rewind window that fits.** The demo keeps the default 500 milliseconds; see UnityLagCompensationManager.

How lag compensation works

> **Driving the core API directly?** See How lag compensation works.

A client aims at the past

A client draws the world a little behind the server: it holds received state back for interpolation, and every packet already spent half a round trip reaching it. When the player clicks a moving target, the server has already moved that target on, and by the time the shot arrives it has moved again. Judge the shot against the server's present and a player who aimed perfectly misses whatever is moving.

Lag compensation lets the server judge the shot against the world as the shooter saw it. It is a Pro feature. In a free build every name below still exists, so the same scripts compile, but nothing is recorded and every query finds nothing.

Colliders are never moved

The server does not rewind transforms, colliders or Unity's physics. It keeps its own history of **hitboxes** and does its own geometry:

> **Record.** A `NetworkHitboxRoot` on a networked object registers every `NetworkHitbox` under it with the server's lag compensation manager. Once per tick, after that tick's state is written, the server records where each hitbox was. On Unity this runs in `LateUpdate`, after animation has posed the bones, so an animated rig is recorded as it was drawn.
>
> **Capture.** On the frame the player fires, the client calls `CaptureShot`, which returns a `LagCompensatedShot`: the ray exactly as drawn, and a view naming the server tick on screen and how far the screen had blended toward the next one. The client sends it to the server.
>
> **Judge.** The server passes the shot to a query such as `TryRaycast`, which rebuilds every hitbox where the shooter saw it, blended exactly as the shooter's screen blended them, and tests the shot against those shapes.

Because only the history is read, `Physics.Raycast` never sees the past. A hitbox is not a collider and the physics engine never sees it; only the lag-compensated queries do.

What the server takes care of

- **Blending.** A view between two ticks rebuilds each hitbox between both ticks' poses. The cheap test that rules whole objects out covers every hitbox at every recorded pose, active or not, so a blended hitbox is never culled.
- **Interval targets.** A target whose `NetworkTransform` position is sent on an interval is drawn later by clients, and the server judges it that much earlier by itself.
- **Grouped objects.** A grouped object's hitboxes are recorded once, under one member, so a shot hits each hitbox once.
- **Stopped targets.** A target that despawns stays hittable until its last tick leaves the rewind window, so a shot fired just before it went away still lands.
- **The shooter.** A shot never hits an object its shooter controls.
- **Hosts.** A host's own view is the present, so its own shots are judged against the present.

The pages that follow

- Making an object hittable: `NetworkHitboxRoot`, `NetworkHitbox`, their inspectors, 2D hitboxes and

making an object unhittable for a while.

- Firing and judging a shot: capturing and sending a shot, every rewound query, ignoring an object and reading a hit.
- Clipping shots at walls: `LagCompensationWalls`, because walls are not in the history.
- UnityLagCompensationManager: the rewind window, the 2D and start-inside settings, the spawn compensation cap, and the clamp violation.
- Worked example: the HitscanRewind demo.

Lag Compensation

Lag compensation is a Pro feature. Pages in this category:

- How lag compensation works
- Making an object hittable
- Firing and judging a shot
- Clipping shots at walls
- UnityLagCompensationManager
- Worked example: the HitscanRewind demo

Making an object hittable

Driving the core API directly? See Registering hitboxes.

Two components

A networked object becomes hittable with two components:

- **`NetworkHitboxRoot`**, beside the object's `NetworkSystemObject`. It gathers every `NetworkHitbox` on its GameObject and children and registers them with the server's lag compensation manager.
- **`NetworkHitbox`**, on each part a shot should hit: a body, a head, a limb. Each one is a single sphere, box or capsule, or for a 2D game a circle, 2D box or 2D capsule, whose world pose the server records every tick. It is not a collider, and the physics engine never sees it.

The root registers when the object's system links and lets go when it unlinks, so a pooled object registers afresh on every life. Only a server records: the root does the same thing on every peer, and on a client the registration is refused quietly and the component does nothing. A root without a `NetworkSystemObject` on the same GameObject logs an error and disables itself.

Where a hitbox takes its shape from

`NetworkHitbox`'s **Setup Mode** is a `HitboxSetupMode`:

- **`Automatic`**, the default, copies the shape from a collider on the same GameObject when the hitbox wakes. It looks for a `SphereCollider`, `BoxCollider`, `CapsuleCollider`, `CircleCollider2D`, `BoxCollider2D` or `CapsuleCollider2D`, in that order, trigger or not, so a hitbox added beside an existing collider matches it with no further setup. It writes what it copied into its own settings, so switching to `Manual` starts from the collider's values.
- **`Manual`** uses the shape typed into the hitbox itself, for a part with no collider, or a hitbox that should differ from the collider it has.

Any other collider, a `MeshCollider` or `PolygonCollider2D` for one, has no exact test and is refused with an error rather than approximated, and so is a `BoxCollider2D` with an edge radius, whose rounded corners have none either. An automatic hitbox with nothing it can copy shows a warning in its inspector and is left out of the root.

A manual hitbox has a **Shape** (`HitboxShape`), a **Center** in local space before scale, and the fields its shape uses: **Radius**, **Size** and **Height** (the full height, both rounded ends included). A capsule's **Direction** is a `HitboxAxis`, `X`, `Y` (the default) or `Z`, whose values match `CapsuleCollider.direction`, so a collider's direction converts by a plain cast. A 2D capsule runs along `X` for a horizontal capsule or `Y` for a vertical one, and its inspector offers just those two.

Layers and per-region rules

A hitbox's layer is its GameObject's layer. A query's layer mask chooses which layers a shot can hit, so put each hitbox's GameObject on the layer your queries should filter by.

For per-region rules such as a headshot multiplier, put your own component on the hitbox's GameObject and read it from a hit, rather than spending a layer on each region:

```
if (NetworkHitboxRoot.TryGetHitbox(lagCompensatedHit, out NetworkHitbox networkHitbox) &&
networkHitbox.TryGetComponent(out DamageZone damageZone))
    damage *= damageZone.Multiplier;
```

`NetworkHitboxRoot.TryGetHitbox(in LagCompensatedHit, out NetworkHitbox)` follows the

hit's system to the networked object linked to it, and its `HitboxIndex` to the hitbox. A target whose system has stopped since the judged tick is still hit, but its hit names no system, so it cannot be followed back to a GameObject; `TryGetHitbox` returns `false`, and `lagCompensatedHit.SystemId` still names the target. A root also has an instance `TryGetHitbox(int hitboxIndex, out NetworkHitbox)` for an index you already hold.

Scale and offsets follow Unity's colliders

A 3D hitbox's scale is its world scale, `Transform.lossyScale`, the product of every scale above it, which is what Unity's own colliders use, so a hitbox and its collider agree at every depth. The center offset goes through `Transform.TransformPoint`, so it is scaled and rotated by the whole parent chain too. Scale is recorded every tick, so an object that grows or shrinks is judged at the size it had on the viewed tick. The one case neither a collider nor a hitbox represents exactly is a non-uniformly scaled parent over a rotated child, which skews the child.

2D hitboxes

A `Circle`, `Box2D` or `Capsule2D` hitbox is recorded in the plane Z = 0, whatever depth its object sits at, so sorting depth never moves it out of a shot. It reads its transform the way Unity's 2D colliders do: a half turn about Y, a sprite flip, mirrors it, and a tilt about its own X or Y axis foreshortens a box or capsule, exactly as the collider does. A circle stays round and takes the larger of its world X and Y scales. A tilt about any other axis makes the collider a parallelogram, which the hitbox follows as the rectangle along the flattened X axis.

2D hitboxes are judged only by the 2D queries, and 3D hitboxes only by the 3D ones, unless **2D Queries Hit 3D Hitboxes** is ticked on the lag compensation manager.

Seeing what will be recorded

A selected `NetworkHitbox` draws the exact shape the server will record, in orange, scaled exactly as it will be, so it can be checked against the collider's own outline. A selected `NetworkHitboxRoot` draws every hitbox it gathers, inactive ones included, so a whole rig can be checked at once.

The `NetworkHitbox` inspector shows the setup mode, then either the shape fields or a readout of the shape copied from the collider. It warns when no collider it can copy is found, and when no `NetworkHitboxRoot` is on its GameObject or above it, since such a hitbox is never recorded.

The hitbox root inspector

In edit mode the `NetworkHitboxRoot` inspector does three things.

Add Hitboxes From Colliders adds an automatic `NetworkHitbox` beside every solid collider under the root that a hitbox can copy and that has no hitbox yet, inactive GameObjects included. The same command is in the root's context menu. It skips a trigger, which a shot passes through; a collider whose GameObject already has a hitbox, since a GameObject holds one; a collider no hitbox can copy; and a collider that is not the one an automatic hitbox on that GameObject would copy first. The Console lists what was added and what was skipped and why, and one undo takes back everything a click added.

The hitbox table lists every hitbox the root will find, in the order it gathers them, with its GameObject, shape, layer, whether it is 2D or 3D, and whether it is usable. The # column is the `HitboxIndex` a hit reports. A hitbox with no usable shape takes no index, so every hitbox after it reports an index one lower than its place in the list, as the column shows. Click a name to find it, and double-click to select it.

Warnings flag a root that would record nothing: no `NetworkSystemObject` on its GameObject, no `NetworkHitbox` found, no hitbox with a usable shape, or more than 64 usable hitboxes, the most one root can hold. It also lists any unusable hitboxes, and notes a root that mixes 2D and 3D hitboxes. **Nucleus >**

Validate Network Prefabs logs the same errors for every prefab, using the same rules.

While playing, the inspector shows how many hitboxes were gathered and whether the server is recording the root (`IsRecording`).

The hitboxes are gathered once

A root gathers its hitboxes once, when the object wakes, inactive ones included. Add every hitbox to the prefab rather than at runtime. A hitbox that is disabled, or whose GameObject is inactive, on a given tick is recorded but never hit on that tick.

Making an object unhittable for a while

Disable the root to make the whole object unhittable, for a downed, spawn-protected or ragdolled player:

```
networkHitboxRoot.enabled = false;
```

A disabled root reports every hitbox inactive, so it is never hit. It keeps recording while disabled, writing every pose as usual, so a shot judged at a tick it was disabled on misses it however late the shot arrives, and it is hittable again from the first tick recorded after `enabled = true`. Neither component shows an enable checkbox in the inspector, so do it from code, on the server. Its colliders stay in the scene throughout, and `LagCompensationWalls` passes over them either way.

Grouped objects record once

A grouped object links several systems to one GameObject, but its hitboxes belong to the object, so the root records them once, under one member, and a shot hits each hitbox once. It prefers the member carrying the object's `TransformComponent`, because a target whose position arrives on an interval is judged where its transform was drawn, which only that member's history knows. Until that member links, the first member to link stands in, and nothing it recorded is lost. A member that leaves on its own hands the recording to one that remains.

UnityLagCompensationManager

Driving the core API directly? See The rewind window and the view clamp.

The inspector seat for lag compensation

`UnityLagCompensationManager` is the inspector-side counterpart of the engine's `LagCompensationManager`. It is added to the `UnityCoreManager`'s GameObject automatically when a scene does not carry one, so a scene only needs it placed to change the defaults. Every value is pushed onto the engine once, in `ManagersInstantiated`, when the session starts, so the inspector disables the fields during play. After that, `NucleusLagCompensationManager` is the engine manager they were pushed onto, the same object as `NucleusUnity.BoundCoreManager.LagCompensationManager`, and code can change its fields directly.

Lag compensation and spawn compensation are Pro. In a free build every value is stored and never read. The inspector says so in its header.

Lag Compensation

Inspector field	Engine setting	Default	What it does
Max Rewind (ms)	`MaximumRewindMilliseconds`	500	How far back hitbox poses are kept and a shot may be judged. Read when an object's hitboxes register, so it applies to objects spawned after it changes. The slider runs to 2000; a longer window is legal from code but already far past any playable link.
2D Queries Hit 3D Hitboxes	`Can2DQueriesHit3DHitboxes`	off	Lets 2D queries also hit 3D hitboxes, for a game that mixes the two. 3D queries never hit 2D hitboxes either way.
Queries Start In Hitboxes	`CanQueriesStartInHitboxes`	on	Lets a ray or cast hit a hitbox it starts inside, at distance zero, the way Unity's 2D queries hit a collider they start in. When off, that hitbox is ignored and the query goes on to what lies beyond it. Overlaps are not affected.

With **2D Queries Hit 3D Hitboxes** ticked, a 2D ray and `OverlapBox2D` touch a 3D hitbox where the plane Z = 0 cuts it, and a circle, a 2D capsule and a circle cast reach it as a ball or capsule of their radius lying in the plane.

Turn **Queries Start In Hitboxes** off when a muzzle can sit inside a hitbox the shot should pass through, such as another body part of a large creature, and the shot should carry on to whatever lies beyond.

Spawn Compensation

Inspector field	Engine setting	Default	What it does
Max Spawn Compensation (ms)	`MaximumSpawnCompensationMilliseconds`	150	The most a freshly spawned object may be caught up through the latency it lost, counted across every hop it travelled. The slider runs from 0 to the engine ceiling, 500, so the inspector cannot author a value the runtime would clamp.

Spawn compensation is the other half of lag compensation: lag compensation judges a shot against the past, and spawn compensation brings a fresh object, such as a projectile, up to the present. A client watching somebody else's shot is owed what the server already spent plus what the delivery cost it, and the two together are held to this one figure. It also bounds how long a catch-up runs. Raise it to watch the catch-up; on a local link the measured latency never asks for the whole of it. See Spawn compensation.

The view clamp violation

A shooter whose view is older than **Max Rewind (ms)**, or newer than anything the server recorded, is reporting a moment the server could never have shown it. The query clamps the view into range and still judges the shot, so a player on a very slow link still gets results, and raises `LagCompensationViewClampedViolation` against the shooter, carrying the `ReportedTick` and the `ClampedTick` the shot was judged at. It defaults to `ViolationAction.Ignore`.

There is no inspector for violations. Subscribe from code, after the manager root has woken:

```
private CoreManager _coreManager;

private void Start()
{
    _coreManager = NucleusUnity.BoundCoreManager;
    _coreManager.ViolationManager.LagCompensationViewClampedViolationDetected += OnLagComp
ensationViewClamped;
}

private void OnDestroy()
{
    if (_coreManager is not null)
        _coreManager.ViolationManager.LagCompensationViewClampedViolationDetected -= OnLag
CompensationViewClamped;
}

private void OnLagCompensationViewClamped(in ViolationContext<LagCompensationViewClampedVi
olation> violationContext)
{
    LagCompensationViewClampedViolation violation = violationContext.Violation;

    Debug.Log($"Connection [{violationContext.Connection.Id}] reported tick [{violation.Re
portedTick}], judged at [{violation.ClampedTick}].");
}
```

A single clamp is ordinary on a link that spiked. A steady stream from one client is either a very slow link or a client asking to be judged against the oldest world it can get. To act on it, register a deciding handler; see Handling violations in Unity.

Registering Call Handlers Across a Component's Lifetime

Driving the core API directly? See Handler Registration and Scope.

The marker's link events

`NetworkSystemObject` is the component that correlates a GameObject with its networked system. It raises two pairs of events as that correlation forms and breaks:

- `SystemLinked` / `SystemUnlinked` fire for each individual system as it links or unlinks. A `NetworkSystemGroup` can put several systems on one GameObject, and these fire once per member.
- `SystemsLinked` / `SystemsUnlinked` fire once for the object's whole set: an ungrouped object is a batch of one and fires on its first link, a grouped object fires when its linked count reaches the group's replicated member count.

For a component that wants to act on the object as a whole, rather than on one member of a group, `SystemsLinked` and `SystemsUnlinked` are the pair to use. There's a further reason `SystemsLinked` in particular is the safer subscription: its `add` accessor checks `IsLinkedBatchComplete()` and, if the object is already complete, invokes the handler immediately on the new subscriber. A component that wakes after the object has already linked doesn't miss the event, it's replayed. `SystemsUnlinked` has no such replay, it's a plain event, so a subscriber that joins after the object has already unlinked hears nothing until the next unlink.

```
private void Awake()
{
    _networkSystemObject = GetComponent<NetworkSystemObject>();

    // Replays at once if the object already linked before this Awake ran.
    _networkSystemObject.SystemsLinked += OnSystemsLinked;
    _networkSystemObject.SystemsUnlinked += OnSystemsUnlinked;
}

private void OnSystemsLinked(NetworkSystemObject networkSystemObject)
{
    if (!networkSystemObject.TryGetFirstSystem(out NetworkSystem system))
        return;

    system.RegisterRpcHandler<BeaconStrikeRpc>(OnBeaconStruck);
}

private void OnSystemsUnlinked(NetworkSystemObject networkSystemObject)
{
    // Reset per-life state here; a pooled, reactivated object never runs Awake again.
}

private void OnDestroy()
{
    if (_networkSystemObject != null)
    {
        _networkSystemObject.SystemsLinked -= OnSystemsLinked;
        _networkSystemObject.SystemsUnlinked -= OnSystemsUnlinked;
    }
}
```

`TryGetFirstSystem` resolves the object's primary system, the canonical member to scope object-wide work against, which is why it's the call a `SystemsLinked` handler reaches for.

The NucleusBehaviourBase alternative

A script deriving from `NucleusBehaviour<TComponent0>` (which itself derives from `NucleusBehaviourBase`) doesn't subscribe to the marker's events by hand. It requires its system on `Awake`, and the base class calls `OnSystemLinked` and `OnSystemUnlinked` for it as that system links and unlinks:

```
public class Beacon : NucleusBehaviour<BeaconComponent>
{
    protected override void OnSystemLinked()
    {
        NetworkSystem.RegisterRpcHandler<BeaconStrikeRpc>(OnBeaconStruck);
    }

    protected override void OnSystemUnlinked()
    {
        // Reset per-life state.
    }
}
```

A call handler belongs in `OnSystemLinked`, not `Awake`. `Awake` runs once per GameObject instantiation, but a pooled instance is reactivated without a second `Awake` running at all, its next life begins from `OnEnable`. `OnSystemLinked`, by contrast, fires every time this behaviour's system links, on the first spawn

and on every respawn after a pool return. Registering there is also naturally scoped to a live `NetworkSystem`: `NetworkSystem` is null before the system links, so a handler registered in `Awake` would have nothing to register against yet on the very first life.

Why NetworkSystem.RegisterRpcHandler, not the RpcManager overload

Both the marker-event and the `NucleusBehaviourBase` examples above register through `NetworkSystem.RegisterRpcHandler<T0>`, not through `RpcManager` directly. `NetworkSystem.RegisterRpcHandler` scopes the handler to that one system: it's invoked only for calls addressed to this system, never for another object's calls of the same type. A handler registered this way skips the branch every unscoped handler otherwise has to open with, checking which object a call was actually meant for. The scope rule itself, and what an unscoped handler on `RpcManager` looks like, is covered on the core API counterpart page linked above.

What the system already handles for you

Once a handler is registered through `NetworkSystem.RegisterRpcHandler`, the system tracks it and takes care of its whole lifecycle from there:

- It deregisters the handler from the manager when the system despawns.
- It re-registers the handler when the system spawns again (a respawn on a pooled instance, or the handler was registered before the system had ever spawned at all).
- It's dropped from tracking only when the instance returns to the pool.

This means `OnDestroy` needs no call to `UnregisterRpcHandler`. The only thing left to unhook in `OnDestroy` is the marker event subscription itself, `SystemsLinked` / `SystemsUnlinked` (or `SystemLinked` / `SystemUnlinked`), when a component subscribed to those by hand instead of deriving from `NucleusBehaviourBase`. A destroyed marker also clears its own events as part of `OnDestroy`, but a component on a separate GameObject that can outlive the marker still owns its own unsubscribe.

Resolving the manager from a plain MonoBehaviour

A script that isn't a `NucleusBehaviourBase` subclass, one that only needs to reach the marker's events, resolves the manager through `NucleusUnity.BoundCoreManager`:

```
CoreManager coreManager = NucleusUnity.BoundCoreManager;

if (coreManager is null)
{
    Debug.LogError("No bound CoreManager; the scene needs a UnityCoreManager.");
    enabled = false;
    return;
}
```

`NucleusUnity.BoundCoreManager` is a plain static property, so with fast enter play mode (no domain reload) it can carry a previous session's value into the next one until `NucleusUnity.Initialize` re-binds it; the reset that guards against this runs at the start of every session. A script that reads it too early in a session, before that reset has run, can find a stale manager left over from the last play session rather than null.

Channels and What Is Guaranteed

Channel.Reliable and Channel.Unreliable

`Channel` is a two-value enum: `Reliable` (ordered and reliable) and `Unreliable`. Every message and every call is sent on one or the other, chosen at the send call:

```
connection.SendMessage(Channel.Reliable, myMessage);
mySystem.TrySendRpc(RpcTarget.Observers, Channel.Unreliable, myRpc);
```

The channel is not a Nucleus feature built on top of the transport - it is a request passed straight through to it. The transport is what makes `Reliable` actually reliable: nothing in `RpcManager`, `MessageManager`, or `Connection` retries a send, buffers it for redelivery, or checks that it arrived. Pick `Reliable` and the transport guarantees order and delivery. Pick `Unreliable` and it guarantees neither.

Calls and messages are not state

A call or a message is deliberately not state. It carries no acknowledgment, no redundancy, and no delta baseline, and it is never resent to reach eventual consistency. It either arrives once, or it does not arrive at all - there is no third outcome where the engine notices a loss and tries again.

That means the channel is the only reliability lever either mechanism has. If a call or a message must arrive, it has to ride `Channel.Reliable`; there is no other way to make that true. Conversely, sending something on `Channel.Unreliable` is an explicit statement that losing it occasionally is fine.

This holds identically for messages and calls. A message has no address beyond the connection it travels to; a call additionally names a `NetworkSystem` so the receiver resolves the object before a handler runs. Neither carries any redelivery machinery, so for both, choosing the channel is most of the decision about whether the send is safe to lose.

Batching and framing

Messages and calls are batched per `Connection` and sent once per frame, one outbound writer per channel. Each one is length-framed: a message's body is written as `PackedBytes` with a known bit count, and a call's envelope (`RpcPacket`) carries the same for its body.

That framing is what keeps one bad payload from taking down the rest of the batch. Decoding runs per message and per call, not around the whole drain, so a body that fails to decode leaves nothing misaligned behind it - the reader already knows where that body ends from its declared length, and every payload packed after it in the same batch still dispatches normally.

A large unreliable send stays unreliable

A message or call larger than the transport's maximum transmission unit keeps the channel you asked for. A reliable one is handed to the transport, which splits it and delivers every piece. An unreliable one is split by the engine into pieces that each fit one datagram, on every transport, and the receiving peer rebuilds it once all of its pieces have arrived.

A split unreliable payload arrives whole or not at all. Losing any one piece loses the payload, so the more pieces it needs, the more often it is lost, and the receiver drops it once it has waited five seconds for a missing piece. An unreliable payload too large to split at all, more than 255 pieces or close to 300 KB at the default transmission unit, is dropped with an error, so send data that large reliably.

When neither one is the right tool

Messages and calls both fire once and are gone - nothing records that they happened for a peer that starts observing afterward. If a value needs to be correct for a late joiner, it belongs in a replicated member instead, which shows a new observer the value as of the last write rather than replaying past sends. See State, a Message, or a Call? and How replicated state works.

Messaging and Remote Calls

Pages in this category:

- Channels and What Is Guaranteed
- Registering Call Handlers Across a Component's Lifetime
- Sending a Remote Call From a Component
- State, a Message, or a Call?
- When a Message or Call Reaches Its Handler
- Typed Messages in Unity
- Messages and Calls That Do Not Arrive

When a Message or Call Reaches Its Handler

Outbound messages and calls leave once per frame

Messages and calls sent by your code do not go on the wire the instant you call the send method. They queue, and the engine sends the queue once per frame, on every frame, at `NetworkLoopSteps.LateSerialize`'s place in the order. On a frame that ticks, the send happens after the `LateSerialize` callbacks, so a message or call sent from a `LateSerialize` callback or any earlier step still leaves this frame. One sent from a `TickAdvance` or `LateVariableUpdate` callback leaves on the next frame. Messages and calls go out in the same send, so a message and a call sent in the same frame leave together.

Inbound calls: LateStateUpdate, deliberately after messages

Inbound calls are dispatched on `NetworkLoopSteps.LateStateUpdate`, after the tick's state has applied. That is later than inbound messages, and on purpose: state, including a spawn that arrived this same tick, has already landed by `LateStateUpdate`. A call addressed to an object that spawned in the same frame resolves against a system that already exists, instead of being stepped over because the object wasn't there yet.

Send order holds across types

A peer's messages reach their handlers in the order they were sent, regardless of how many different message types are mixed in. This isn't per-type ordering — `MessageManager` backs local delivery with one ordered list (`_pendingLocalMessages`) rather than one queue per type, so the drain walks sends in the order they happened, not grouped by type.

Host loopback: Immediate, and only Immediate

A host is one `CoreManager` holding both the server and client roles. When one half addresses a message or call to the other, `TransportManager.HostLoopbackDelivery` decides when that delivery runs. Only `HostLoopbackDelivery.Immediate` is settable from game code: the receiving half's handlers run at the send site, before the send call returns, on whatever loop step the sender happened to be on.

`HostLoopbackDelivery.Deferred` exists in the enum but is refused with a logged reason when a game tries to set it. It's kept only so the engine's own tests can measure what the alternative would cost — handlers running one step later, at `EarlyVariableUpdate` on the following frame, the same step a real remote peer's copy would land on. Deferred delivery is message-only; a call is always dispatched at the send site because its delivery has to run the server's own admission (resolve the system, check the sender's permission, decide relay) inline.

Re-entrancy

Because `Immediate` runs a handler at the send site, a handler that sends something back re-enters the same pass that sent the original message or call. The engine supports this. The one consequence worth knowing: if a handler reached this way starts a spawn, and the sending step is `LateSerialize` or `TickAdvance`, after the tick's state was sent, that spawn is deferred until the tick advances. That is the same deferral any other late spawn gets, not a special case for loopback.

A handler throw costs only its own payload

Both `MessageManager` and `RpcManager` wrap each individual message or call dispatch in its own `try`/

`catch`, not the whole batch. A handler that throws is logged and the payload is dropped; every other message or call in the same drain — from the same sender, in the same frame — still reaches its own handler. A batch-wide guard would have cost a client everything the server sent that frame over one bad payload; the per-payload boundary is why it doesn't.

Typed Messages in Unity

Driving the core API directly? See Typed Messages.

The manager

`UnityCoreManager.EnsureManagers` adds a `UnityMessageManager` to the CoreManager's GameObject alongside the rest of the Unity managers, so it is already present whenever a `UnityCoreManager` bootstraps a session. It exposes one thing, `NucleusMessageManager`, which is the core `MessageManager` this component drives. `UnityMessageManager` carries no inspector settings — its own summary says as much — so there is nothing to configure on it.

Resolving the manager

From a script sitting on the manager GameObject (or a child of it), get the component directly:

```
UnityMessageManager unityMessageManager = GetComponent<UnityMessageManager>();
MessageManager messageManager = unityMessageManager.NucleusMessageManager;
```

From anywhere else, go through the bound CoreManager instead:

```
MessageManager messageManager = NucleusUnity.BoundCoreManager.MessageManager;
```

`NucleusUnity.BoundCoreManager` is the CoreManager the Unity integration bound at startup; it is set once `UnityCoreManager.Awake` calls `NucleusUnity.Initialize` and cleared on teardown.

Registering and unregistering a handler

`MessageManager.RegisterMessageHandler<T0>` and `UnregisterMessageHandler<T0>` take a `MessageReceivedHandler<T0>` delegate. Registration is not scoped to any GameObject or component — nothing despawns it, nothing tears it down when the object that registered it goes away. Pair the two calls by hand:

```
public class DamageAlertListener : MonoBehaviour
{
    private MessageManager _messageManager;

    private void Start()
    {
        _messageManager = NucleusUnity.BoundCoreManager.MessageManager;
        _messageManager.RegisterMessageHandler<DamageAlertMessage>(OnDamageAlert);
    }

    private void OnDestroy()
    {
        _messageManager?.UnregisterMessageHandler<DamageAlertMessage>(OnDamageAlert);
    }

    private void OnDamageAlert(in MessageContext messageContext, DamageAlertMessage messag
e)
    {
        // Handle the message.
    }
}
```

A handler left registered after its component is destroyed keeps firing against a `this` Unity has already torn down. The `MessageManager` reference is cached in `Start` rather than resolved fresh in `OnDestroy`, since

destruction order across objects is not guaranteed and `NucleusUnity.BoundCoreManager` may already be null by then.

Getting a Connection to send to

`SendMessage<T0>` is a member of `Connection`, so sending needs a `Connection` to call it on. From a client script, the connection to the server comes from `TransportManager.TryGetServerConnection`:

```
if (NucleusUnity.BoundCoreManager.TransportManager.TryGetServerConnection(out Connection s
erverConnection))
    serverConnection.SendMessage(Channel.Reliable, new DamageAlertMessage());
```

On the server side, sending to one or many clients uses the server-side connection helpers covered in Sending a Message to Many Peers.

The generator prerequisite

A struct used as an `IMessage` payload needs the Nucleus source generator to see it, the same requirement as any call payload. In a Unity project this means `Nucleus.CodeAnalysis.SourceGenerators.dll` must be marked as a Roslyn analyzer (the `RoslynAnalyzer` label on its `.meta`) in every assembly that declares or sends the message type. Without it, the project still compiles, but the generated wire code for that type is never produced, and the message silently never sends.

Messages and Calls That Do Not Arrive

The message hash could not be found for Type

`Connection.SendMessage<T0>` looks up the type's wire hash before anything else, and if no usable hash comes back it logs `The message hash could not be found for Type [...]` and returns. Nothing is queued.

The hash is worked out from the type's full name, so a sender never has to register a type before sending it. The lookup fails in only two cases:

- **Another message type this peer has already used hashes to the same value.** This is the usual cause. The first time it happens a separate error names both types (see the next section), and from then on every send of either type logs the error above.
- **The type has no full name to hash.** That logs `The FullName for [...] could not be found.` first. An ordinary concrete message type always has one.

This check runs before the host-loopback branch is even considered, so a host sending to its own other half is refused the same way as every other peer: loudly, and before anything is written.

The fix for a collision is renaming one of the two types, or moving it to another namespace, because the hash follows the full name. Registering a handler does not help: the same lookup refuses the handler too.

Two types resolving to the same wire hash

`MessageManager` and `RpcManager` each remember which type first claimed each wire hash. If a second, different type produces the same hash, the registry does not pick a winner: it refuses both types, for sending, for registering a handler and for delivering a received body, for as long as they collide.

The alternative would be delivering the second type's body to the first type's handler, reading bytes as the wrong shape. The registry treats a collision as unresolvable and names both types when it happens, rather than let ambiguous data through.

A call that reaches nobody

The receiver looks up the call's target system before any handler runs. If the receiver does not hold that system, the call is dropped with no log and no violation: an interest cull, a despawn, or a spawn whose priming full has not landed yet all look identical from here, and none of them are faults.

A routed call (`RpcRoute.TargetRelay`) has one more check once the system is found, still before any handler runs: the connection it names must be connected and must actually observe the addressed system. A target that has left or no longer observes it is refused the same way, silently.

So when a call goes nowhere, check interest before framing: does the receiver still hold the system, and does the intended recipient still observe it. A wire-level bug would produce a decode error or a violation. Silence is interest.

Nothing happens on the sender

`RpcSelfDelivery` defaults to `None`, meaning the sender does not see its own call at all. A sender that expects immediate local feedback and gets nothing has not lost a call; it never asked to see it.

- `RpcSelfDelivery.Immediate` runs the sender's own handlers at the send site, before the call has gone anywhere. It runs even if the server later refuses the call.
- `RpcSelfDelivery.OnDelivery` runs the sender's handlers only once the call has actually been

delivered (and not at all if refused). On the server's own send this behaves like `Immediate`, since the server is the delivery point.

If a sender's own handler never fires, check what `RpcSelfDelivery` the send asked for.

A client's call refused

Three separate gates can refuse a client's call, in order:

`RpcAccess`. The server checks the sender against the system's `RpcAccess`. The sender must observe the system, and then `RpcSendAccess.AnyClient` admits any client while otherwise only the system's controller is admitted. The same check applies to a remote client's call and to a host's own client half calling its own server half. `CanLocalClientSendRpc` is a separate, earlier gate on the sending side: it lets a client queue a call upstream at all even when it is not the controller, and does not enter the server's admission check. If the system's access does not admit the sender, the receive is refused, and the server raises `RpcSendPermissionViolation` with the `SystemId` of the addressed system.

A server handler returning `RpcRelayAction.Cancel`. The server runs every registered handler and folds their answers: any single `Cancel` settles it, and no later `Relay` can overturn it. A handler that throws is treated as `Cancel` too: the throw counts as a refusal to relay, and the handler collection logs it, but the remaining handlers still receive the call.

`MaximumInboundRpcsPerConnectionPerDrain`. The server bounds how many calls one connection may deliver in a single drain (default 64). Once a connection exceeds it, `RpcFloodViolation` is raised (once per drain) with the received and allowed counts, and the rest of that drain's calls from that connection are dropped.

`RpcSendPermissionViolation` and `RpcFloodViolation` tell these two failure modes apart: a permission violation means the system's access rule rejected the sender; a flood violation means the sender delivered more calls in one drain than the bound admits.

The real host symptom: not "fires twice"

The engine already prevents every double-fire shape a naive host implementation could produce. The actual symptom to watch for with a host is different: with `RpcSelfDelivery.Immediate` on an addressed send, the host's client half runs a copy of calls addressed to *other* players, not only the ones addressed to itself.

This falls out of what a host is: one peer running both the server and a client. A call routed to another connection is relayed by the server half; if that call was sent with `Immediate` self-delivery, the sender's own handlers already ran at the send site, on the same process the server is running on. The host's client half is not a separate machine, so it sees the immediate copy of every call its server half sends, not just the ones meant for it. Check the target of the call, not whether it "happened twice."

A message type the receiver never registered

A receiver with no handler registered for a type is not an error. The message packet carries its own byte count, so a peer with no parser registered for that hash skips the packet's bytes and moves to the next one. No error, no log.

This is why a missing handler looks exactly like silence: the send succeeds, the packet arrives, and the receiver quietly steps over it. If a message type seems to vanish in transit, check that the receiving peer registered a handler for it. The sender needs no registration of its own.

Sending a Remote Call From a Component

Driving the core API directly? See System RPCs.

Resolving the system to send from

`TrySendRpc` is a method on `NetworkSystem`, so a `MonoBehaviour` needs that system before it can call it. A script derived from `NucleusBehaviour<TComponent0>` already has one: `NetworkSystem`, inherited from `NucleusBehaviourBase`, is set once the script's required system links, and stays set until it unlinks. `NucleusBehaviour<TComponent0>` separately exposes `Component`, the typed `NetworkComponent` the system carries; that property is for reading replicated values, not for sending calls.

```
public class DemoScoreRpcSender : NucleusBehaviour<DemoScoreComponent>
{
    private void SendScoreRpc()
    {
        bool sent = NetworkSystem.TrySendRpc(RpcTarget.Server, Channel.Reliable, new AddSc
oreRpc { Amount = 1 });
    }
}
```

A script that is not built around one required component instead resolves a `NetworkSystemObject` on the object and asks it for a system:

```
if (networkSystemObject.TryGetFirstSystem(out NetworkSystem networkSystem))
    networkSystem.TrySendRpc(RpcTarget.Server, Channel.Reliable, new AddScoreRpc { Amount
= 1 });
```

`TryGetFirstSystem` returns the object's first-linked system, the correct one to address a call from when several systems share the object.

Declaring the payload

A call is a plain struct that implements `IRpc` and carries `[NetworkType]`:

```
[NetworkType]
public struct AddScoreRpc : IRpc
{
    public int Amount;
}
```

`[NetworkType]` is what the Nucleus source generator uses to produce the struct's serialization; without the generator having run over it, the struct compiles but never replicates. In a Unity project that generator ships as `Nucleus.CodeAnalysis.SourceGenerators.dll` under `Assets/Nucleus`, and it must keep its `RoslynAnalyzer` asset label for Unity to run it at compile time. A struct declared with the label stripped, or with the DLL missing, still builds, still passes type-checking, and still sends nothing.

Sending the call

`NetworkSystem.TrySendRpc<T0>` queues the call to the peers a target names, and returns a `bool`:

```
bool sent = NetworkSystem.TrySendRpc(RpcTarget.Server, Channel.Reliable, new AddScoreRpc {
 Amount = 1 });
```

The return value reports only what this peer can see. `true` means the call was queued, or was invoked locally; it does not mean the call was delivered, and a client routing a call through the server has no way to know that it arrived. `false` means nothing was sent, most often because the local client does not hold the

access this call's target requires. Call `TrySendRpc` from wherever the game decides to act: input handling, a UI callback, or a script method fired by an animation event.

Guarding the send

Address checks and role checks belong to `NucleusBehaviourBase`, not to hand-written comparisons against connections or transport state. `EnsureIsController(ControllerType)` reports whether this peer controls the linked system, and logs a warning naming the caller the first time it fails from that calling member:

```
private void OnFireButton()
{
    if (!EnsureIsController(ControllerType.Client))
        return;

    NetworkSystem.TrySendRpc(RpcTarget.Server, Channel.Reliable, new AddScoreRpc { Amount
= 1 });
}
```

`IsStarted(Invoker)` answers whether a role is up for a behaviour whose system is linked, for code that needs to know the role rather than control:

```
if (IsStarted(Invoker.Client))
    NetworkSystem.TrySendRpc(RpcTarget.Server, Channel.Reliable, new AddScoreRpc { Amount
= 1 });
```

Both read `false` before a system is linked, so a script can call them without a separate null check on `NetworkSystem`.

Reaching the RPC manager

There is no Unity component for the RPC manager, and nothing about a call's targets, self-delivery, or access appears in the inspector. `RpcManager` lives on `CoreManager`, reached as `NucleusUnity.BoundCoreManager.RpcManager` from outside a `NucleusBehaviourBase`, or as `CoreManager.RpcManager` from inside one:

```
RpcManager rpcManager = CoreManager.RpcManager;
```

Every setting on that subject is a plain C# assignment against the manager or the system, not a field a script exposes to the inspector.

Targets, self-delivery, and access

For `RpcTarget`, `RpcSelfDelivery`, and `RpcSendAccess`, see System RPCs.

State, a Message, or a Call?

The three shapes

A **replicated member** carries a current value. The server writes it, the wire keeps every observer's copy converging on it, and a late observer is brought up to date the moment it starts observing.

A **message** is addressed to a peer and forgotten. It has no address beyond the connection it travels to, no acknowledgment, and no record that it happened once the handler returns.

A **call** is addressed to a `NetworkSystem`. The receiver resolves that system before any handler runs, so a call for a system the receiver does not have is stepped over rather than delivered to the wrong place.

What each costs to send

A message's envelope carries a hash of the message type (`ushort`), so the receiver knows which handler decodes the body. There is no address and no framing beyond that.

A call's envelope (`RpcPacket`) carries the addressed system's id (`SystemId`), a route (`RpcRoute`, telling the server whether it is a fan-out to observers or bound for one target), and the call's own type hash. The body itself is length-framed (`PackedBytes`) so an unresolvable call can be skipped without decoding it, and the packet after it still reads correctly.

A replicated member sends a delta against a baseline rather than a fresh value every time, so its ongoing cost is the size of what changed, not the size of the whole value.

The late-joiner rule

A replicated member shows a new observer the value as of the last write. That is what "current value" means: the member's job is to be caught up to, not to be replayed.

A message or a call that fired before a peer started observing is gone. Nothing records it for a peer that arrives later, and neither `RpcManager` nor `MessageManager` keeps a backlog to replay. If a peer needs to know something happened before it joined, that fact has to live in a replicated member, not in the message or call that first announced it.

This is usually what decides between the three: does a newly-arrived peer need to know this, or only peers who were already there to see it happen?

Wire-type hashes are per mechanism

Messages and calls each keep their own registry of type hashes, resolved once per type and cached. Two message types (or two call types) landing on the same hash are caught the first time the second one is used, whether to send it or to register a handler for it, and an error names both types. From then on both types are refused, rather than silently letting one type's body be decoded as the other's. A collision between a message type and a call type cannot happen this way, because the two registries are separate; it can only happen within one mechanism.

The same protection holds on receive: a message or call whose hash two types collide on is skipped rather than guessed at. A hash with no handler registered is not an error: a message is skipped because nothing is there to decode it, and a call is not decoded, though the server still relays it.

Picking one

- **A score.** Every peer, including one that joins mid-match, needs to see the current value. Replicated member.

- **A door's open state.** Same reasoning: a peer that walks up to a door five minutes after it opened needs to see it open, not miss the event that opened it. Replicated member.
- **A chat line.** Nobody who wasn't there needs to receive it later, and it isn't addressed to an object in the world, just to peers. Message.
- **A hit effect.** It plays once, on the object it happened to, and a peer who joins after the hit doesn't need to see it play. That address to a specific `NetworkSystem` is what makes it a call rather than a message.
- **A server answer to one player's request.** Addressed to that one player's system, resolved once, done. Call, sent with a routed `RpcRoute` rather than fanned out to every observer.

NetworkNavMeshAgent

NetworkNavMeshAgent sends the leg being walked, not the pose

`NetworkNavMeshAgent` replicates a `NavMeshAgent` by sending the leg it is walking, not its pose. The peer that controls the object captures a leg — the point it is heading for, the point it set off from, and the agent's speed, angular speed, and acceleration — only when a new leg begins. Every other peer walks its own copy of that agent from one point to the other at the replicated settings, turning toward its direction of travel, with its own `NavMeshAgent` switched off so nothing path-finds twice.

An agent that holds its course costs nothing however far it travels: silence means the follower is still walking the leg it was last told about. Use this instead of `NetworkTransform`, not alongside it — a spawn already carries where the agent stands, and a `NetworkTransform` added on top would replicate the same motion twice.

NetworkNavMeshAgent needs Unity's AI module

`NetworkNavMeshAgent` compiles only while Unity's AI module (`com.unity.modules.ai`) is enabled, as it is in a new project. The integration sets the `NUCLEUS_NAVMESH` define for you when the module is present. With the module disabled, the component is left out and the rest of the integration still compiles.

Inspector fields

Field	Default	Purpose
Point Change Tolerance (`_pointChangeTolerance`)	0.05	How far the agent's steering target must move, in metres, before it counts as a new leg. A path recalculation nudges the target by small amounts without the agent having actually turned; too small a value sends a leg for that nudge.
Resync Distance (`_resyncDistance`)	0.5	How far a follower may stand from a new leg's origin before it starts closing the gap. Below this it just retargets, so an ordinary correction is invisible.
Teleport Distance (`_teleportDistance`)	6	How far a follower may stand from a new leg's origin before it is placed there outright instead of walking the gap off. Above this the gap is a discontinuity — a warp, or a peer that has fallen too far behind to catch up in a reasonable time.

Rotation Enabled (`_rotationEnabled`)	on	Turns a following peer toward its direction of travel at the replicated angular speed. Rotation is derived from the path, not sent — an agent's rotation changes on nearly every tick, so replicating it would erase most of what this component saves.
Agent Management Enabled (`_agentManagementEnabled`)	on	Disables the attached `NavMeshAgent` on every non-controlling peer and enables it on the controller. A follower's own agent must not run: it would path-find toward a destination it was never told about and fight the replicated leg.
Send Interval (`_sendInterval`)	`SendInterval.Normal`	How often the replicated members ride. Normal sends a leg on the tick it begins; a larger interval paces the whole agent, delaying a leg by up to that interval.

Driving it from code

`SetDestination`, `Stop`, `Resume`, and `Warp` all act on the attached `NavMeshAgent`, and all are gated on the calling peer controlling the object (`IsController(ControllerType.AnyController)`). A call from a peer that only follows is refused rather than silently diverging.

```
public bool SetDestination(Vector3 destination);
public void Stop();
public void Resume();
public bool Warp(Vector3 position);
```

The path is found locally on the controlling peer, and only the legs it produces are replicated, so no other peer ever learns the destination itself. A `Warp` reaches followers through the next leg, whose origin is the position warped to. A follower is placed there outright only when the warp leaves it at or past Teleport Distance from that origin; a shorter warp past Resync Distance is walked off like any other gap, and one inside Resync Distance is not corrected at all.

Values you can read

```
public Vector3 NextPoint { get; }
public Vector3 PreviousPoint { get; }
public float FollowerSpeed { get; }
```

`NextPoint` and `PreviousPoint` are the current leg's endpoints in world space, as the replication currently has them. `FollowerSpeed` is the speed a following peer is walking at right now — it ramps toward the replicated speed rather than starting there, and drops away again through a turn, so it reads as zero on the controller (which walks its real agent instead).

The replicated component

Behind the `MonoBehaviour` sits `UnityNavMeshAgentComponent`, the replicated `NetworkComponent` that actually carries the leg:

- `NextPoint` and `PreviousPoint` — the leg's endpoints.
- `Speed`, `AngularSpeed`, `Acceleration` — the agent settings a follower needs to reproduce the walk.
- `IsStopped` — whether the agent is halted.

`NetworkNavMeshAgent` binds this component on link, applies the inspector settings to it, and captures a leg whenever this peer controls the object; on every other peer it seats the object where the sender stood when it spawns and advances it toward `NextPoint` each frame.

Corrections

A correction must not read as a stop. Re-seating a follower on every new leg would make an agent that is tracking perfectly well twitch once per corner, which looks like a replication fault rather than the correction it is. So:

- Under Resync Distance, a follower simply retargets onto the new leg and keeps walking — the error is invisible.
- Between Resync Distance and Teleport Distance, the follower owes the gap as ground to make up, and walks it off over the next moment instead of being moved.
- At or past Teleport Distance, the gap is treated as a discontinuity, and the follower is placed on the new leg's origin outright.

A received spawn is always seated regardless of these thresholds, because a peer arriving mid-leg has no position of its own to correct from. It is seated where the sender stood when the spawn was sent, moving at the sender's velocity, rather than on the leg origin, which can be metres behind an agent that is well into a long leg.

Requirements

- The controlling peer needs a baked NavMesh covering the agent's path, and so does any peer that may take control later. A follower never path-finds or queries the NavMesh; it moves its transform along the leg it was told about, so following needs no mesh.
- The prefab's agent settings (tolerance, resync and teleport distances, rotation, agent management, send interval) must match on every peer, since the follower walks entirely from the replicated leg and its own local settings.

The network scene manifest and scene references

Why scenes need a manifest

The wire never carries a scene's path or name, only a `ushort` identifier. Something has to map that number back to an actual scene asset on each peer, and that something is `NetworkSceneManifest`, a `ScriptableObject` under `Nucleus.Integrations.Unity.Scenes`. It is also the identifier registry itself: a networked prefab carries its id stamped on the prefab asset, but a scene has nowhere comparable to hold one, so the manifest entry keyed by the scene's GUID *is* where the id lives.

A project has exactly one manifest. It lives under a `Resources` folder and is resolved by convention, not by being dragged into an inspector.

NetworkSceneManifest

`NetworkSceneManifest` exposes:

- `Entries` — the `IReadOnlyList<SceneEntry>` of every networked scene this peer knows.
- `UnsetSceneId` — the constant `0`, reserved so a lookup that failed can be told apart from "the first scene."
- `ResourceName` — the constant `"NetworkSceneManifest"`, the file name the manifest must have under a `Resources` folder.
- `TryGetScene(ushort sceneId, out SceneEntry sceneEntry)` — resolves an entry from the id the server named on the wire.
- `TryGetSceneId(string scenePath, out ushort sceneId)` — resolves an id from a scene's asset path.
- `TryGetSceneIdByGuid(string sceneGuid, out ushort sceneId)` — resolves an id from a scene's GUID; prefer this one, since it is what the id is actually keyed to.
- `LoadProjectManifest()` — a static method that loads the project's manifest via `Resources.Load<NetworkSceneManifest>(ResourceName)`. Nothing is cached: a manifest held in a static would keep answering for an asset a later session replaced.

```
NetworkSceneManifest manifest = NetworkSceneManifest.LoadProjectManifest();

if (manifest != null && manifest.TryGetScene(sceneId, out NetworkSceneManifest.SceneEntry
entry))
{
    // entry.ScenePath is what you hand to Unity's scene loader.
}
```

SceneEntry

Each `SceneEntry` in `Entries` carries:

Field	Purpose
`SceneId`	The identifier the server names this scene by on the wire. Assigned once by the build and never renumbered.

`SceneGuid`	The scene asset's GUID. The identifier is keyed to this, so a rename or move keeps it.
`ScenePath`	The scene asset's path, which is what the loader hands to Unity.
`RequiredBundleId`	The content bundle this scene ships in, or zero when it ships inside the player build. A scene in a bundle can't load until that bundle is held.
`DeliveryMode`	A `SceneDeliveryMode`, either `BuildSettings` or `Addressables`, deciding which machinery the loader opens the scene through.

NetworkSceneReference

Fields that need to name a scene do not store a path or an id. They store a `NetworkSceneReference`, a serializable struct that holds only a scene asset's GUID (`SceneGuid`), plus `HasScene` (true once a GUID is assigned) and `TryGetSceneId(out ushort sceneId)`.

`TryGetSceneId` resolves through the project's manifest at the moment it's called, via `TryGetSceneIdByGuid`. Nothing is baked in at authoring time, so a scene added to the manifest after the reference was set resolves as soon as the manifest is rebuilt, and a scene that stopped being networked resolves to `UnsetSceneId` instead of silently opening the wrong thing.

```
[SerializeField]
private NetworkSceneReference _lobbyScene;

if (_lobbyScene.TryGetSceneId(out ushort sceneId))
{
    // Tell the server to open sceneId.
}
```

Because only the GUID is stored, renaming or moving the scene asset never repoints the field to a different scene and never breaks it.

In the inspector, `NetworkSceneReferenceDrawer` draws a `NetworkSceneReference` field as an ordinary scene-asset picker (an `ObjectField` for `SceneAsset`), with the resolved scene id shown alongside it as a small label. You drag a scene in exactly as you would for a plain `SceneAsset` field; the GUID is captured for you.

Minting a scene id

Scene ids are not assigned by hand. **Nucleus > Rebuild Network Scene Manifest** rescans the project and writes `Entries`, and `NetworkSceneManifestAutoRebuild` runs that rebuild automatically. It listens for four triggers: the Build Settings scene list changing, a scene asset being imported, the project's Addressables entries changing, and a player build starting (the last is the backstop, catching anything the other three missed before it ships).

The scan covers both ways a scene can be networked: scenes listed in Build Settings, and scenes the project has made Addressable. The Addressables pass matters because marking a scene Addressable *removes it from Build Settings* — without a dedicated pass, the very act of making a scene streamable would drop it out of the manifest. Whichever way a scene is found, its `DeliveryMode` is derived from the project (Addressable scenes record `SceneDeliveryMode.Addressables`, everything else records `BuildSettings`), so making a scene Addressable is the whole of the change needed to switch it.

Once assigned, a scene's id is preserved across rebuilds; a rebuild only adds ids for newly-found scenes and drops entries for scenes no longer present.

Why only a ushort crosses the wire

The server names a scene by `SceneId` alone. No path, no name, no build index, and how the scene is delivered (`DeliveryMode`, `RequiredBundleId`) is never replicated either — those are purely how the local peer resolves the id into something it can load. This is deliberate: paths and names change with renames and refactors, but the manifest that maps a stable id to a stable GUID does not need to, so a build can point at different content, or ship a renamed scene, without ever touching what's on the wire.

Driving a predicted body

Driving the core API directly? See Locally predicted bodies.

A body a client drives directly - a player's own physics character, for example - should not wait a round trip to respond. `ProjectedRigidbody` supports this: the driving peer predicts the body locally, applying its own client state the instant it arrives, while the server still owns where the body really is and the driver reconciles toward it as confirmations return.

Turning prediction on

`ProjectedRigidbody.Convergence` is a `PhysicsConvergence`. Set `Convergence.IsLocallyPredicted` to `true` on the peer steering the body with its own client state, and back to `false` when it stops:

```
projectedRigidbody.Convergence.IsLocallyPredicted = true;
```

The flag only has an effect on a peer that is not the body's controller. `ProjectedRigidbody` runs its follower only where `NetworkSystem.IsController(ControllerType.AnyController)` is false; on the controlling peer the body simulates freely and its result is sent, so there is nothing to predict. The pattern the flag is for is the one the PhysicsBallMover demo uses: the server controls and simulates the body, and one client steers it through its own client state channel while applying the same client state to its local copy.

While it is set, the follower only trims a position or rotation error once that error passes its threshold, and never teleports the body. It does not keep the velocity your own simulation produced: every step it sets the body's velocity from its projected target, as it does for any proxy. For your client state to show up at once, the target has to include it; the PhysicsBallMover demo assigns `Convergence.TargetProvider` a provider that does. Clear the flag the moment this peer stops driving the body - a proxy left with `IsLocallyPredicted` set has no client state of its own to trust.

Applying force on the right cadence

Under a `PhysicsSimulationDriver`, the physics world steps on the network tick, not on Unity's `FixedUpdate`. Applying force from client state in `FixedUpdate` there either does nothing or fires at the wrong rate.

Two ways to run on the actual step cadence:

- Subscribe to `PhysicsSimulationDriver.WorldStepStarted`, raised before each world step so gameplay code can apply per-step forces from client state on the simulated cadence:

```
driver.WorldStepStarted += (tick, stepDelta) =>
{
    rigidbody.AddForce(clientStateDirection * forceScale);
};
```

- Or, on a `NucleusBehaviourBase` (the base every Nucleus MonoBehaviour that participates in the network loop derives from), override `OnEarlyFixedUpdate`:

```
protected override void OnEarlyFixedUpdate(StepDelta stepDelta, SimulationContext simulati
onContext)
{
    rigidbody.AddForce(clientStateDirection * forceScale);
}
```

`OnEarlyFixedUpdate` maps to `NetworkLoopSteps.EarlyFixedUpdate`, which runs before the world

step - the same point `WorldStepStarted` fires. Its parameters are a `StepDelta`, not a plain `float` (use its `FixedDelta` for the step's fixed timing), and a `SimulationContext` naming the pass the step belongs to (see Replaying a correction in Unity).

Tuning the feel

A predicted body does not use `Blend Per Tick` (`PhysicsConvergenceSettings.BlendPerTick`) - that setting governs ordinary proxies. Instead it uses `Prediction Correction Rate` (`Convergence.PredictionCorrectionRate`, exposed in the inspector as *Prediction Correction Rate*): the fraction of a genuine position or rotation divergence closed per tick, once past the threshold.

Higher values snap the body back to the server harder and sooner. Lower values let the prediction ride looser and correct gently. Zero disables the correction entirely, so the body is pure, unreconciled prediction - it never blends toward the server, and it is exempt from the teleport-on-large-divergence path too, so a body left to drift this way effectively never snaps back. That is rarely what you want; leave some nonzero rate so a genuine desync still closes.

Remote client state proxies

A body a client controls outright (its `ControllerConnectionId` is set) is a different case: on every other peer it is a proxy driven by that remote controller's client state, not a predicted body. By default (`RemoteExtrapolationEnabled = true`) such a proxy extrapolates its motion forward from the last received state, riding nearer to present time at the cost of overshooting on abrupt client state changes: a stopping body slides on a little, then corrects once the slower state arrives.

Set `RemoteExtrapolationEnabled` to `false` on a body driven by a remote controller and its proxy copy instead follows the received state on the interpolation buffer as a kinematic body - always a step behind, but smooth and free of overshoot. This only takes effect once the system also has a `ControllerConnectionId` set (unset, the body has no remote controller to speak of, and none of this applies).

Prefer extrapolation for a fast, client-state-driven body where responsiveness matters more than a rare overshoot. Prefer the kinematic, extrapolation-off path where overshoot would look worse than lag - dense contacts, tight geometry, or a body other players might stand near.

The client state channel itself

None of the above covers how the driving peer's client state reaches the network or gets reconciled against the server's confirmation - that is the client state and reconcile pipeline, not the body. See the client state and reconcile pages for that half.

What a correction looks like

A healthy predicted body is quiet almost all the time: your own client state drives it, and a correct prediction that matches what the server later confirms reconciles to nothing visible. A correction only becomes visible on a genuine divergence - a hit, a collision the server resolved differently, a dropped tick of client state.

There are two distinct phases to that correction. The impact is the moment the divergence is detected - the tick the position or rotation error crosses its threshold. The recovery is everything after: the velocity bias `PredictionCorrectionRate` applies each tick, closing the gap smoothly through the solver rather than snapping the pose. A body correcting well looks like a brief nudge off its expected path that eases back in over a few ticks, not a teleport. If corrections look like a snap, `PredictionCorrectionRate` is too high, the threshold is too tight, or the divergence causing them is worth chasing on its own.

How physics replication works

The loop

Every physics body Nucleus replicates follows the same loop, regardless of which engine drives it.

The controlling peer simulates the body normally and, once per tick, captures the stepped result: world position, world rotation, and a set of condition flags. That capture rides the wire as ordinary replicated members, alongside the tick it was captured at.

Every other peer holds a proxy for that body. Each physics step, the proxy rebuilds the latest server snapshot from those members, projects it forward by however many ticks old it is, and converges its own simulated body onto the projected result. The proxy is never teleported onto the raw received pose; it is steered toward a moving target that accounts for the snapshot's age.

Velocity is derived, not sent

Linear and angular velocity never cross the wire. Each side derives them independently from the change in position and rotation across the last two committed ticks, divided by the time between those ticks.

This halves what a moving body costs: a body in motion would otherwise carry position, rotation, *and* both velocities every tick it changes. A resting body carries nothing new at all, since an unchanged capture doesn't restamp its tick.

A proxy keeps simulating

A proxy body is never written directly onto the received pose. It keeps stepping through the same physics engine as the controller, with its velocity steered toward the projected target and its pose blended or corrected against divergence. Contacts, stacks, and collisions involving a proxy stay real on every peer, because every peer is still actually simulating it — not just moving a transform to match a number that arrived.

What's on the wire, and what isn't

Three things replicate: world position, world rotation, and a small set of condition flags (whether the body is sleeping, whether it's kinematic, and whether it has been captured at all). A capture tick rides alongside them so a receiver can anchor its projection to the snapshot's exact age.

No engine type ever crosses that boundary. The replicated members and the snapshot they rebuild are expressed in `System.Numerics` — `Vector3` and `Quaternion` — never an engine-specific vector, quaternion, or rigidbody handle. The core has no dependency on any particular physics engine; an integration is responsible for reading its engine's body into that shape and writing the projected result back onto it.

Position and rotation ride Divine or Interval transmission like any other member; the choice is per-member, not physics-specific. Divine can go quiet for stretches where Interval would still send every tick. Divine projection is Pro: a free build accepts `Divine` and replicates the member with ordinary deltas, as `Interval` does.

What isn't built

This model does not include:

- Replay-correct collision callbacks

This is planned, not hidden. A resimulated, full-world rollback approach to physics was tried and archived;

the model here (capture, project, converge) is what shipped in its place.

Judging a shot against where targets were is a separate feature that does not touch physics bodies at all: lag compensation (Pro) keeps its own hitbox history and its own geometry. Catching a freshly spawned projectile up through the latency it lost is spawn compensation (Pro).

Where core ends and integration begins

Everything above lives in the core engine, engine-agnostic and expressed in `System.Numerics`. Turning it into motion on an actual rigidbody — reading a body's state each tick, applying a projected target, wiring up sleep and kinematic flags — is an integration's job. Nucleus ships a Unity integration that does this for Unity's physics.

The core-side member and convergence types are the entry point for a custom or unsupported engine. The Unity-side components are the entry point for Unity.

Physics Synchronization

Pages in this category:

- Physics execution modes
- ProjectedRigidbody
- How physics replication works
- NetworkTransform or ProjectedRigidbody
- Replicate a rigidbody
- Driving a predicted body
- NetworkNavMeshAgent
- PhysicsSimulationDriver
- Local physics scenes
- Physics troubleshooting

Local physics scenes

Each scene gets its own physics world

Give each loaded scene its own local physics world so a body only collides with other bodies loaded into that same scene. Two switches control it, both on the Unity scene-loading components.

`UnitySceneLoader.LocalPhysicsEnabled` (backing field `_localPhysicsEnabled`) loads each networked scene with `LocalPhysicsMode.Physics3D` instead of the default shared world, and attaches a `PhysicsSimulationDriver` to it.

`UnitySceneManager`'s "Auto Simulate Stacked Scenes" inspector field (`_automaticStackedSceneSimulationEnabled`) pushes that same setting onto the `UnitySceneLoader` the manager creates, so every stacked instance gets a local physics world without touching the loader directly.

Why the driver is needed

Unity never auto-simulates a local physics scene. A scene loaded with `LocalPhysicsMode.Physics3D` sits inert until something steps it. `NetworkSceneBinder.AttachPhysicsSimulationDriver` adds a `PhysicsSimulationDriver` to the scene the moment it loads, and that driver is what the network loop steps on tick. Without local physics enabled, bodies share the one default world instead, which under the default `PhysicsExecutionMode.NucleusPerTick` is stepped on tick by the driver Nucleus adds for it automatically.

What changes for a body

A body only collides within its own scene's physics world. Moving a `ProjectedRigidbody` to another scene changes what it can hit: it stops seeing the bodies it left behind and starts seeing whatever local world it landed in.

Moving a body across scenes

`ProjectedRigidbody.NotifySceneChanged` re-resolves which driver a body steps under. A scene move made with `MoveGameObjectToScene` fires no `OnEnable` or `OnDisable` on the component, and the driver is cached at `OnEnable`, so without this call the body keeps stepping the world it left. `NotifySceneChanged` leaves the old driver and asks the new scene for its driver; a local-physics destination already has its driver attached from load, so the ask resolves immediately and the body steps the right world from the next tick.

You don't need to call this yourself for a spawn move: the spawn handler calls it on every `ProjectedRigidbody` under a moved system right after the move.

Honest limits

This is step routing onto Unity's own per-scene physics world, not a fully separated per-scene physics feature. The engine's feature list tracks "Per-scene local physics" (#302) as step routing shipped, world separation not shipped.

Bodies in a local physics scene still need manual interpolation; local physics does not give you that for free.

See also

- UnitySceneLoader component and UnitySceneManager component for the loader and manager settings

themselves

- Opening and closing scene instances for opening the stacked instances this applies to
- Moving and carrying objects between scenes for moving systems between scenes

NetworkNavMeshAgent

NetworkNavMeshAgent sends the leg being walked, not the pose

`NetworkNavMeshAgent` replicates a `NavMeshAgent` by sending the leg it is walking, not its pose. The peer that controls the object captures a leg — the point it is heading for, the point it set off from, and the agent's speed, angular speed, and acceleration — only when a new leg begins. Every other peer walks its own copy of that agent from one point to the other at the replicated settings, turning toward its direction of travel, with its own `NavMeshAgent` switched off so nothing path-finds twice.

An agent that holds its course costs nothing however far it travels: silence means the follower is still walking the leg it was last told about. Use this instead of `NetworkTransform`, not alongside it — a spawn already carries where the agent stands, and a `NetworkTransform` added on top would replicate the same motion twice.

NetworkNavMeshAgent needs Unity's AI module

`NetworkNavMeshAgent` compiles only while Unity's AI module (`com.unity.modules.ai`) is enabled, as it is in a new project. The integration sets the `NUCLEUS_NAVMESH` define for you when the module is present. With the module disabled, the component is left out and the rest of the integration still compiles.

Inspector fields

Field	Default	Purpose
Point Change Tolerance (`_pointChangeTolerance`)	0.05	How far the agent's steering target must move, in metres, before it counts as a new leg. A path recalculation nudges the target by small amounts without the agent having actually turned; too small a value sends a leg for that nudge.
Resync Distance (`_resyncDistance`)	0.5	How far a follower may stand from a new leg's origin before it starts closing the gap. Below this it just retargets, so an ordinary correction is invisible.
Teleport Distance (`_teleportDistance`)	6	How far a follower may stand from a new leg's origin before it is placed there outright instead of walking the gap off. Above this the gap is a discontinuity — a warp, or a peer that has fallen too far behind to catch up in a reasonable time.

Rotation Enabled (`_rotationEnabled`)	on	Turns a following peer toward its direction of travel at the replicated angular speed. Rotation is derived from the path, not sent — an agent's rotation changes on nearly every tick, so replicating it would erase most of what this component saves.
Agent Management Enabled (`_agentManagementEnabled`)	on	Disables the attached `NavMeshAgent` on every non-controlling peer and enables it on the controller. A follower's own agent must not run: it would path-find toward a destination it was never told about and fight the replicated leg.
Send Interval (`_sendInterval`)	`SendInterval.Normal`	How often the replicated members ride. Normal sends a leg on the tick it begins; a larger interval paces the whole agent, delaying a leg by up to that interval.

Driving it from code

`SetDestination`, `Stop`, `Resume`, and `Warp` all act on the attached `NavMeshAgent`, and all are gated on the calling peer controlling the object (`IsController(ControllerType.AnyController)`). A call from a peer that only follows is refused rather than silently diverging.

```
public bool SetDestination(Vector3 destination);
public void Stop();
public void Resume();
public bool Warp(Vector3 position);
```

The path is found locally on the controlling peer, and only the legs it produces are replicated, so no other peer ever learns the destination itself. A `Warp` reaches followers through the next leg, whose origin is the position warped to. A follower is placed there outright only when the warp leaves it at or past Teleport Distance from that origin; a shorter warp past Resync Distance is walked off like any other gap, and one inside Resync Distance is not corrected at all.

Values you can read

```
public Vector3 NextPoint { get; }
public Vector3 PreviousPoint { get; }
public float FollowerSpeed { get; }
```

`NextPoint` and `PreviousPoint` are the current leg's endpoints in world space, as the replication currently has them. `FollowerSpeed` is the speed a following peer is walking at right now — it ramps toward the replicated speed rather than starting there, and drops away again through a turn, so it reads as zero on the controller (which walks its real agent instead).

The replicated component

Behind the `MonoBehaviour` sits `UnityNavMeshAgentComponent`, the replicated `NetworkComponent` that actually carries the leg:

- `NextPoint` and `PreviousPoint` — the leg's endpoints.
- `Speed`, `AngularSpeed`, `Acceleration` — the agent settings a follower needs to reproduce the walk.
- `IsStopped` — whether the agent is halted.

`NetworkNavMeshAgent` binds this component on link, applies the inspector settings to it, and captures a leg whenever this peer controls the object; on every other peer it seats the object where the sender stood when it spawns and advances it toward `NextPoint` each frame.

Corrections

A correction must not read as a stop. Re-seating a follower on every new leg would make an agent that is tracking perfectly well twitch once per corner, which looks like a replication fault rather than the correction it is. So:

- Under Resync Distance, a follower simply retargets onto the new leg and keeps walking — the error is invisible.
- Between Resync Distance and Teleport Distance, the follower owes the gap as ground to make up, and walks it off over the next moment instead of being moved.
- At or past Teleport Distance, the gap is treated as a discontinuity, and the follower is placed on the new leg's origin outright.

A received spawn is always seated regardless of these thresholds, because a peer arriving mid-leg has no position of its own to correct from. It is seated where the sender stood when the spawn was sent, moving at the sender's velocity, rather than on the leg origin, which can be metres behind an agent that is well into a long leg.

Requirements

- The controlling peer needs a baked NavMesh covering the agent's path, and so does any peer that may take control later. A follower never path-finds or queries the NavMesh; it moves its transform along the leg it was told about, so following needs no mesh.
- The prefab's agent settings (tolerance, resync and teleport distances, rotation, agent management, send interval) must match on every peer, since the follower walks entirely from the replicated leg and its own local settings.

Physics execution modes

`UnityPhysicsManager` decides who steps Unity's physics worlds: Nucleus, once per network tick, or Unity itself on its own fixed step. The choice is process-global and read once when the session starts.

The component

`UnityPhysicsManager` exposes one read-only property, `ExecutionMode`, backed by a serialized field:

- `_executionMode` (`PhysicsExecutionMode`) — which mode is active. Defaults to `PhysicsExecutionMode.NucleusPerTick`.
- `_matchFixedDeltaTimeToTickRateEnabled` (`bool`) — whether Unity's fixed step is aligned to the tick rate. Defaults to `false`, and only applies under `PhysicsExecutionMode.UnityRun`.

The manager pushes `_executionMode` into `PhysicsSimulationCoordinator` from `ManagersInstantiated`, before any `PhysicsSimulationDriver` registers, so the very first driver already sees the chosen mode.

Both fields are read once at session start and are disabled in the inspector while playing; stop play mode to change them.

PhysicsExecutionMode.NucleusPerTick

The default. `PhysicsSimulationCoordinator` takes `UnityEngine.Physics.simulationMode` to `SimulationMode.Script` and steps every physics world itself, once per network tick, removing the sampling phase error `FixedUpdate` would otherwise introduce between the tick and the physics step. Nucleus projected physics on the default world requires this mode. Because the network loop runs on unscaled time, physics in this mode keeps stepping at the tick rate while `Time.timeScale` is zero or slowed.

PhysicsExecutionMode.UnityRun

Not a cheaper version of the same thing — a hard limit. The default world is left on Unity's own auto-simulated step and is never Nucleus-stepped, so it carries no projected physics at all in this mode. `Rigidbody.Interpolate` works again on the default world, because Unity is the one advancing it. Projected physics still works, but only in the local-physics worlds a stacked scene's own `PhysicsSimulationDriver` registers — the default world is excluded from what Nucleus steps.

Matching the fixed step to the tick

`_matchFixedDeltaTimeToTickRateEnabled` applies only under `PhysicsExecutionMode.UnityRun`; it is ignored under `NucleusPerTick`, where the tick already is the step. When enabled, the manager sets `Time.fixedDeltaTime` to one over the network loop's tick rate, so Unity's own fixed step lands on the same cadence as the tick.

The manager captures the previous value in `_previousFixedDeltaTime` and restores it on teardown (`OnDestroy`), tracked by `_hasFixedDeltaTimeOwnership`, so a play session doesn't leave the project's fixed step changed behind it.

Mode versus driver

The execution mode is process-global, chosen once by the `UnityPhysicsManager` in the scene that starts the session. A `PhysicsSimulationDriver` is per scene, registering which scene steps which physics world. This page answers which cadence steps a world; see PhysicsSimulationDriver for which

scene owns which world.

Checking which mode is live at runtime

- `UnityEngine.Physics.simulationMode` reads `Script` under `NucleusPerTick` and stays `FixedUpdate` under `UnityRun`.
- `Time.fixedDeltaTime` equals one over the tick rate only when `UnityRun` is active and the match toggle is on; otherwise it stays at the project default.

PhysicsSimulationDriver

PhysicsSimulationDriver ties one scene's physics to the network tick

`PhysicsSimulationDriver` is a `MonoBehaviour` that ties one scene's physics to the network tick. A scene with a driver steps its bodies on `NetworkLoopSteps.EarlyFixedUpdate` and steps its physics world on `NetworkLoopSteps.LateFixedUpdate`, instead of on Unity's own `FixedUpdate` cadence. Received state deserializes earlier in the same frame and the tick's captures serialize later in it, so captures and transmissions share one clock instead of drifting against Unity's own physics timing.

A driver is per scene. It registers itself with `PhysicsSimulationCoordinator`, the process-wide object that owns the one network-loop registration, the simulation mode, and the peer's tick anchor. A scene is stepped by exactly one driver.

Scenes that share a physics world (every scene not loaded with a local physics world of its own) are stepped once between them. A driver per scene is not the same thing as a physics world per scene: multiple drivers can point at the same world, and the coordinator steps that world once per tick regardless of how many drivers resolve to it.

Events this component raises

Event	Signature	Fires
`WorldStepStarted`	`WorldStepStartedHandler(uint tick, float stepDelta)`	Before each world step. This is where gameplay applies per-step forces from client state on the simulated cadence, since Unity's own `FixedUpdate` no longer matches when the world actually steps.
`BodyRegistered`	`BodyRegisteredHandler(ProjectedRigidbody projectedRigidbody)`	When a body joins the driver's step set.
`BodyDeregistered`	`BodyDeregisteredHandler(ProjectedRigidbody projectedRigidbody)`	When a body leaves the driver's step set.

```
driver.WorldStepStarted += (tick, stepDelta) =>
{
    // Apply per-step forces from client state here.
};
```

`BodyRegistered`/`BodyDeregistered` exist so a diagnostic, editor tool, or convergence monitor can follow the live body set without walking the scene for it every frame. Subscribe and then read `Bodies` to adopt what already exists, otherwise anything attaching after the first spawn misses every body already present.

Values you can read

- `CurrentTick` (`uint`) - the network loop tick currently being simulated.
- `SubtickFraction` (`float`) - how far the render frame has progressed through the current tick, for

phasing smoothed visuals between the last two stepped poses.

- `Bodies` (`IReadOnlyList<ProjectedRigidbody>`) - the bodies this driver steps, in registration order. A diagnostic reads this instead of walking the scene.
- `TryGetSnapshotLocalTick(uint captureTick, out uint snapshotLocalTick)` - maps a received capture tick into local ticks, through the session's shared drift-adaptive anchor. Returns `false` when the capture tick carried no attribution or nothing mapped.

Finding the driver for a scene

`ForScene(Scene scene)` resolves the driver stepping a given scene's physics, or `null` when nothing steps it (the object then runs Unity's own physics cadence). Resolve once and hold the result: this is a scene read plus a dictionary probe, not a field read, and it is also what lets a later deregistration reach the same driver.

```
PhysicsSimulationDriver driver = PhysicsSimulationDriver.ForScene(gameObject.scene);
```

For a driver that may not exist yet - it can be added at runtime rather than authored into the scene, and component `Start` order between two objects is unspecified - use `RegisterForScene`/ `UnregisterForScene` instead:

```
PhysicsSimulationDriver.RegisterForScene(gameObject.scene, OnDriverReady);

void OnDriverReady(PhysicsSimulationDriver driver) { /* ... */ }

// later
PhysicsSimulationDriver.UnregisterForScene(gameObject.scene, OnDriverReady);
```

`DriverReadyHandler` is `void DriverReadyHandler(PhysicsSimulationDriver physicsSimulationDriver)`. If a driver already steps the scene, the callback fires immediately rather than waiting for a registration edge that already passed - a subscriber attaching after the driver registered would otherwise wait forever.

Register / Deregister

`Register(ProjectedRigidbody projectedRigidbody)` and `Deregister(ProjectedRigidbody projectedRigidbody)` add and remove a body from the driver's step set. `Register` is idempotent.

Two drivers cannot claim the same scene. If a second driver enables on a scene a driver already steps, it logs an error naming both GameObjects and does nothing - the first driver keeps stepping the scene, the second driver's bodies are simply never registered to it.

This component is Unity only

Unity only. There is no plain-C# counterpart: physics stepping is a Unity concept.

Physics troubleshooting

Nothing steps at all

A `UnityPhysicsManager` is always present, so a missing manager is never the cause. Check these instead.

The driver logs "found no bound CoreManager to attach its network loop to". `PhysicsSimulationDriver` registers with the shared `PhysicsSimulationCoordinator` on `OnEnable`, and the coordinator needs a bound `CoreManager` to attach its network-loop callbacks. If none is bound yet, the coordinator logs this error and the scene keeps Unity's own physics cadence (`FixedUpdate`) instead of stepping on the tick. The scene needs a `UnityCoreManager`, which binds a `CoreManager` from its own `Awake`, ahead of every ordinary component. Registration is retried the next time a driver registers, so once a `UnityCoreManager` exists, a later driver attaches and steps everything, including scenes whose own driver registered too early.

A scene was loaded with a local physics scene and no driver. A `ProjectedRigidbody` under `PhysicsSimulationDriver.ForScene` finds a driver by the scene it lives in first, then by the physics world that scene belongs to. A scene with its own local physics world and no `PhysicsSimulationDriver` component has nothing stepping that world at all, tick or otherwise; add a driver to the scene.

`PhysicsExecutionMode.UnityRun` is deliberately skipping the default world. `PhysicsSimulationCoordinator.ExecutionMode` set to `UnityRun` leaves the process default physics world on Unity's own `FixedUpdate` and steps only the local-physics worlds that stacked-scene drivers report. A body sitting in the default world under `UnityRun` looks exactly like the "nothing steps" fault but is fine: it is still Unity's own cadence stepping it, not the network tick.

A body jitters or trails

Visual Smoothing with no Smoothed Visual. `ProjectedRigidbody`'s Visual Smoothing (`_visualSmoothing`) low-passes the render pose of `SmoothedVisual` toward the interpolated physics pose, hiding tick-to-tick correction jitter. Without a `SmoothedVisual` assigned it has nothing to act on. Assign a child of the body that holds its renderer, as the demo prefabs do with a child named `Visual`: `ProjectedRigidbody` writes a world pose onto it every frame, so the parent's own motion does not fight it. Leave that child's transform to `ProjectedRigidbody`; another script or animation writing the same transform would.

Blend Mode Position on a body in contact. `ConvergenceBlendMode.Position` (the `_blendMode` field) writes the pose directly toward the target every over-threshold step. On a body resting in a stack or against the ground, that direct pose write fights the solver's own contact resolution, producing visible jitter. Switch to `ConvergenceBlendMode.Velocity` (see below).

Residual Close Maximum Speed set too low. `_residualCloseMaximumSpeed` gates the sub-threshold residual close: above that speed the body falls back to the plain deadband and holds its offset instead of bleeding it off. Set too low, a body that is still genuinely moving reads above the gate on every step, the residual close never runs, and the un-closed offset shows up as a small, persistent trail behind the target. Raise it so a body's ordinary travel speed stays under it.

A body fights the ground or a stack

Set `_blendMode` to `ConvergenceBlendMode.Velocity`. Instead of writing the pose directly, Velocity mode applies the same closing fraction as a solver-integrated velocity bias, so the correction is folded into

physics rather than overriding it, and never disrupts a contact or a stack.

The sub-threshold residual close (`_residualCloseRate`) gates itself on speed for the same reason: it only runs below `_residualCloseMaximumSpeed`, so a body that was just bumped or is still settling holds its offset on the plain deadband instead of the residual close fighting the fresh contact. Above that speed the gentle bleed-off is simply not applied until the body slows back down.

A body snaps

`_teleportDistance` is the positional divergence, in meters, past which the body's whole pose snaps onto the target instead of blending; the smoothed visual jumps too. Zero disables it, so the body always blends however far off it is.

A snap is not always a bug. It legitimately fires on:

- A respawn, where the body's old and new positions have nothing to do with each other.
- A proxy that lost and then recovered its received state, where the buffered pose is stale by more than the threshold.
- Any other large miss between the projected pose and the server's actual pose.

If bodies are snapping on ordinary corrections rather than these cases, raise `_teleportDistance` (or set it to zero to disable teleporting outright) so more of the divergence range blends instead.

A body still steps the world it left after a scene move

A scene move (for example a spawn handler calling `MoveGameObjectToScene`) fires no `OnDisable`/`OnEnable`, but `ProjectedRigidbody` caches its `PhysicsSimulationDriver` at enable. Without further action the body keeps registered with, and stepped by, the driver of the scene it left, not the scene it now lives in.

Call `ProjectedRigidbody.NotifySceneChanged()` right after the move. It deregisters from the old driver and asks the scene the object now lives in for that scene's driver; a local-physics destination already has its driver attached by the time anything can move into it, so the ask resolves immediately and the body steps the right world from the next tick.

Two drivers on one scene

`PhysicsSimulationCoordinator.Register` refuses a second driver for a scene that already has one and logs: *"Scene [...] is already stepped by the driver on [...]; the one on [...] does nothing. A scene is stepped by exactly one driver."* The second `PhysicsSimulationDriver` component does nothing; remove it, there is exactly one driver per scene.

A body finds the driver of the scene its prefab came from rather than the one it lives in. `PhysicsSimulationDriver.ForScene` (and `ProjectedRigidbody`'s own resolution) looks up a driver by the object's own scene handle first, then falls back to whichever driver steps the same physics world. A pooled or wire-spawned instance often lands in the active scene rather than the scene it is conceptually part of; that fallback is what lets it still find the driver that owns its actual physics world instead of one for a scene it never really entered. If a body is being stepped by the wrong driver, check which scene it was actually instantiated or moved into, and call `NotifySceneChanged()` if it moved after `OnEnable`.

A NetworkTransform and a ProjectedRigidbody on the same object

`ProjectedRigidbody` follows a non-server body as a proxy: it either adopts the first received state, follows a remote client state proxy on the interpolation buffer as a kinematic body, or runs the convergence

follower. When a remote-controlled proxy follows the interpolation buffer, `RefreshRemoteClientStateProxy` forces the rigidbody kinematic (`Rigidbody.isKinematic = true`) for as long as that role holds, restoring the original flag when it ends.

Putting a `NetworkTransform` on the same object as a `ProjectedRigidbody` means two components are both trying to move the same transform from replicated state: the `ProjectedRigidbody` drives the physics body (kinematically, while it is a remote client state proxy), and the `NetworkTransform` drives the same transform independently. Use one or the other for a given object's pose, not both.

Reading bandwidth and round-trip time

When grading whether a correction is worth its cost, read the diagnostics pages rather than guessing: Reading network statistics in Unity covers bandwidth, and Round-trip time, jitter and packet loss covers the link figures a locally predicted body's projection depth is measured against.

ProjectedRigidbody

Driving the core API directly? See PhysicsConvergence.

`ProjectedRigidbody` is a `MonoBehaviour` that drives physics sync for one rigidbody. It requires a `Rigidbody` and a `NetworkSystemObject`, declares its need for the `NetworkSystem` carrying that body's replicated `NetworkPhysicsComponent`, and steps the body each tick: proxies adopt, follow the interpolation buffer, or run its `PhysicsConvergence` follower before the world simulates, and the server captures the stepped result afterward.

Runtime surface

Member	Type	Notes
`Convergence`	`PhysicsConvergence` (readonly)	The convergence follower this component owns. Tune it through `Convergence.Settings`, which the inspector fields below write into.
`SmoothedVisual`	`Transform`	Optional render transform, normally a child of the body that holds its renderer, interpolated between physics steps for tick-rate rendering.
`RemoteExtrapolationEnabled`	`bool` (default `true`)	On, a remote-controlled proxy extrapolates forward from its last received state: more current, but overshoots on sudden client state changes. Off, it interpolates between received states in remote time: smooth, never overshoots, always a step behind.
`Body`	`UnityPhysicsBody` (read-only)	The adapter around the attached `Rigidbody`.
`PhysicsComponent`	`NetworkPhysicsComponent` (read-only)	The replicated physics component this body captures into or converges from.
`NetworkSystem`	`NetworkSystem` (read-only)	The system this body belongs to; its controller state decides which peer's simulation is treated as ground truth.

Divergence thresholds

These gate both the predicted and the proxy correction paths.

- **Position Threshold** (`_positionThreshold`, meters) — the positional divergence past which the body

blends back toward the projected target. Below it, counter-velocity alone holds the offset steady.

- **Rotation Threshold** (`_rotationThreshold`, degrees) — the rotational divergence past which the body blends back toward the target.

Closing fields

How a body over its threshold corrects splits by role:

- **Blend Per Tick** (`_blendPerTick`) — the fraction of the remaining gap a *proxy* closes per tick while over threshold. A predicted body ignores this.
- **Prediction Correction Rate** (`_predictionCorrectionRate`, default `0.1`): how aggressively a *predicted* body (one whose `Convergence.IsLocallyPredicted` is set) corrects toward the server: the fraction of the gap taken off per tick, once past Position Threshold. Higher snaps back harder and sooner; lower lets the prediction ride looser. `ProjectedRigidbody` runs its follower only on a peer that is not the body's controller, so this field only acts where such a peer steers the body with its own client state and sets `IsLocallyPredicted`.
- **Blend Mode** (`_blendMode`, `ConvergenceBlendMode.Position` or `.Velocity`) — how the closing fraction is applied. `Position` writes the pose directly toward the target each step, identical to `Velocity` in free space, but bypasses the solver, so a blend into geometry penetrates and depenetrates rather than resolving as a contact. `Velocity` applies the same closing fraction as a solver-integrated velocity bias, so contacts, stacks, and other bodies are never disrupted — a blend against geometry becomes a bounded press instead of a penetration.

Residual fields

Below threshold, a sub-threshold residual can still be bled off rather than held frozen:

- **Residual Close Rate** (`_residualCloseRate`) — the per-second rate a sub-threshold positional residual is bled off by a velocity bias toward the target. Higher closes a slow body's drift faster, lower is gentler at first contact, zero holds the offset frozen.
- **Residual Close Maximum Speed** (`_residualCloseMaximumSpeed`) — the maximum body speed at which the residual close runs. A freshly bumped or still-moving body reads above it and falls back to the plain deadband, so the gentle close never fights a fresh contact.

Escape hatches

- **Teleport Distance** (`_teleportDistance`, meters, proxy-only) — the divergence past which a proxy snaps its whole pose onto the target instead of gliding; the smoothed visual jumps too. Zero disables it, so the body always blends however far off it is. A predicted body is exempt and never teleports.
- **Maximum Projection Time** (`_maximumProjectionSeconds`, seconds, proxy-only) — the maximum time a proxy's projection may lead the last received state. A predicted body spans its whole round trip and ignores this field entirely.
- **Gravity Enabled** (`_gravityEnabled`, default on) — whether gravity participates in the projection. On, a falling body is carried down under gravity while resting or supported bodies are left alone, so they are never projected into the ground. Off, the projection ignores gravity entirely.

Visual smoothing changes the render, never the body

- **Visual Smoothing** (`_visualSmoothing`, seconds, default `0`) — low-passes `SmoothedVisual` toward the interpolated physics pose to hide tick-to-tick correction jitter. Requires a `SmoothedVisual`; without one it has nothing to act on. The physics body itself stays exact — captured state and collisions are

untouched, so nothing can desync — only the render eases toward it. Larger smooths harder but lets the render trail the body more; zero, the default, is off.

How the fields reach the follower

At bring-up (`Awake`, and again on every inspector edit) most of these fields are written into a `PhysicsConvergenceSettings` and assigned to `Convergence.Settings`. Prediction Correction Rate is the exception: it is assigned straight to `Convergence.PredictionCorrectionRate` instead, since it lives on the follower itself rather than in `Settings`.

The maths behind each of these values, and the full settings surface, live on the PhysicsConvergence API page — including `PredictionMaximumProjectionSeconds`, the predicted-body counterpart to Maximum Projection Time, which has no inspector field here at all.

Methods you can call

`ProjectedRigidbody` exposes three public methods, all already called for you by the `PhysicsSimulationDriver` and the spawn handler:

- `PreSimulate(float stepDelta)` — the pre-step half of a tick-aligned world step. Proxies adopt, follow, or run the follower here; authorities do their work post-step.
- `PostSimulate(uint tick)` — the post-step half. A server captures the freshly stepped state into its replicated component, and the step-pose window advances for the smoothed visual's render interpolation.
- `NotifySceneChanged()` — re-resolves the physics driver after this body's `GameObject` moves to another scene. A scene move fires no disable or enable, and the driver is cached at enable, so without this the body keeps stepping the world it left.

You will not normally call any of these directly.

Replicate a rigidbody

> **Driving the core API directly?** See Replicate a physics body.

Add `ProjectedRigidbody` to a GameObject that has a `Rigidbody`. That's the whole setup.

Adding ProjectedRigidbody pulls in two more components

`ProjectedRigidbody` carries two `[RequireComponent]` attributes, so adding it pulls in the other two components you need:

- **Rigidbody** — the physics body itself. `ProjectedRigidbody` never simulates on its own; it steps and reads Unity's own rigidbody.
- **NetworkSystemObject** — the marker that gets this object a `NetworkSystem` to belong to.
- **ProjectedRigidbody** — the component you actually add, which ties the two together.

You never rent or spawn anything by hand. In its own `Awake`, `ProjectedRigidbody` calls `NetworkSystemObjectPool.RequireSystem<NetworkSystem, NetworkPhysicsComponent>`, declaring that this object needs a system carrying a replicated `NetworkPhysicsComponent`. Once that requirement is satisfied, three properties are populated for you to read:

- `NetworkSystem` — the system this body belongs to; its controller state decides which peer's simulation is treated as ground truth.
- `PhysicsComponent` — the replicated `NetworkPhysicsComponent` this body captures into or converges from.
- `Body` — a `UnityPhysicsBody`, the adapter wrapping your `Rigidbody`.

Nothing else needs wiring. No manual `RequireSystem` call, no manual component lookup.

Exactly one peer simulates the body for real

Exactly one peer simulates this body for real: the one where `NetworkSystem.IsController(ControllerType.AnyController)` is true. Every tick, that peer's `Rigidbody` steps normally, and the stepped result is captured into `PhysicsComponent` for replication.

Every other peer's `Rigidbody` does not run free simulation. Instead it follows: it adopts the first state it receives outright (so a freshly spawned proxy doesn't glide in from the prefab pose), then either follows the interpolation buffer as a kinematic body (for a proxy driven by a remote controller's own client state) or runs the convergence follower against the freshest snapshot, projected forward toward present time. The Rigidbody on a non-controlling peer is still a real, simulated body doing collision response: it's being steered toward the server's target, not teleported onto it.

Smoothing the render without touching physics

`Smoothed Visual` takes a child `Transform` of the body that holds its renderer, such as a child named `Visual`. It interpolates between the last two stepped poses in `LateUpdate`, hiding tick-rate stepping behind frame-rate rendering. Because only that child moves, the physics body's own pose is never touched by this smoothing, so nothing here can desync a collision or a captured state.

`Visual Smoothing` (seconds) additionally low-passes that visual toward the interpolated pose, easing out the jitter a correction leaves. Leave it at `0` on a body that should not trail — a fast-falling object that needs its visual pinned exactly to the physics pose.

Don't also add NetworkTransform

This object replicates its pose through `ProjectedRigidbody`'s own `NetworkPhysicsComponent`. Adding `NetworkTransform` to the same object as well double-replicates the pose through two independent systems. See the transform-replication choosing page for which component fits which object.

A `PhysicsSimulationDriver` present in this body's scene is what puts the body on the tick cadence: it calls the pre- and post-step halves around the world step so capture and follow line up with ticks. You don't have to add one yourself. A `UnityPhysicsManager` is added to the manager graph automatically, and under its default execution mode, `PhysicsExecutionMode.NucleusPerTick`, a `PhysicsSimulationDriver` for the default physics world is added with the managers too, so a body in an ordinary scene is on the tick from the start. A stacked local-physics scene gets its own driver when it loads. Under `PhysicsExecutionMode.UnityRun` the default world is left to Unity and projected physics is unavailable there; see Physics execution modes.

Check the result across two editors

Run two editors. On the controlling peer, move or nudge the `Rigidbody` — push it, let it fall, whatever your scene does. On the second editor, the same `Rigidbody` follows that motion: not by teleporting to each new position, but converging toward it tick by tick.

Before any state has arrived at all, the second editor's copy sits at its prefab pose and does not move — there's nothing to follow yet, so the first received state is what gets adopted outright.

NetworkTransform or ProjectedRigidbody

Two different models

`NetworkTransform` and `ProjectedRigidbody` both replicate a moving object, and they do it in incompatible ways.

`NetworkTransform` makes every non-controlling peer a **pose follower**. The controller writes position, rotation, and scale each tick; every other peer applies the received values onto the transform through the interpolation buffer, animated one tick behind fact. Velocity is never synchronized — a proxy has no physical motion of its own, it is just redrawn along the received path.

`ProjectedRigidbody` keeps every peer **simulating**. The controller's `Rigidbody` steps normally and the stepped result is captured for replication. Every other peer's `Rigidbody` also steps — real collision response, real physics — while the convergence follower (`PhysicsConvergence`) steers it toward the server's snapshot, blending a diverged position or rotation back once it passes a threshold instead of teleporting onto it.

Pick by what a proxy needs to do: look right, or behave right. A pose follower is cheaper and never fights local physics. A converging body can be bumped, land on, or collide with on every peer, because it's actually simulating there.

Never both on one object

Adding both to the same GameObject is not redundant, it's broken. `NetworkTransform`'s Kinematic Management (on by default) holds an attached `Rigidbody` kinematic on every peer that isn't the controller — including the server while a client controls the system — so the interpolated pose writes don't fight local physics. That's exactly the body `ProjectedRigidbody` needs to step.

`PhysicsConvergence.Step` checks `physicsBody.IsKinematic` first and returns immediately if it's true. So on a non-controlling peer with both components: `NetworkTransform` sets the rigidbody kinematic, and `ProjectedRigidbody`'s follower short-circuits on every call and never runs. The body goes inert on every proxy — it sits wherever `NetworkTransform` last interpolated it to, never converging, never colliding correctly.

An object replicates its pose through one system or the other, never both.

Choosing by object

- **Player capsules, doors** — `NetworkTransform`. The pose is driven by client state or a controller, proxies don't need to be physically bumpable, and pose-follow interpolation is cheap and smooth.
- **Thrown and bumped bodies** — `ProjectedRigidbody`. These need to collide correctly on every peer — a thrown grenade or a kicked prop has to bounce off geometry the same way for everyone watching, not just the controller.
- **Debris nobody steers** — `ProjectedRigidbody`, or nothing at all if it never needs to agree across peers. If it's not gameplay-relevant, it may not need replication at all.
- **Agents that walk a path** — neither. See NetworkNavMeshAgent — a path-following agent replicates the leg it's walking, not a per-tick pose or a physics body.

Cost, in outline

`NetworkTransform` replicates position, rotation, and (optionally) scale as ordinary members.

`ProjectedRigidbody` replicates through `NetworkPhysicsComponent`, and additionally spends `PhysicsConvergence.Settings` steering: threshold, blend rate, teleport distance, and the rest, tuned per body rather than sent per tick.

Both go through the same per-member choice of `TransmissionMode` — `Divine` or `Interval` — and, under `Interval`, `SendInterval`. `Divine` can go quiet for stretches where `Interval` would still be sending deltas, at Pro-only cost; Free packs every length exactly, Pro packs more tightly. See How Often a Member Is Sent for what each mode actually costs and how to choose between them — that decision doesn't differ between the two components.

Setup cost

A `NetworkTransform` object needs no driver and no `UnityPhysicsManager` decision. It's self-contained: add the component to an object carrying a `NetworkSystemObject`, and it works.

A `ProjectedRigidbody` object needs both a `Rigidbody` and a scene physics story. A `UnityPhysicsManager` is added to the manager graph automatically, and under its default `PhysicsExecutionMode.NucleusPerTick` so is a `PhysicsSimulationDriver` for the default physics world, so capture and follow already run on the tick cadence without adding a driver by hand. See PhysicsSimulationDriver and Replicate a rigidbody.

PhysicsSimulationDriver

PhysicsSimulationDriver ties one scene's physics to the network tick

`PhysicsSimulationDriver` is a `MonoBehaviour` that ties one scene's physics to the network tick. A scene with a driver steps its bodies on `NetworkLoopSteps.EarlyFixedUpdate` and steps its physics world on `NetworkLoopSteps.LateFixedUpdate`, instead of on Unity's own `FixedUpdate` cadence. Received state deserializes earlier in the same frame and the tick's captures serialize later in it, so captures and transmissions share one clock instead of drifting against Unity's own physics timing.

A driver is per scene. It registers itself with `PhysicsSimulationCoordinator`, the process-wide object that owns the one network-loop registration, the simulation mode, and the peer's tick anchor. A scene is stepped by exactly one driver.

Scenes that share a physics world (every scene not loaded with a local physics world of its own) are stepped once between them. A driver per scene is not the same thing as a physics world per scene: multiple drivers can point at the same world, and the coordinator steps that world once per tick regardless of how many drivers resolve to it.

Events this component raises

Event	Signature	Fires
`WorldStepStarted`	`WorldStepStartedHandler(uint tick, float stepDelta)`	Before each world step. This is where gameplay applies per-step forces from client state on the simulated cadence, since Unity's own `FixedUpdate` no longer matches when the world actually steps.
`BodyRegistered`	`BodyRegisteredHandler(ProjectedRigidbody projectedRigidbody)`	When a body joins the driver's step set.
`BodyDeregistered`	`BodyDeregisteredHandler(ProjectedRigidbody projectedRigidbody)`	When a body leaves the driver's step set.

```
driver.WorldStepStarted += (tick, stepDelta) =>
{
    // Apply per-step forces from client state here.
};
```

`BodyRegistered`/`BodyDeregistered` exist so a diagnostic, editor tool, or convergence monitor can follow the live body set without walking the scene for it every frame. Subscribe and then read `Bodies` to adopt what already exists, otherwise anything attaching after the first spawn misses every body already present.

Values you can read

- `CurrentTick` (`uint`) - the network loop tick currently being simulated.
- `SubtickFraction` (`float`) - how far the render frame has progressed through the current tick, for

phasing smoothed visuals between the last two stepped poses.

- `Bodies` (`IReadOnlyList<ProjectedRigidbody>`) - the bodies this driver steps, in registration order. A diagnostic reads this instead of walking the scene.
- `TryGetSnapshotLocalTick(uint captureTick, out uint snapshotLocalTick)` - maps a received capture tick into local ticks, through the session's shared drift-adaptive anchor. Returns `false` when the capture tick carried no attribution or nothing mapped.

Finding the driver for a scene

`ForScene(Scene scene)` resolves the driver stepping a given scene's physics, or `null` when nothing steps it (the object then runs Unity's own physics cadence). Resolve once and hold the result: this is a scene read plus a dictionary probe, not a field read, and it is also what lets a later deregistration reach the same driver.

```
PhysicsSimulationDriver driver = PhysicsSimulationDriver.ForScene(gameObject.scene);
```

For a driver that may not exist yet - it can be added at runtime rather than authored into the scene, and component `Start` order between two objects is unspecified - use `RegisterForScene`/ `UnregisterForScene` instead:

```
PhysicsSimulationDriver.RegisterForScene(gameObject.scene, OnDriverReady);

void OnDriverReady(PhysicsSimulationDriver driver) { /* ... */ }

// later
PhysicsSimulationDriver.UnregisterForScene(gameObject.scene, OnDriverReady);
```

`DriverReadyHandler` is `void DriverReadyHandler(PhysicsSimulationDriver physicsSimulationDriver)`. If a driver already steps the scene, the callback fires immediately rather than waiting for a registration edge that already passed - a subscriber attaching after the driver registered would otherwise wait forever.

Register / Deregister

`Register(ProjectedRigidbody projectedRigidbody)` and `Deregister(ProjectedRigidbody projectedRigidbody)` add and remove a body from the driver's step set. `Register` is idempotent.

Two drivers cannot claim the same scene. If a second driver enables on a scene a driver already steps, it logs an error naming both GameObjects and does nothing - the first driver keeps stepping the scene, the second driver's bodies are simply never registered to it.

This component is Unity only

Unity only. There is no plain-C# counterpart: physics stepping is a Unity concept.

Project assets and loaders

Prefabs

NetworkPrefabCollection

A `ScriptableObject` (`Nucleus/Network Prefab Collection` in the Create menu) that lists the networked prefabs shipped with one content shard. The base build ships a default shard, and each asset bundle ships its own alongside the prefabs it carries.

- `_prefabs` — the prefabs in the shard, exposed as `Prefabs`.
- `_prefabBundleId` — the bundle identifier this shard registers its prefabs under; zero is the base build's default shard. Exposed as `PrefabBundleId`, stamped by the editor scan from the shard assembly's `[NetworkBundle]` attribute.
- `_bundleAssemblyName` — the simple name of the assembly whose `[NetworkBundle]` attribute supplies that identifier; empty for the base build's default shard. Exposed as `BundleAssemblyName`.

`RegisterAll()` claims the shard's bundle identifier and registers every prefab; `UnregisterAll()` removes them and releases the claim, called as the shard's content unloads. Two loaded collections claiming the same bundle identifier is an error, not a silent overwrite.

The base build's collection is loaded from `Resources/NetworkPrefabCollection` automatically at startup. A project with no prefab collection there registers nothing, and a dynamic spawn for a prefab identity nobody registered fails to resolve on the receiving peer.

NetworkPrefabRegistry

The runtime lookup a collection registers into, keyed by `(BundleId, LocalId)`. A shard registers under its own bundle id, so separately built content never collides. `NetworkPrefabRegistry.TryGet` is what a spawn resolves its prefab through; a spawn whose identity has no entry cannot instantiate.

The prefab pool seam

Every rent and return of a dynamic spawn's GameObject goes through `INetworkPrefabPool`:

```
public interface INetworkPrefabPool
{
    GameObject Rent(GameObject prefabGameObject, ushort prefabBundleId, ushort prefabLocal
Id, Vector3 position, Quaternion rotation, Vector3 scale);
    void Return(GameObject pooledGameObject, ushort prefabBundleId, ushort prefabLocalId);
}
```

Two implementations ship:

- **DefaultNetworkPrefabPool** — instantiates on rent and destroys on return. No pooling, and the default a project starts with.
- **NetworkPrefabPool** — recycles: a returned instance is deactivated and parked under a shared inactive root, keyed by prefab identity, and a later rent reactivates the longest-rested parked instance instead of instantiating. `ReturnCooldownSeconds` keeps a just-returned instance parked before it is eligible for reuse; `InstantiatedCount` and `ReusedCount` report the cold-miss and recycle counts.

Because a recycled instance does not re-run `Awake`, a script needing per-life setup binds through the marker's `SystemLinked`/`SystemUnlinked` events, or initializes in `OnEnable` instead.

The active pool is assigned through `NucleusUnity.PrefabPool`:

```
NucleusUnity.PrefabPool = new NetworkPrefabPool { ReturnCooldownSeconds = 1f };
```

Assigning `null` restores `DefaultNetworkPrefabPool`. A project that never touches this property keeps the default, instantiate-and-destroy behavior.

Scenes

NetworkSceneManifest

A `ScriptableObject` (`Nucleus/Network Scene Manifest` in the Create menu) mapping the scene identifiers the server names on the wire to the scene assets a peer loads them from, and the content bundle each scene ships in. It is also the identifier registry itself: a prefab is stamped on the prefab asset, but a scene has no comparable place to carry one, so the entry keyed by the scene's GUID is where the identifier lives.

`_sceneEntries` holds the list, exposed as `Entries`; each entry is a `SceneEntry`:

Field	Purpose
`SceneId`	The identifier the server names this scene by on the wire.
`SceneGuid`	The scene asset's GUID — what the identifier is really keyed to.
`ScenePath`	The scene asset's path, handed to Unity's loader.
`RequiredBundleId`	The content bundle the scene ships in; the base build's identifier means it ships in the player.
`DeliveryMode`	How the peer gets the scene (`SceneDeliveryMode`), derived from the project by the build step.

A project has one manifest, loaded by convention from `Resources/NetworkSceneManifest` (`NetworkSceneManifest.LoadProjectManifest()`), so nothing has to be dragged into an inspector.

NetworkSceneReference

A struct field type that names a networked scene by dragging the scene asset in, instead of typing its wire number by hand. It stores only the scene's **GUID** (`_sceneGuid`) — never the scene identifier itself — so renaming or moving the scene asset never repoints the reference or silently starts naming a different scene. The identifier is resolved through the project's manifest at the moment it's used (`TryGetSceneId`), so a scene added after the reference was authored resolves as soon as the manifest is rebuilt.

Handle-to-Scene lookups

Once a scene instance is loading, code resolves it through the `UnitySceneManager` component (the CoreManager's Unity-side scene driver), not by walking Unity's own `SceneManager` by name or path — several instances of the same scene asset can be loaded at once, and `GetSceneByName`/`GetSceneByPath` cannot tell them apart.

```
public bool TryGetScene(uint sceneHandle, out Scene scene);
public bool TryGetSceneHandle(Scene scene, out uint sceneHandle);
```

`TryGetScene` turns the server's scene instance handle into the live Unity `Scene`; it answers false while the scene is still loading. `TryGetSceneHandle` is the inverse, for code holding a scene or object that needs to know which instance it belongs to — a true result with `NetworkSystem.UnsetSceneHandle` means the

peer loaded that scene itself and no instance will ever claim it.

The correlation between a handle and its `Scene` is tracked internally; these two methods are the supported way to read it.

NetworkSceneLoader and UnitySceneLoader

`NetworkSceneLoader` is the abstract `MonoBehaviour` base for loading networked scenes: it implements `ISceneLoader` and registers itself as the peer's scene loader automatically on `Awake` (toggle `_automaticRegistrationEnabled` off to call `Initialize` explicitly). Loading and unloading are left to the derived class via `LoadSceneAsync`/`UnloadSceneAsync`.

`UnitySceneLoader` is the shipped implementation, built on Unity's own additive scene loading. It resolves a scene through a manifest, takes the scene's content bundle first if it needs one, then loads the scene additively.

- `_sceneManifest` — the manifest this loader reads. Left empty, it falls back to the project's own manifest, so nothing needs assigning for the common case.
- `_localPhysicsEnabled` — loads each scene into its own local physics world (`LocalPhysicsMode.Physics3D`) with a `PhysicsSimulationDriver` stepping it each tick, so a stacked scene's physics runs independently of the default world.

A project doesn't have to place a `UnitySceneLoader` itself: `UnitySceneManager` adds one automatically if no `NetworkSceneLoader` of any kind is present in the loaded scenes.

Content bundles

`NetworkBundleManifest` ships in every build. `NetworkBundleLoader` and `UnityAssetBundleLoader` are Pro-only files, absent entirely from a Free build, so a Free project can carry a bundle manifest but has nothing built in that loads against it.

NetworkBundleManifest

A `ScriptableObject` (`Nucleus/Network Bundle Manifest` in the Create menu) mapping the bundle identifiers the server names on the wire to the asset bundle files a client loads them from.

- `_bundleEntries` — the mapped bundles, exposed as `Entries`; each `BundleEntry` pairs a `BundleId` with a `FileName`.
- `_rootDirectory` — the directory file names resolve against; empty falls back to `Application.streamingAssetsPath`.

NetworkBundleLoader and UnityAssetBundleLoader (Pro)

`NetworkBundleLoader` is the abstract `MonoBehaviour` base for loading content bundles: it implements `IBundleLoader` and self-registers on `Awake`, the same way `NetworkSceneLoader` does for scenes. `LoadBundleAsync`/`UnloadBundleAsync` are left to the derived class.

`UnityAssetBundleLoader` is the shipped implementation over Unity asset bundles. Its `_bundleManifest` field maps bundle identifiers to files; left unassigned, a load request logs an error and fails rather than resolving anything. Once a bundle loads, it registers every `NetworkPrefabCollection` the bundle ships, so spawns waiting on that content can land.

What a project actually needs

A project with only base-build prefabs and scenes needs:

- One `NetworkPrefabCollection` at `Resources/NetworkPrefabCollection` (registers automatically).

- One `NetworkSceneManifest` at `Resources/NetworkSceneManifest`, if it uses networked scenes at all.

Everything else is optional:

- **Prefab pool** — unassigned, spawns use `DefaultNetworkPrefabPool` (instantiate/destroy). Assign a `NetworkPrefabPool` through `NucleusUnity.PrefabPool` only to recycle instances.
- **Scene loader** — a project doesn't have to add one; `UnitySceneManager` provides a `UnitySceneLoader` automatically. Add one manually only to override the loading pipeline.
- **Content bundles** — only needed once the project ships prefabs or scenes in asset bundles rather than the base build. A `UnityAssetBundleLoader` with no manifest assigned fails every bundle load with a logged error, rather than silently doing nothing.

Addressables-delivered scenes

What delivery mode means

Every scene the server can send a peer to is one row in the project's `NetworkSceneManifest`, built by `Nucleus/Rebuild Network Scene Manifest`. Each `NetworkSceneManifest.SceneEntry` carries a `SceneDeliveryMode`, derived from the project rather than chosen by hand:

- `SceneDeliveryMode.BuildSettings` — the scene ships inside the player and is opened by path. This is the default for any scene not marked Addressable, and it is what every scene mapped before delivery modes existed still reads as.
- `SceneDeliveryMode.Addressables` — the scene is an Addressables entry, opened through Addressables, which resolves and fetches whatever carries it.

Making a scene Addressable in the project and rebuilding the manifest is the whole of the change; nothing else in the scene's setup differs between the two modes.

Why scenes don't ship in raw asset bundles

`SceneEntry` also has a `RequiredBundleId`, and `UnitySceneLoader.EnsureSceneContentAsync` does honor it if it's set to a non-base-build value — that plumbing exists. But the shipped tooling never actually produces a scene entry pointing at a raw asset bundle.

`NetworkSceneManifestBuilder`'s `ResolveLegacyBundleId` always resolves a scene's bundle id to `Connection.BaseBuildBundleId`, and logs an error instead if the scene's importer has an asset bundle name assigned. The reason is a Unity limitation: it cannot pack a scene and an asset into one bundle, and a scene-only bundle has no assembly to carry an identifier. If the build finds a scene assigned to the same asset bundle name as a `NetworkPrefabCollection`, or assigned to any asset bundle name at all, it reports the conflict and maps the scene as base build anyway, so the scene still loads correctly out of the player. This applies only to scenes the build ships or the project has made Addressable. A scene in neither is not networked and is ignored, whatever asset bundle name it carries.

Net effect: for scene content, `RequiredBundleId` is never anything but the base build id. Addressables is the only supported way to stream a scene's content ahead of the load.

Turning it on

1. Mark the scene Addressable in the project (its own group, address, whatever the project's Addressables setup calls for).
2. Rebuild the manifest with `Nucleus/Rebuild Network Scene Manifest`. The builder finds the scene among the project's Addressable GUIDs and records `DeliveryMode = SceneDeliveryMode.Addressables` for it.
3. Make sure `com.unity.addressables` 1.19 or newer is installed. The integration's assembly definition then sets the `NUCLEUS_ADDRESSABLES` define for you through a `versionDefines` entry, so there is no scripting define symbol to add by hand. The Addressables code path in `UnitySceneLoader` is compiled out without it.

The Addressables path itself ships in the non-Pro `UnitySceneLoader.cs`. It is not gated by edition — only fetching content out of an asset bundle is.

How the load actually runs

`UnitySceneLoader.LoadSceneCoreAsync` resolves the manifest entry, then calls `EnsureSceneContentAsync(sceneEntry.RequiredBundleId)` before it does anything else — content has to be held before a scene can be loaded out of it. `EnsureSceneContentAsync` calls the protected `LoadSceneContentBundle` partial method hook.

Without the Pro assembly present, that hook has no implementation, so it does nothing and the load proceeds straight to the scene. `UnitySceneLoader.Bundles.Pro.cs` supplies the Pro implementation: for a scene whose `RequiredBundleId` is `Connection.BaseBuildBundleId` it also does nothing (the common, base-build case pays nothing), and only for a non-base-build id does it ask `CoreManager.BundleManager.LoadBundleAsync` to fetch the bundle. Since the manifest builder never assigns a scene a non-base-build id, this hook is a no-op for every scene in practice — Addressables content isn't fetched through it.

Once content is confirmed (or, for `BuildSettings` scenes, immediately), `LoadSceneCoreAsync` branches on `sceneEntry.DeliveryMode`. `SceneDeliveryMode.Addressables` routes to `LoadAddressableSceneCoreAsync`, which calls `Addressables.LoadSceneAsync` keyed by the scene's GUID (`sceneEntry.SceneGuid`), additive, with local physics settings carried over the same way the build-settings path does.

The one gotcha: failed loads don't throw

A failed Addressables load does not throw and does not fault the returned task. An unresolvable key comes back as a handle that is already complete with a failed status. `LoadAddressableSceneCoreAsync` checks `loadAsyncOperationHandle.Status is not AsyncOperationStatus.Succeeded` explicitly and logs an error if so — that check is the only way the failure is ever noticed. Anything that calls into this path without checking the handle's status will treat a failed load as if it succeeded.

The same method also checks that the loaded scene's path matches the manifest's `ScenePath`, because Addressables names a scene by its asset path only while the group keeps the default internal naming; a group set to GUID or Dynamic naming breaks the match silently otherwise.

Content bundles in Unity

> **Driving the core API directly?** See The content bundle protocol.

Content bundles let you ship a set of networked prefabs outside the base build and load them into a running game as an asset bundle. Clients that have not yet loaded a bundle simply do not receive spawns for the prefabs it ships — the server withholds them until the bundle is loaded and registered.

Shard your prefabs

A shard is one `NetworkPrefabCollection` asset. Which prefabs belong to it is decided by Unity's own asset bundle assignment: the collection assigned to asset bundle `x` owns every networked prefab assigned to `x`, and the collection assigned to no bundle owns the prefabs assigned to none — the base build.

Each collection exposes:

- `PrefabBundleId`: the bundle identifier this shard's prefabs register under. You never author this directly; it is stamped at rebuild time from `BundleAssemblyName`.
- `BundleAssemblyName`: the simple name of the assembly whose `[assembly: NetworkBundle(n)]` attribute supplies the bundle identifier. Empty means the base build's default shard, bundle id `0`.

Each separately built assembly that ships its own content declares its own bundle identifier:

```
[assembly: NetworkBundle(1)]
```

`NetworkBundleAttribute` is assembly-scoped (`AttributeUsage(AttributeTargets.Assembly)`). The main assembly defaults to bundle `0` when the attribute is absent. A named assembly that isn't loaded, or that carries no `NetworkBundleAttribute`, resolves to the default shard (bundle `0`) as well, with an editor error logged either way — so give every other bundle its own id rather than relying on that fallback.

Build the shards

Two menu items, run in order:

Nucleus > Rebuild Network Prefab Collection scans the project for prefabs carrying a `NetworkSystemObject`, groups them by the asset bundle they're assigned to, and stamps each prefab's local `PrefabId` plus the shard's `PrefabBundleId`. Identifiers are preserved across rescans, never renumbered from position, so a bundle built before a change keeps resolving correctly against a client that rebuilt after it.

Nucleus > Build Content Bundles rebuilds the collections (so identity is never stale), builds every authored asset bundle into `Assets/StreamingAssets`, and then fills in the `NetworkBundleManifest` from what was actually produced.

If your project has no `NetworkBundleManifest` yet, the build creates one and reports where. If it finds more than one, it leaves them all alone and warns, because it can't tell which owns the content it just built — keep a single manifest so this step stays automatic.

The manifest

`NetworkBundleManifest` maps the bundle identifiers the server names on the wire to the files a client loads them from:

- `Entries`: a list of `BundleEntry { BundleId, FileName }`.
- `RootDirectory`: the directory file names resolve against; falls back to `Application.streamingAssetsPath` when left unset, which is `Assets/StreamingAssets` in the

editor and the shipped copy in a build.

- `TryGetBundlePath(ushort bundleId, out string path)`: resolves a bundle identifier to its full file path, combining `RootDirectory` with the matching entry's `FileName`.

You don't hand-author entries. **Build Content Bundles** writes them for you, upserting by identifier so an entry you keep for content loaded through some other pipeline survives.

Load bundles at runtime

Add a `UnityAssetBundleLoader` component to a GameObject in your bootstrap scene and assign its **Bundle Manifest** field to your `NetworkBundleManifest` asset.

`UnityAssetBundleLoader` derives from `NetworkBundleLoader`, which self-registers as the CoreManager's bundle loader. With `_automaticRegistrationEnabled` on (the default), this happens in `Awake` by resolving the integration's `CoreManager`, so ordinary game code can request a bundle from its own `Start` and find a loader already registered. Turn that flag off and call `Initialize(CoreManager)` yourself if your game owns more than one `CoreManager` or constructs one later.

`UnityAssetBundleLoader.LoadBundleAsync(ushort bundleId)` resolves the file through the manifest, loads the Unity asset bundle, and registers every `NetworkPrefabCollection` it finds inside — all before the returned task completes, since completing tells the server this client may now be spawned into.

`TryGetBundlePath` on `UnityAssetBundleLoader` is `protected virtual` specifically so a project can override it to resolve a bundle's path from a patcher or a downloaded cache instead of `StreamingAssets`, while keeping the rest of the loader's behavior.

Overriding a prefab's required bundle

Normally a prefab's required content is read straight from the shard it shipped in. Occasionally a prefab needs a different bundle than the one that ships it — content the editor scan couldn't stamp, or an object whose prefab ships in the base build while the assets it needs do not.

`NetworkSystemObject` exposes this as an override:

- **Override Required Bundle** (`_requiredBundleOverrideEnabled`): turns the override on. Left off, the requirement comes from the prefab's own `PrefabBundleId`.
- **Required Bundle Id** (`_requiredBundleId`): the bundle identifier required when the override is enabled. The inspector only shows this field while the override is on.

`NetworkSystemObject.RequiredBundleId` resolves between the two: the override value when enabled, otherwise `PrefabBundleId`. This never changes the prefab's own wire identity — that stays `PrefabBundleId` paired with `PrefabId` — it only changes which bundle must be loaded before the server will spawn it.

Content bundles ship only in Pro

Content bundles are a Pro feature. `CoreManager.BundleManager` lives in a Pro-only partial file; a Free build has no such field at all, and the runtime loaders (`NetworkBundleLoader`, `UnityAssetBundleLoader`) exist only to register against it. A Free project builds and ships without content bundles.

Networked Scenes, Content Bundles and Persistence

Pages in this category:

- The network scene manifest and scene references
- UnitySceneLoader component
- UnitySceneManager component
- How networked scenes work
- Networked scenes in Unity
- Content bundles in Unity
- Objects authored into a scene
- Scenes and bundles: things that look broken
- Addressables-delivered scenes
- Saving and loading a world from Unity

The network scene manifest and scene references

Why scenes need a manifest

The wire never carries a scene's path or name, only a `ushort` identifier. Something has to map that number back to an actual scene asset on each peer, and that something is `NetworkSceneManifest`, a `ScriptableObject` under `Nucleus.Integrations.Unity.Scenes`. It is also the identifier registry itself: a networked prefab carries its id stamped on the prefab asset, but a scene has nowhere comparable to hold one, so the manifest entry keyed by the scene's GUID *is* where the id lives.

A project has exactly one manifest. It lives under a `Resources` folder and is resolved by convention, not by being dragged into an inspector.

NetworkSceneManifest

`NetworkSceneManifest` exposes:

- `Entries` — the `IReadOnlyList<SceneEntry>` of every networked scene this peer knows.
- `UnsetSceneId` — the constant `0`, reserved so a lookup that failed can be told apart from "the first scene."
- `ResourceName` — the constant `"NetworkSceneManifest"`, the file name the manifest must have under a `Resources` folder.
- `TryGetScene(ushort sceneId, out SceneEntry sceneEntry)` — resolves an entry from the id the server named on the wire.
- `TryGetSceneId(string scenePath, out ushort sceneId)` — resolves an id from a scene's asset path.
- `TryGetSceneIdByGuid(string sceneGuid, out ushort sceneId)` — resolves an id from a scene's GUID; prefer this one, since it is what the id is actually keyed to.
- `LoadProjectManifest()` — a static method that loads the project's manifest via `Resources.Load<NetworkSceneManifest>(ResourceName)`. Nothing is cached: a manifest held in a static would keep answering for an asset a later session replaced.

```
NetworkSceneManifest manifest = NetworkSceneManifest.LoadProjectManifest();

if (manifest != null && manifest.TryGetScene(sceneId, out NetworkSceneManifest.SceneEntry
entry))
{
    // entry.ScenePath is what you hand to Unity's scene loader.
}
```

SceneEntry

Each `SceneEntry` in `Entries` carries:

Field	Purpose
`SceneId`	The identifier the server names this scene by on the wire. Assigned once by the build and never renumbered.

`SceneGuid`	The scene asset's GUID. The identifier is keyed to this, so a rename or move keeps it.
`ScenePath`	The scene asset's path, which is what the loader hands to Unity.
`RequiredBundleId`	The content bundle this scene ships in, or zero when it ships inside the player build. A scene in a bundle can't load until that bundle is held.
`DeliveryMode`	A `SceneDeliveryMode`, either `BuildSettings` or `Addressables`, deciding which machinery the loader opens the scene through.

NetworkSceneReference

Fields that need to name a scene do not store a path or an id. They store a `NetworkSceneReference`, a serializable struct that holds only a scene asset's GUID (`SceneGuid`), plus `HasScene` (true once a GUID is assigned) and `TryGetSceneId(out ushort sceneId)`.

`TryGetSceneId` resolves through the project's manifest at the moment it's called, via `TryGetSceneIdByGuid`. Nothing is baked in at authoring time, so a scene added to the manifest after the reference was set resolves as soon as the manifest is rebuilt, and a scene that stopped being networked resolves to `UnsetSceneId` instead of silently opening the wrong thing.

```
[SerializeField]
private NetworkSceneReference _lobbyScene;

if (_lobbyScene.TryGetSceneId(out ushort sceneId))
{
    // Tell the server to open sceneId.
}
```

Because only the GUID is stored, renaming or moving the scene asset never repoints the field to a different scene and never breaks it.

In the inspector, `NetworkSceneReferenceDrawer` draws a `NetworkSceneReference` field as an ordinary scene-asset picker (an `ObjectField` for `SceneAsset`), with the resolved scene id shown alongside it as a small label. You drag a scene in exactly as you would for a plain `SceneAsset` field; the GUID is captured for you.

Minting a scene id

Scene ids are not assigned by hand. **Nucleus > Rebuild Network Scene Manifest** rescans the project and writes `Entries`, and `NetworkSceneManifestAutoRebuild` runs that rebuild automatically. It listens for four triggers: the Build Settings scene list changing, a scene asset being imported, the project's Addressables entries changing, and a player build starting (the last is the backstop, catching anything the other three missed before it ships).

The scan covers both ways a scene can be networked: scenes listed in Build Settings, and scenes the project has made Addressable. The Addressables pass matters because marking a scene Addressable *removes it from Build Settings* — without a dedicated pass, the very act of making a scene streamable would drop it out of the manifest. Whichever way a scene is found, its `DeliveryMode` is derived from the project (Addressable scenes record `SceneDeliveryMode.Addressables`, everything else records `BuildSettings`), so making a scene Addressable is the whole of the change needed to switch it.

Once assigned, a scene's id is preserved across rebuilds; a rebuild only adds ids for newly-found scenes and drops entries for scenes no longer present.

Why only a ushort crosses the wire

The server names a scene by `SceneId` alone. No path, no name, no build index, and how the scene is delivered (`DeliveryMode`, `RequiredBundleId`) is never replicated either — those are purely how the local peer resolves the id into something it can load. This is deliberate: paths and names change with renames and refactors, but the manifest that maps a stable id to a stable GUID does not need to, so a build can point at different content, or ship a renamed scene, without ever touching what's on the wire.

Networked scenes in Unity

Driving the core API directly? See Opening and closing scene instances.

Add the manager and loader

`UnityCoreManager` adds a `UnitySceneManager` with default settings to its own GameObject if that GameObject has none. To change its settings, add one yourself on that same GameObject, not on a child object: `UnityCoreManager` only looks on its own GameObject before adding one, so a copy on a child leaves you with two. It drives the core `SceneManager` for you: on `ManagersInstantiated` it copies its `Join Placement`, `Automatic Request On Blocked Spawn Enabled` and `Scene Load Timeout Seconds` fields onto the core manager, then makes sure some scene loader exists.

That last step, `EnsureSceneLoader`, only fires when `Scene Loader Enabled` is on and no `NetworkSceneLoader` is found anywhere in the loaded scenes (inactive objects count). If it finds none, it adds a `UnitySceneLoader` component to its own GameObject and sets its `LocalPhysicsEnabled` from `Automatic Stacked Scene Simulation Enabled`. If a loader already exists, `UnitySceneManager` leaves it alone.

A `UnitySceneLoader` registers itself: `NetworkSceneLoader.Initialize(CoreManager)` runs from `Awake` whenever its `Automatic Registration Enabled` toggle is on, which is the default. Turn that off only if you construct or register the loader yourself, later, with your own call to `Initialize`. Either way, `UnitySceneManager` never binds the loader to the manager itself - registration is the loader's own job.

For a first pass, leave both defaults on: add `UnitySceneManager`, let it add a `UnitySceneLoader`, and move on.

Give the scene a wire id

Add your second scene to Build Settings. Then run **Nucleus > Rebuild Network Scene Manifest** from the menu - this is what stamps every scene in Build Settings with an id in the project's `NetworkSceneManifest` and is what `UnitySceneLoader` resolves ids against. In practice you rarely have to run it by hand: `NetworkSceneManifestAutoRebuild` hooks Unity's build scene list and rebuilds the manifest whenever it changes, so the menu command is a fallback for when that hasn't happened yet.

`UnitySceneLoader` has a `Scene Manifest` field, but its tooltip says it plainly: leave it empty. An empty field resolves to the project's own manifest automatically, and assigning one only makes sense if your project deliberately keeps more than one manifest. For a single-manifest project, don't touch this field at all.

Open an instance from a script

The manager exposes the core scene manager it drives as `NucleusSceneManager` on the `UnitySceneManager` component:

```
ushort sceneId; // the id the manifest gave your scene
if (unitySceneManager.NucleusSceneManager.TryOpenScene(sceneId, SceneScope.Connections, ou
t uint sceneHandle))
{
    unitySceneManager.NucleusSceneManager.TryRequestSceneLoad(connection, sceneHandle);
}
```

`TryOpenScene` only opens the instance on the server - it records the handle and, for `SceneScope.Connections`, places nobody. Nothing loads on a client until you ask for it with `TryRequestSceneLoad(connection, sceneHandle)`, which is the step that actually tells that

connection to load the scene.

Skip `TryRequestSceneLoad` and nothing goes wrong loudly: the instance exists on the server, but no client was ever told to load it, so the scene simply never appears on the other end. There's no error for it, because from the engine's side you never asked.

Join placement only covers new joins

`SceneScope.Global` places every currently connected client the moment you open the instance, and places each later client as it authenticates. `Join Placement` (`UnitySceneManager`'s field, `JoinPlacement` on the core manager) governs only the join-time moment: which of the scenes already open a freshly authenticated client is dropped into.

So a scene you open after clients have joined reaches them only if it is `SceneScope.Global`. A `SceneScope.Connections` scene opened after a client joined is never offered to that client by `Join Placement`, even under the default `EveryOpenScene`, and needs its own explicit `TryRequestSceneLoad` per connection you want in it.

Where the two halves live

The inspector fields you just set - `Scene Loader Enabled`, `Join Placement`, `Automatic Request On Blocked Spawn Enabled`, `Scene Load Timeout Seconds`, `Automatic Stacked Scene Simulation Enabled`, and the loader's `Scene Manifest` and `Local Physics Enabled` - are documented on the `UnitySceneManager` reference page. The calls you made from script - `TryOpenScene`, `TryRequestSceneLoad`, and the rest of the core `SceneManager` API - are documented on the scene API pages under Core API.

How networked scenes work

Three identifiers, never conflated

A networked scene involves three different numbers, and mixing them up is the most common source of confusion:

- **Scene id** (`ushort`) — which scene asset. The same id always resolves to the same content, through whatever `ISceneLoader` is registered.
- **Scene handle** (`uint`) — which live instance of that asset. Opening the same scene id twice gives two handles and two independent worlds.
- **Scene object id** (`uint`) — which object inside a scene file. It is stamped by editor tooling and is only unique within that one scene asset.

None of these substitutes for another. The id tells you what to load, the handle tells you which loaded copy you mean, and the object id only makes sense once you already know which instance it lives in.

SceneObjectKey

An object authored into a scene is identified by the pair of scene handle and scene object id together, `SceneObjectKey`. A `SceneObjectId` alone is ambiguous the moment two live instances of the same scene asset are open, since both instances stamped their objects from the same file and both objects carry the same id. Pairing it with the handle is what lets a peer holding several scene instances at once bind a spawn to the right one.

`NetworkSystem.SceneHandle` reports which instance an object belongs to. Its unset value is `NetworkSystem.UnsetSceneHandle` (zero), which means the object belongs to no server-opened scene — the scene a peer booted into, rather than one `SceneManager` opened and tracks.

The server drives everything

`SceneManager` opens instances, places and removes clients, and closes instances. A client never decides any of this for itself: it loads a scene only when asked, releases one only when asked, and reports back only what it was asked to report. That is what makes the server's record of who holds what trustworthy enough to serve objects from — a client cannot inflate what the server believes it holds by loading or reporting on its own initiative.

`SceneManager.TryOpenScene` and `OpenSceneAsync` open instances. `TryRequestSceneLoad` and `RequestSceneUnload` place and remove clients. `EnsureCloseScene` and `CloseSceneAsync` close instances. See Opening and closing scene instances and Placing clients in scenes for the full API.

Every instance coexists, nothing is replaced

Open scene instances are tracked by handle in a table the server owns, and closing one only ever removes that one entry. Nothing about opening a scene id again reuses or merges state with an instance already open for that id. Handles are handed out from a counter that only climbs and is never recycled, so a handle you have seen always names the same instance for as long as anything remembers it, and a later instance can never be confused with an earlier, closed one.

This is what makes scene stacking possible: opening one asset several times gives you several coexisting instances, each with its own handle, and the handle is the only thing that tells them apart — they still share the same world coordinates. The shipped Unity scene loader (`UnitySceneLoader`) implements this by loading each instance additively (`LoadSceneMode.Additive`), so several copies of one Unity scene can

be open in the same process at once. That is the loader's choice of how to realize the model, not a rule the model itself imposes.

SceneScope: who an instance is for

`SceneScope` is decided once, when the instance is opened, and cannot change afterward:

- `SceneScope.Connections` — nobody is placed until `TryRequestSceneLoad` asks for it. The instancing case: an arena, a dungeon, a match.
- `SceneScope.Global` — every connected client is placed immediately, and every client that authenticates afterward is placed as it arrives. The single-world or lobby case.

Because scope is per instance rather than per asset, one scene asset can run as a shared `Global` lobby while a dozen other instances of a different (or the same) asset run as private `Connections` matches, all open at once.

The wire carries a number

A scene is never sent as a path or a name. Every message that names a scene carries the scene id or the scene handle — plain numbers — and the registered `ISceneLoader` is what turns an id into actual content on each peer.

Where to go next

- UnitySceneLoader component and UnitySceneManager component for the Unity setup.
- Opening and closing scene instances and Placing clients in scenes for the API.

Objects authored into a scene

What a scene object is

A scene object is a `NetworkSystemObject` placed directly in a scene asset rather than instantiated from a prefab at runtime. It already exists on every peer that has the scene loaded, so spawning it is a matter of agreeing which local instance a given network system binds to, not creating a new GameObject.

That agreement runs on identity. `NetworkSystemObject.SceneObjectId` names the object within its scene, and `IsSceneObject` reports `true` whenever that id is non-zero. On the wire side, `NetworkSystem.IsSceneObject` and `NetworkSystem.PlatformId` carry the same facts: a scene object's `PlatformId` is its scene identifier, and `IsSceneObject` tells the receiver to bind to an object it already has instead of instantiating one.

Stamping the id

`SceneObjectId` is not something you type in. `SceneNetworkObjectStamper`, an editor-only hook on `EditorSceneManager.sceneSaving`, assigns it automatically: every `NetworkSystemObject` in the scene being saved is scanned, ids already in use are kept, and any marker with an unset or duplicated id is stamped with the next free value. This is why the field is serialized but hidden in the inspector — it is scene-save state, not something to author by hand.

Duplicates are the normal case that makes this necessary. Copying an already-stamped object copies its serialized id along with it, so the copy and the original briefly share one `SceneObjectId`. On the next scene save, the stamper keeps the id on whichever object it saw first and reassigns the other to a fresh, unused value.

Why the id only has to be unique per scene

Identity on the wire is the pair of scene handle and scene object id, `SceneObjectKey(SceneHandle, SceneObjectId)`. Neither half identifies an object on its own: `SceneObjectId` is stamped per scene file and is only unique within it, so two different scenes — or two live instances of the same scene loaded side by side — can both contain an object stamped with id `1`. `SceneHandle` supplies the other half, naming which live scene instance the object belongs to. A receiver holding several scenes, or several stacked copies of one scene, binds each incoming spawn to the right instance by matching both halves together.

This also means a scene stacked twice never collides. Each load gets its own `SceneHandle`, so the two copies of an authored object are distinguished by handle even though they carry the identical `SceneObjectId`.

How this differs from a spawned prefab on the wire

A dynamically spawned prefab's `NetworkSystem.PlatformId` names a prefab to instantiate, and `IsSceneObject` is `false`. For a scene object, `IsSceneObject` is `true` and `PlatformId` carries the scene identifier instead. The receiver reads that flag to choose its path: a spawned system is constructed and instantiated fresh, while a scene object's system is bound onto the GameObject that is already sitting in the loaded scene.

When a copied object binds to the wrong thing

If you duplicate a stamped `NetworkSystemObject` and the scene is not saved afterward, both copies still carry the same `SceneObjectId` in memory. A receiver resolving a spawn against that id can bind to the wrong instance, since nothing on disk yet distinguishes the two.

The fix is to save the scene. Saving runs `SceneNetworkObjectStamper`, which reassigns the duplicate's id to a free value and writes it into the scene asset. Because the field is hidden in the inspector, there is no manual value to correct — re-saving is the only way to resolve it.

Scenes and bundles: things that look broken

A client connects and sees nothing

`SceneManager.JoinPlacement` decides what a newly authenticated client is put into, and its default, `JoinScenePlacement.EveryOpenScene`, is what most projects want without calling `TryRequestSceneLoad` anywhere. If `JoinPlacement` was set to `JoinScenePlacement.None`, no joining client is placed automatically, and under `JoinScenePlacement.GlobalScenes` a scene opened `SceneScope.Connections` rather than `SceneScope.Global` is skipped at join. Either way, a client that never receives a `TryRequestSceneLoad` call for that scene sees nothing.

The other cause is timing rather than configuration: a `SceneScope.Connections` scene opened *after* a client already joined is not pushed to that client. Join placement happens once, when the client authenticates, and only covers the scenes open at that moment. A `SceneScope.Global` scene is the exception: opening it places every client already connected. A client connected before a `Connections` scene existed has to be asked for it explicitly.

Check `JoinPlacement`, the scope the scene was opened with, and, for a `Connections` scene, whether it existed before the client connected.

Objects appear in the wrong copy of a stacked scene

A stacked scene is one asset opened more than once, so every copy shares the same name and path — `GetSceneByName` and `GetSceneByPath` cannot tell them apart, only the scene handle can. `UnitySceneManager.TryGetScene` resolves a handle to the live Unity `Scene`, and that resolved scene is what has to be made active, or the target for `MoveGameObjectToScene`, before anything spawns into it.

The handle itself is never the cause: a handle is issued once per open scene instance and never reused across opens. If objects are landing in the wrong copy, look at the spawner instead: it instantiated while the wrong Unity scene was active, or it moved the wrong scene's copy.

A second, unrelated cause produces the same symptom for a parented object: the Unity integration only moves a root `GameObject` between scenes, because that is the only kind of move Unity itself supports. An object with a parent is refused rather than moved and stays in its parent's scene, and an error naming the parent is logged. Un-parent the object, or move its parent instead.

A load never completes

Two different timeouts produce this, and they are not the same failure:

- `SceneManager.LoadRequestTimeoutSeconds` (default 180 seconds, zero or less waits indefinitely) is how long the server waits for a client to answer a scene load request. If the client never answers, the server treats it as a failure, logs a warning naming the scene handle and the timeout, and raises `ClientSceneLoadFailed`.
- On the client, nothing completes if the loader never calls back. A custom `ISceneLoader.LoadSceneAsync` implementation has to eventually call `SceneManager.NotifySceneLoaded` (or fail through `NotifySceneLoadFailed`) — a loader that awaits something that never resolves leaves the load hanging until the server's own timeout catches it.

If the load is stuck client-side with no timeout firing at all, check that the loader in use actually reports back through `NotifySceneLoaded`.

A load comes back as Refused rather than Failed

`SceneLoadOutcome.Refused` means the client's `ISceneLoader.MaximumConcurrentScenes` was already at its limit — the loader was never consulted, because the request was rejected before it got there. `SceneLoadOutcome.Failed` means the loader was asked and the load itself did not work: the content would not fetch, the scene would not resolve, or the load threw.

The two are deliberately distinct outcomes because the remedy differs. A `Failed` load can just be asked for again with `SceneManager.TryClearSceneLoadFailure`. A `Refused` load cannot — asking again hits the same limit — the client has to be moved with `SceneReplaceMode.AllScenes` instead, so it releases what it holds before taking on the new one.

A client is kicked during scene or bundle traffic

Four violations cover unexpected scene and bundle messages, and all default to a kick:

- **`UnsolicitedSceneReportViolation`** — a client reported a scene load or release the server never asked for. Carries `SceneHandle` and `IsLoadReported` (true for a claimed load, false for a claimed release).
- **`UnexpectedSceneRequestViolation`** — a client sent a scene load request, a message that only ever travels server to client. Carries `SceneHandle` and `IsLoadRequested`.
- **`UnsolicitedBundleReportViolation`** — a client reported a bundle load or unload the server never asked for, or does not record it holding. Carries `BundleId` and `IsLoadReported`.
- **`UnexpectedBundleRequestViolation`** — a client sent the server a bundle load request, a message that only the server sends. Carries `BundleId`.

All four are a client trying to drive placement or content that only the server is allowed to drive. A well-behaved client cannot produce any of them, so seeing one means a modified or malicious client, not a legitimate race — including the unset scene handle, which is never requested and never recorded held, and which reports as unsolicited too.

Prefabs silently never spawn for one client

`BundleInterestCondition` withholds a `NetworkSystem`'s spawn from a `Connection` until that client holds the content bundle its prefab shipped in (`NetworkSystem.RequiredBundleId`), so the spawn is never delivered to a peer that cannot instantiate it — no error, the object just does not exist there yet. It abstains for a scene object (already bound rather than instantiated), for the base build (`Connection.BaseBuildBundleId`), and for a host's own local or loopback connection.

Whether the client is ever told to fetch the missing bundle depends on `BundleManager.AutomaticRequestOnBlockedSpawnEnabled`. When it is on, the withheld spawn triggers `BundleManager.RequestBundleLoad` for that connection and bundle automatically. When it is off, nothing asks the client for the bundle — the spawn stays withheld until the client is given the bundle some other way. If an object is missing for exactly one client and no error appears, check whether that client holds the required bundle (`Connection.IsBundleLoaded`) and whether automatic requesting is enabled.

A load brings a world back missing objects

Two different lookups can fail on a world load, and each is logged and skipped rather than aborting the whole load:

- A saved scene instance names a `sceneId` the manifest no longer resolves. `WorldPersistenceManager` reopens each saved scene; if that fails, `IWorldStore.OnSceneUnavailable` decides what happens: either the whole load is abandoned (logged as "the world is empty rather than half built"), or the objects in that scene are dropped and the

rest of the load continues.

- A saved object's identity no longer resolves. For a scene object, `WorldSystemIdentity.PlatformId` is looked up in the restored scene; if nothing matches, that object is dropped and logged as not found. For a constructed object, `WorldSystemIdentity.SystemTypeName` is looked up with `NetworkTypeRegistry.TryCreateSystemByName`; a type no longer registered in this build is dropped the same way.

Missing objects after a load are one of these two lookups failing, not a corrupted save — check the log for which scene id or system type it named.

In Unity: a duplicated prefab or scene object carrying a copied stamped id

Copying an object that already carries a stamped id copies the id with it, and two objects sharing one id is a correlation collision.

For scene objects, `SceneNetworkObjectStamper` runs on every scene save: it keeps the first occurrence of each `SceneObjectId` and reassigns any later duplicate to the next free value, so a duplicated scene object gets a fresh id automatically the next time the scene is saved. Until that save happens, the duplicate still carries the copied id.

For a duplicated prefab, the network id is stamped onto the prefab asset itself, so copying the prefab duplicates its id along with it and a spawn built from the copy resolves to the wrong object. Nothing re-stamps it on save, but **Nucleus > Rebuild Network Prefab Collection** does: the original keeps its id, because the collection already lists it, and the copy is given the next free id. A prefab variant that inherited its base's id is renumbered the same way, and its base keeps the id. Then rebuild both the server and the client builds so every peer carries the same ids. A player build made before the rebuild fails with the clash listed, and the registry logs an error naming both prefabs if two that share an id are ever registered together, so the problem shows up in the Console rather than as a wrong spawn.

UnitySceneLoader component

Driving the core API directly? See Registering a scene loader.

`UnitySceneLoader` is the shipped `ISceneLoader` implementation for Unity: it maps the scene identifiers a server sends to scene assets, loads them additively, and reports back so withheld spawns can land. Add it to a GameObject alongside (or reachable from) the integration's CoreManager and it registers itself automatically.

Inspector fields

Field	Inspector label	Description
`_sceneManifest`	Manifest	Maps scene identifiers to scene assets and the content bundle each ships in. Optional: leave it empty and the loader resolves the project's manifest from Resources on first use.
`_localPhysicsEnabled`	Local Physics	Loads each networked scene into its own local physics world and attaches a driver so Nucleus steps it every tick.
`_automaticRegistrationEnabled` (inherited from `NetworkSceneLoader`)	Auto Register	Registers with the integration's CoreManager automatically on `Awake`. Turn it off to call `Initialize` yourself.

`Manifest` is a public property that resolves through `ResolveManifest()`, so reading it triggers the same lazy Resources lookup.

Local Physics

With Local Physics on, each scene loads with `LocalPhysicsMode.Physics3D` instead of the default physics world, and a physics simulation driver is attached to step it. This is what keeps two stacked copies of the same arena from colliding with each other: Unity never auto-simulates a local physics scene, so without this a stacked scene's bodies would just sit still. Bodies inside a locally-physics scene need manual interpolation.

NetworkSceneLoader

`NetworkSceneLoader` is the abstract `MonoBehaviour` that implements `ISceneLoader`. `UnitySceneLoader` derives from it. It owns:

- `LoadSceneAsync(uint sceneHandle, ushort sceneId)` and `UnloadSceneAsync(uint sceneHandle)` — abstract, asynchronous, and may take as long as needed; the network loop never blocks on them.
- `MaximumConcurrentScenes` — a `virtual uint` defaulting to `SceneManager.UnlimitedConcurrentScenes`, because Unity loads scenes additively and holds as many as memory allows. Override it only if a replacement loading pipeline genuinely can't hold every scene the server asks for concurrently.

- `_automaticRegistrationEnabled` and `Initialize(CoreManager)` — registration is deliberately on `Awake`, not `Start`, so game code that loads scenes from its own `Start` always finds a loader already registered.

Extension points

Subclass `UnitySceneLoader` and override its protected hooks instead of reimplementing `ISceneLoader` from scratch:

- `TryGetScene(ushort sceneId, out NetworkSceneManifest.SceneEntry sceneEntry)` — resolves a scene identifier through the manifest. Override to source scene entries from somewhere other than a `NetworkSceneManifest`.
- `EnsureSceneContentAsync(ushort requiredBundleId)` — makes sure the content bundle a scene ships in is available before the scene loads.
- `LoadSceneCoreAsync(uint sceneHandle, ushort sceneId)` — the default additive load via `UnitySceneManager.LoadSceneAsync`, or `LoadAddressableSceneCoreAsync` when the entry is an Addressable.
- `LoadAddressableSceneCoreAsync(uint sceneHandle, ushort sceneId, NetworkSceneManifest.SceneEntry sceneEntry, Task<bool> bindingTask)` — the Addressables load path, via `Addressables.LoadSceneAsync`. Only compiled under `NUCLEUS_ADDRESSABLES`.
- `UnloadSceneCoreAsync(uint sceneHandle)` — the unload counterpart to `LoadSceneCoreAsync`.
- `ResolveManifest()` — lazily resolves and caches `_sceneManifest`, falling back to `NetworkSceneManifest.LoadProjectManifest()` when none is assigned.

Each is `protected virtual` (`TryGetScene`, `EnsureSceneContentAsync`, `LoadSceneCoreAsync`, `UnloadSceneCoreAsync`, `LoadAddressableSceneCoreAsync`) or `protected` (`ResolveManifest`), so a subclass can override just the piece it needs while keeping the rest of `UnitySceneLoader`'s behavior.

Addressables support

Addressable scene loading (`LoadAddressableSceneCoreAsync`) compiles only under the `NUCLEUS_ADDRESSABLES` define. The integration's assembly definition sets that define through a `versionDefines` entry keyed on the `com.unity.addressables` package at version `1.19` or newer — install Addressables and the define, and the Addressables load path, follow automatically.

UnitySceneManager component

Driving the core API directly? See Placing clients in scenes.

`UnitySceneManager` is the inspector-side driver for the core `SceneManager`. It pushes its serialized fields into the core manager once on `ManagersInstantiated`, and gives the peer a scene loader if the project has not placed one itself.

Fields

Inspector label	Field	Default	Pushed into
Join Placement	`_joinPlacement`	`JoinScenePlacement.EveryScene`	`SceneManager.JoinPlacement`
Auto Request On Blocked Spawn	`_automaticRequestOnBlockedSpawnEnabled`	`false`	`SceneManager.AutomaticRequestOn`
Scene Loader Enabled	`_sceneLoaderEnabled`	`true`	nothing on `SceneManager`; gates whether `EnsureSceneLoader()` adds a `UnitySceneLoader`
Auto Simulate Stacked Scenes	`_automaticStackedSceneSimulationEnabled`	`false`	nothing on `SceneManager`; sets `UnitySceneLoader.LocalPhysicsEnab`
Scene Load Timeout (s)	`_sceneLoadTimeoutSeconds`	`180`	`SceneManager.LoadRequestTimeout`

Only three of the five reach the core manager. Join Placement, Auto Request On Blocked Spawn, and Scene Load Timeout are assigned straight onto `SceneManager` in `ManagersInstantiated`. The other two only affect the scene loader this component may add.

Scene Loader Enabled decides whether `EnsureSceneLoader()` runs at all. When true, and no `NetworkSceneLoader` exists yet anywhere in the loaded scenes (active or inactive), a `UnitySceneLoader` is added to this component's GameObject. A loader the project placed itself is found first and always wins; this only fills a gap.

Auto Simulate Stacked Scenes only writes `UnitySceneLoader.LocalPhysicsEnabled` on the loader this component adds through `EnsureSceneLoader()`. A loader the project placed itself keeps whatever `LocalPhysicsEnabled` value it was given directly; this field never touches it.

Scene Load Timeout controls how long the server waits for a client to answer a scene load request before treating the load as failed. Zero or less waits indefinitely.

Reaching the core SceneManager

`NucleusSceneManager` is the property every script uses to reach the core manager from this component:

```
public Nucleus.Managers.Scenes.SceneManager NucleusSceneManager { get; private set; }
```

It is set once, in `ManagersInstantiated`, from `unityCoreManager.CoreManager.SceneManager`.

Translating between a scene handle and a Unity Scene

Every scene loads additively, so a world that opens the same asset more than once holds several Unity

scenes with the same name and the same path, and Unity's `GetSceneByName`/`GetSceneByPath` answer with an arbitrary one of them. The scene handle - the instance id the server assigned when it opened the scene - is the only thing that tells those copies apart, and `TryGetScene`/`TryGetSceneHandle` are the only supported translation between it and a live `Scene`:

```
public bool TryGetScene(uint sceneHandle, out Scene scene)
public bool TryGetSceneHandle(Scene scene, out uint sceneHandle)
```

`TryGetScene` resolves the live Unity scene that a server-opened instance became on this peer. It returns false while the scene is still loading, and false on a peer the instance was never opened for. A handle can be known - the server already told this peer about the instance - before its `Scene` exists locally, because the load is still in progress; that is the gap this method reports.

`TryGetSceneHandle` is the inverse: given a live `Scene`, it resolves which instance it is. It has three possible outcomes, not two: false means the scene is still loading and its objects are running `Awake` right now; true with `NetworkSystem.UnsetSceneHandle` means this peer loaded the scene itself and no instance will ever claim it; true with any other value names the instance. Collapsing the last two into one "false" would lose the difference between "not yet" and "never".

Saving and loading a world from Unity

Driving the core API directly? See Saving and loading a world.

There is no Unity persistence component. `WorldPersistenceManager` lives on `CoreManager`, and you drive it from a `MonoBehaviour` you write yourself: resolve `CoreManager` from `UnityCoreManager`, then call into `CoreManager.WorldPersistenceManager`.

Getting to the manager

```
UnityCoreManager unityCoreManager = GetComponent<UnityCoreManager>();
CoreManager coreManager = unityCoreManager.CoreManager;
WorldPersistenceManager worldPersistenceManager = coreManager.WorldPersistenceManager;
```

Both `SaveAsync` and `LoadAsync` are refused unless `TransportManager.IsServerStarted` is true. A client holds a copy of the server's world, not one of its own, so persistence only runs on the server.

Registering a store

`WorldPersistenceManager` needs a store before it will save or load anything. `JsonWorldStore` is the store Nucleus ships: it takes the `CoreManager` and a file path, and writes/reads a JSON file there.

```
string filePath = Path.Combine(Application.persistentDataPath, "world.json");
JsonWorldStore jsonWorldStore = new(coreManager, filePath);

worldPersistenceManager.SetWorldStore(jsonWorldStore);
```

`Application.persistentDataPath` is the usual choice in Unity because it is writable on every platform Unity targets. `JsonWorldStore` writes to a temporary file beside the target and moves it over the real path only once the write finishes, so a crash mid-save leaves the previous world intact rather than a truncated one.

Saving

```
bool saved = await worldPersistenceManager.SaveAsync();
```

`SaveAsync` runs on the loop and doesn't need the pump described below - it completes without switching threads.

Loading, and why the MonoBehaviour has to pump the loop

`LoadAsync` is different. Despawning and rebuilding the routing table can only happen on the network loop, but the awaited store call (`JsonWorldStore.LoadAsync`, file I/O) can resume on any thread. To get back onto the loop, `LoadAsync` queues the rest of its work onto the network loop, which only runs it when the loop is being driven and drains its work.

That means the loop has to keep being driven while the load is outstanding. In Unity, the loop is driven from `Update`/`LateUpdate` on the main thread, so blocking the main thread on this task deadlocks: the load is waiting for the loop to drain, and the loop is waiting for your thread to come back and drive it. An ordinary `await` from an `async` method on the main thread does not block. It hands the thread back to Unity, which keeps ticking, and your method resumes once the load finishes.

Await it from an `async` method:

```
public class WorldLoader : MonoBehaviour
{
    public void BeginLoad() => _ = LoadWorldAsync();

    private async Task LoadWorldAsync()
    {
        bool loaded = await GetComponent<UnityCoreManager>().CoreManager
            .WorldPersistenceManager.LoadAsync();

        // handle the result
    }
}
```

Or start the task and poll it from `Update`:

```
public class WorldLoader : MonoBehaviour
{
    private Task<bool> _loadTask;

    public void BeginLoad()
    {
        _loadTask = GetComponent<UnityCoreManager>().CoreManager
            .WorldPersistenceManager.LoadAsync();
    }

    private void Update()
    {
        if (_loadTask is { IsCompleted: true })
        {
            bool loaded = _loadTask.Result;
            _loadTask = null;
            // handle the result
        }
    }
}
```

Either way, control goes back to Unity while the load runs, so its own `Update` loop keeps pumping the network loop on subsequent frames, which is what lets the load get back onto the loop and finish. Anything that blocks synchronously on the task (`.Wait()`, `.Result` before it completes) from that same main thread reproduces the deadlock.

What a load does to the hierarchy

A load replaces the world rather than merging into it:

Every standing object is despawned.

Every open scene is closed.

Saved scenes are reopened through the registered `ISceneLoader`, under the handle each was saved by. A scene the loader cannot open is the most common load failure - the object can be dropped or the whole load can fail, depending on the store's `WorldLoadFailureAction`. A failed load does not restore the old world, since steps 1 and 2 have already run: it leaves an empty one.

Saved objects are rebuilt. A scene object is bound to the placed object already in the reopened scene, matched by `PlatformId`. A non-scene object is constructed from its saved system type and, once built, is looked up in the prefab registry by its stamped `PlatformId` (exposed as `PrefabId` on `NetworkSystemObject`) and prefab bundle id. A prefab missing from the collection that shipped that

bundle cannot come back.

On every connected client this looks like a world of despawns followed by a world of spawns - there's no incremental diff between the old and new world.

Requirements

- Pro only.
- Authority only (`TransportManager.IsServerStarted`).
- A load over a live session is disruptive to every connected client for the reasons above; consider whether players should be on a loading scene while it runs.

Replicated State and Change Callbacks

Pages in this category:

- NetworkTransform
- How replicated state works
- Replicating a value from a Unity script
- Reacting to a replicated change in Unity
- NetworkAnimator

NetworkAnimator

NetworkAnimator replicates parameters, not animation state

`NetworkAnimator` replicates a GameObject's `Animator` across peers. Parameters are the primary channel: the peer driving the object reads its float, int, bool and trigger parameters, its layer weights and its playback speed, and every other peer applies those values and lets its own `Animator` run its own state machine from them.

The layer state (each layer's current state hash and normalized time) rides on top as a correction, not as the primary replication path. It costs nothing between transitions, because a hash only changes when a layer transitions. It serves two purposes: a peer whose state machine branched differently from the controller is pulled back onto the server's state, and a peer that joins mid-session opens each layer on the reported state instead of at the controller's entry state.

The phase a joiner opens on is the live one. A spawn, a late join, an interest serve and a resync each carry every layer's phase as the authority's animator holds it at that moment, so a joiner opens a looping idle where the authority is now, not where the authority entered the state.

When parameters alone are not close enough, for example an animated part that lag-compensated hitboxes are judged against, turn on Precise Playback and followers play the authority's exact pose instead.

Add `NetworkAnimator` to the same GameObject as the `Animator` it should replicate; it requires a `NetworkSystemObject`.

Animator field

```
[SerializeField]
private Animator _animator;
```

The `Animator` to replicate. Left empty in the inspector, the component falls back to the `Animator` on its own GameObject at `Awake`.

The values actually captured and applied live on `UnityAnimatorComponent`, the replicated component `NetworkAnimator` binds to through its `NetworkSystem`: `FloatParameters`, `IntParameters`, `BoolParameters` (packed one bit per parameter, up to 64), `TriggerPulses`, `LayerWeights`, `Speed`, `LayerStateHashes` and `LayerNormalizedTimes`, plus `LayerPlaybackPositions`, `LayerNextStateHashes`, `LayerNextPlaybackPositions` and `LayerTransitionProgress`, which stay empty and cost nothing unless Precise Playback is on. `NetworkAnimator` is the MonoBehaviour surface over that component; it doesn't hold the replicated data itself.

An animator with more than 64 bool parameters replicates the first 64 and drops the rest. `NetworkAnimator` logs an error naming the count when it binds, so split the extra bools into int parameters or a second animator if they need to reach other peers.

State Correction Interval

```
public enum StateCorrectionInterval : byte
{
    Disabled,
    Always,
    Short,
    Long
}
```

Controls how often the replicated normalized time of each layer is refreshed while no layer is transitioning, which bounds how far a follower already in the session can drift from the authority's phase during a long loop. The state hash costs nothing between transitions since it doesn't change, but the phase advances every tick a clip plays, so refreshing it on a cadence means a character idling in a loop pays for that idle forever.

- **Disabled** (default): the phase only updates on a transition. A joiner still opens on the live phase, because the statement it joins on carries it; after that, a follower in a long loop runs on its own animator's clock until the next transition.
- **Always** — refreshed every tick. The most expensive tier; a follower's phase tracks the server's exactly rather than merely opening near it.
- **Short** — refreshed at most every 250ms.
- **Long** — refreshed at most once a second, for a loop a joiner only needs to open roughly in the right place.

`Short` and `Long` mirror the same-named `SendInterval` tiers and resolve through the same `SendIntervalTicks.Resolve`, so a span read on this field means the same thing it means on a send interval. Correction of a peer whose state machine took a different branch doesn't depend on this setting at all: the state hash carries that on the tick the divergence happens either way; this setting only bounds how far a follower's phase can drift between transitions.

Unlike a paced send interval, none of this is a Pro feature. It's a local decision about how often to re-read an animator, so every tier here paces in a free build.

Precise Playback Enabled

```
[SerializeField]
private bool _precisePlaybackEnabled;
```

Off by default. With it on, the authority captures every layer's playback position each tick, and during a transition also the next state, its position and how far the blend has progressed. A follower stops running its own state machine: it holds its animator still and places each layer at the authority's position, interpolated between the two newest ticks it applied, cross-fading at the replicated progress. The pose is evaluated at once, so a raycast in the same frame sees it.

Use it for an animated part whose pose has to match the authority's, such as a limb carrying hitboxes that a lag-compensated shot is judged against. Measured at 30 Hz, a precisely played part sits within about a millimetre of the server's recorded pose, where parameters alone sit a few centimetres off.

It costs a value per layer every tick, and more during a transition, where parameters alone cost nothing between changes. A state that does not loop is held at its end, so a finished or static clip stops being sent. Animation events do not fire on a follower in this mode, because its clips are placed rather than played through; fire game logic from replicated state or a remote call instead.

Root Motion Management Enabled

```
[SerializeField]
private bool _rootMotionManagementEnabled = true;
```

When enabled (the default), `Animator.applyRootMotion` is cleared on every peer that doesn't drive the object, so root motion can't fight the replicated pose, and restored on the peer that does drive it. Disable it when other code owns the flag, or when the object doesn't replicate a transform.

The public members

All writing methods are silently no-ops on a peer that doesn't drive the object — the same script runs on every peer, and a call that has effect on the controller is inert everywhere else.

```
public float Speed { get; set; }

public void SetFloat(string parameterName, float value);
public void SetFloat(int parameterHash, float value);

public void SetInteger(string parameterName, int value);
public void SetInteger(int parameterHash, int value);

public void SetBool(string parameterName, bool isParameterTrue);
public void SetBool(int parameterHash, bool isParameterTrue);

public void SetLayerWeight(int layerIndex, float weight);

public void SetTrigger(string parameterName);
public void SetTrigger(int parameterHash);

public void ResetTrigger(string parameterName);
public void ResetTrigger(int parameterHash);

public void Play(int stateNameHash, int layerIndex = 0, float normalizedTime = 0f);
public void CrossFade(int stateNameHash, float normalizedTransitionDuration, int layerInde
x = 0);
```

`Speed` maps to `Animator.speed`, the whole animator's playback rate, distinct from any parameter a controller happens to name "Speed". `Play` and `CrossFade` drive the local animator on the controller; the resulting state reaches every other peer through the replicated layer state, not through the fade itself — a follower is corrected onto the destination state at the server's reported phase rather than told to blend into it, so the fade `CrossFade` would have played is the server's alone.

Triggers must go through SetTrigger

Never call `Animator.SetTrigger` directly on the replicated animator. Unity consumes a trigger inside the state machine, so a trigger set directly can be raised and consumed between two ticks and be gone before anything here could observe it to replicate the fire.

`NetworkAnimator.SetTrigger` fires the local animator and raises the replicated trigger count in the same call, which is why it must be the only way a trigger is fired on this object. `ResetTrigger` clears it locally and, if the fire hasn't been serialized yet this tick, takes back the count too; a reset arriving on a later tick is local only, since the fire has already gone out to observers.

Late joiners don't replay history

A trigger fires as a rising count rather than a one-shot event, so a receiver applies the difference between what it last applied and what it just received. The first counts a joining peer receives establish its baseline instead of being treated as fires to play — a peer that joins after ten triggers have already fired sees its animator start clean, not fire all ten at once.

Reacting to a parameter landing

To run code when a replicated parameter or layer state changes on an observing peer, hook into the change-callback path on the replicated component (`OnMembersChanged`) rather than polling the `Animator`

every frame.

NetworkTransform

Driving the core API directly? See TransformComponent.

`NetworkTransform` replicates a GameObject's position, rotation, and scale. Add it to a networked object, either on the GameObject that carries the `NetworkSystemObject` or on a child of it, and the controller's transform replicates to every other peer. A `NetworkTransform` belongs to the nearest `NetworkSystemObject` at or above it; with none there, it is never given a system (**Nucleus > Validate Network Prefabs** reports that).

The controller's pose is sampled just before serialization, on the `EarlySerialize` loop step, so it does not matter what order your own scripts run in relative to it. The capture always happens right before the tick's state goes out, not whenever some earlier `Update` left the transform. A non-controller never drives the transform: it is a pose follower, animated between the last two received values through the interpolation buffer. Velocity is not replicated.

Inspector Fields

Field	Default	Effect
Transforms	empty	The transforms this component replicates, all on one system. Empty replicates the component's own GameObject. See [Replicating several parts](#replicating-several-parts).
Transform Space	`Local`	Which space the pose replicates in. See below.
Interpolate Scale Enabled	`true`	Interpolates the replicated scale on non-controllers alongside position and rotation. Disabled, local scale is left alone.
Kinematic Management Enabled	`true`	Holds an attached `Rigidbody` kinematic on every non-controlling peer, including the server while a client controls the system, and returns it to simulation on the controller.
Send Interval	`Normal`	Paces ordinary delta replication while Transmission Mode is `Interval`. Has no effect on a projecting (`Divine`) member — a projected path already streams and smooths on its own.

Transmission Mode	`TransmissionModeDefaults.Motion`	The replication strategy applied to position, rotation, and scale on bind. Resolves to `Divine` projection where a projector is registered, `Interval` otherwise.
Extrapolation Enabled	`true`	Whether other peers keep the object moving through the sender's silence, under either Transmission Mode. On, a lost update is covered by motion but may briefly overshoot; turn it off if that overshoot is visible, and the object holds still until told instead. Under `Interval`, on costs a tiny overhead on each update plus one extra update each time the object stops. See [How Often a Member Is Sent](../../core-api/state/send-pacing.md).

Transmission Mode, Extrapolation Enabled, and Send Interval are pushed onto the underlying `UnityTransformComponentBase` on bind (`SetTransmissionMode`, `SetExtrapolationEnabled`, `SetSendInterval`), so a pooled instance re-applies the same authored values on every reuse. Interpolate Scale Enabled and Kinematic Management Enabled stay on `NetworkTransform` itself and are read directly each step, not pushed to the component. Transforms and Transform Space are spent earlier still, once in `Awake`, before any bind happens, and both are locked in the inspector during play mode.

Replicating several parts

An object made of several moving pieces (a turret and its barrel, a body and an animated arm) does not need a `NetworkTransform` on every piece. List the pieces in **Transforms** on one `NetworkTransform` and they all replicate on a single system: one controller, one spawn, and one system header per update for all of them, with every piece sharing the component's settings. Each listed transform gets its own transform component on that system, so moving one piece sends only that piece.

The order of the list matters. Entry *i* is bound to component Id *i* on every peer, and both peers instantiate the same prefab, so they agree as long as the list is part of the prefab rather than edited per instance. Every entry must sit at or below the object's `NetworkSystemObject`, null entries are skipped, and a list longer than a system can hold (`NetworkSystem.MaximumComponentCount`, 31) is refused with an error.

You can also put a `NetworkTransform` directly on a child, for example an animated part that should replicate with its own settings. It joins the nearest `NetworkSystemObject` above it and declares its own system under that one marker, so several `NetworkTransform` components can share a marker this way. Do not add a second `NetworkSystemObject` to the child: only a prefab's root marker is spawned.

When the parts share settings, prefer one `NetworkTransform` with a list over one per part: it registers with the network loop once instead of once per part, and saves a system header on every update.

Transform Space

`TransformSpace` is read once, when the component declares its requirement, and fixes which transform

component type the object's `NetworkSystem` carries for the rest of its life:

- **`TransformSpace.Local`** (default) — the system carries a `UnityLocalTransformComponent`, which replicates `Transform.localPosition` and `Transform.localRotation`. A carried object rides for free: as long as it stays where it was inside its parent, its own replication stays silent no matter how far the parent travels. This requires the parent to be replicated too, since an offset only means the same thing on both peers if they agree on the frame it's measured against. A reparent carries the local values straight through the move — the object lands at the same offset, now measured inside the new parent, so the replicated numbers never jump (`IsWorldPoseKeptOnReparent` is `false`).
- **`TransformSpace.World`** — the system carries a `UnityWorldTransformComponent`, which replicates `Transform.position` and `Transform.rotation`. The pose means the same thing on every peer regardless of what the object is parented to, so use it when an object can hang from something that isn't itself replicated (a client-local rig, a per-peer presentation root). A reparent moves nothing (`IsWorldPoseKeptOnReparent` is `true`); the cost is that a carried object pays full pose replication for as long as it rides, since its world pose changes every tick even while it sits still relative to its carrier.

Both peers instantiate the same prefab, so they always agree on which space an object uses — the choice is part of the system's identity on the wire, not something that could drift between peers.

Kinematic Management

With Kinematic Management Enabled on, an attached `Rigidbody` is held kinematic on every peer that is not the controller — including the server, while a client controls the system — so local physics can't fight the interpolated pose writes. The controller's rigidbody is returned to simulation. This is applied only when the controlling role actually changes, not every frame. Disable it when other code owns the rigidbody's kinematic state.

Projected Physics Bodies

`NetworkTransform` interpolates a received pose; it does not run local physics on non-controllers. For a rigidbody that should simulate locally on every peer and converge toward the server's state instead of just following it, use `ProjectedRigidbody`.

See Also

The replicated component this behaviour owns and binds to is `UnityTransformComponentBase` (`UnityLocalTransformComponent` or `UnityWorldTransformComponent` depending on Transform Space) — see TransformComponent for its API.

Reacting to a replicated change in Unity

Driving the core API directly? See Reacting to changes: OnMembersChanged.

Override OnMembersChanged, forward it to your MonoBehaviour

`NetworkComponent` declares a virtual for this:

```
public virtual void OnMembersChanged(ulong memberFlags, MemberChangeDirection memberChange
Direction) { }
```

It's called once per tick per direction when any of the component's members changed, so a Unity script never has to poll a value to find out it moved.

A `NetworkComponent` is a pooled engine object, not a scene object, so a MonoBehaviour that wants to react cannot inherit from it. Override `OnMembersChanged` on your component partial and raise an event instead, and have the MonoBehaviour that owns the visuals subscribe:

```
public partial class CoinComponent : NetworkComponent
{
    public event MembersChangedHandler MembersChanged;
    public delegate void MembersChangedHandler(ulong memberFlags, MemberChangeDirection me
mberChangeDirection);

    public override void OnMembersChanged(ulong memberFlags, MemberChangeDirection memberC
hangeDirection)
        => MembersChanged?.Invoke(memberFlags, memberChangeDirection);
}
```

The MonoBehaviour subscribes when it rents the system and unsubscribes when it's destroyed:

```
_coinComponent.MembersChanged += OnCoinMembersChanged;
// ...
_coinComponent.MembersChanged -= OnCoinMembersChanged;
```

An event rather than a direct override on the MonoBehaviour, because the component can be rented and returned independently of the scene object watching it.

Find which member moved

The source generator emits a `[Flags] enum <ComponentName>Flags : ulong` for every `NetworkComponent`, one bit per member in declaration order. `CoinComponent` declares `Collector` then `PickupSequence`, so its generated enum carries `CoinComponentFlags.Collector` and `CoinComponentFlags.PickupSequence`. Cast the raw `memberFlags` to it and test the bits you care about:

```
private void OnCoinMembersChanged(ulong memberFlags, MemberChangeDirection memberChangeDir
ection)
{
    CoinComponentFlags changed = (CoinComponentFlags)memberFlags;

    if ((changed & CoinComponentFlags.Collector) != 0)
    {
        // React to the collector reference changing.
    }
}
```

Delta flags name the members the wire carried, not a guarantee the value differs — a predicted member's

confirmed outcome still sets its bit.

Almost always test for MemberChangeDirection.Read

`MemberChangeDirection` has two values:

- `Write` — this peer changed the members itself; the raise arrives before the tick's serialization runs.
- `Read` — the members were applied from the wire; the raise arrives after every packet of the inbound pass has landed, before the Reconcile step.

A visuals script almost always wants `Read` only, because it's reacting to something that arrived, not something it just wrote. Anything that calls `NetworkMemberBase.TryGetWritingClient` needs `Read` specifically — that claim only resolves under a read.

This matters most on a host. A host raises both directions in the same frame for its own change, write first, without waiting for its own packet to come back — so a handler that doesn't branch on direction reacts twice to the same pickup on a host and once everywhere else. Guard every reaction with the direction check:

```
if (memberChangeDirection is not MemberChangeDirection.Read)
    return;
```

A value that's a plain function of the replicated state — recomputed the same way on every peer regardless of who wrote it — doesn't need the branch. Only genuinely one-sided reactions (a cooldown, a "you scored" label) do.

Full applies report every member

A spawn, a resync, a reconcile, or a whole-component recovery hands you `memberFlags` as `ulong.MaxValue` under `MemberChangeDirection.Read`: treat every member as changed. This is what a late-joining client's first raise looks like — it never saw a delta, so the full apply is the only place it learns the component's starting values. Don't assume a specific bit is set before checking `memberFlags == ulong.MaxValue` for this case; a member-granular recovery instead reports only the members it actually repaired.

Do the work on the right loop step

`OnMembersChanged` already fires event-driven at the right time, so nothing here should live in `Update`. If a script derived from `NucleusBehaviourBase` needs to do more than react to the event — read state the apply just landed, or run something every tick tied to that same point in the frame — override the per-step virtual instead of polling in `Update`:

```
protected override void OnLateStateUpdate(StepDelta stepDelta)
{
    // Runs once replicated state has been applied for this tick.
}
```

`OnEarlyStateUpdate` runs just before replicated state is applied, `OnLateStateUpdate` once it has been, and `OnReconcile` after state has been set back for a reconcile. `Update` runs on Unity's own loop with no defined relationship to any of those points in a given frame; the per-step virtuals do.

Replicating a value from a Unity script

Driving the core API directly? See Your first networked component.

Declare the component

A networked component is a plain class, not a `MonoBehaviour`. It never goes on a GameObject itself.

```
public partial class HillTimerComponent : NetworkComponent
{
    public readonly NetworkMember<float> SecondsRemaining = new();
}
```

`partial` matters: the source generator fills in the rest of this class from the `NetworkMember<T0>` fields it finds, so `Read`, `Write`, `WriteDelta` and the rest are written for you. Add fields, don't hand-write serialization.

Attach it through NucleusBehaviour

Since the component itself can't sit on a GameObject, a `MonoBehaviour` adopts it by inheriting `NucleusBehaviour<TComponent0>`:

```
public class HillTimer : NucleusBehaviour<HillTimerComponent>
{
}
```

`NucleusBehaviour<TComponent0>.Awake` calls `NetworkSystemObjectPool.RequireSystem` for you, asking for a system that carries a `HillTimerComponent`. Once that system links, the base class hands it back through the `Component` property. Before linking, `Component` is null.

Where to read and write

`Component` is only safe to use from the lifecycle hooks `NucleusBehaviourBase` raises once a system is linked: `OnSystemLinked`, `OnServerStarted`, `OnClientStarted` (and their `OnServerStopped` / `OnClientStopped` / `OnSystemUnlinked` counterparts). A write attempted from `Awake`, or from any point before the system links, has nowhere to go — `Component` is still null, so it isn't a matter of the write being ignored, it's a `NullReferenceException` waiting to happen.

`OnSystemLinked` fires once, when the system arrives, whether that happens before, after, or exactly as this peer's own role starts. `OnServerStarted` and `OnClientStarted` fire once each, whenever that peer's role comes up while a system is linked — including immediately, if the role was already up at link time. A component that arms itself on `OnSystemLinked` for the server alone still needs a role check, because every peer's copy of the behaviour links the system, not just the server's:

```
protected override void OnSystemLinked()
{
    if (!IsController(ControllerType.Server))
        return;

    Component.SecondsRemaining.Value = _survivalSeconds;
}
```

Guarding writes

`IsController(ControllerType)` reports whether this peer controls the linked system; `IsStarted(Invoker)` reports whether a named role (`Invoker.Server` or `Invoker.Client`) is up, for

a behaviour whose system has linked. Both answer false while no system is linked, so they're safe to call anywhere, unlike reading `Component` directly.

When a failed guard is a bug rather than an expected outcome, use the loud forms instead: `EnsureIsController(ControllerType)` and `EnsureIsStarted(Invoker)` return the same answer but also log a warning naming the calling member, once per calling member on each behaviour. Reach for the loud forms in one-off code that must never run on the wrong peer - a button handler, an RPC. A per-tick write like the one below runs on every peer every tick, and a non-controlling peer failing the check there is the normal, expected outcome, not a bug, so use the quiet form:

```
protected override void OnEarlyFixedUpdate(StepDelta stepDelta, SimulationContext simulati
onContext)
{
    if (!IsController(ControllerType.Server))
        return;

    Component.SecondsRemaining.Value = Mathf.Max(Component.SecondsRemaining.Value - _secon
dsPerTick, 0f);
}
```

Reading a replicated value has no such guard: every peer's `Component` holds whatever the wire last delivered, and reading it is always safe once a system is linked.

The GameObject side

`NucleusBehaviourBase` carries `[RequireComponent(typeof(NetworkSystemObject))]`, so any `NucleusBehaviour<TComponent0>` needs a `NetworkSystemObject` marker on the same GameObject. That marker is `[DisallowMultipleComponent]` — one per GameObject — but it isn't limited to hosting a single system. Several `NucleusBehaviour` scripts on the same GameObject, each requiring a different component, link their systems together into one `NetworkSystemGroup`, so one marker can host a group of systems rather than just one.

Member surface

`NetworkMember<T0>` has more surface than a plain getter/setter — previous-value access, interpolation, constructor options for how it replicates. See the API reference for `NetworkMember<T0>`.

How replicated state works

See Start Here / How replication works for the pipeline overview. This page covers only the member-and-component contract every other page in this section assumes.

Containment

A `NetworkSystem` holds `NetworkComponents`, up to 64 per system. A `NetworkComponent` holds `NetworkMembers`, at most 64, one for each bit of the `ulong memberFlags` that `OnMembersChanged` reports. The generator does not check that limit on a component, so one that declares more still builds but does not replicate correctly. `Constants.MaximumNetworkMemberCount` (63) is a separate limit, on the structs and classes a member's value is made of: the generator reports `SERIALIZERS000` once such a type has 63 serialized fields and properties, so 62 is the most one can hold.

Data lives in components, not in the `NetworkSystem` subclass itself. `NetworkSystem` tracks identity, lifecycle, and observers; the values that actually replicate are declared on the `NetworkComponents` attached to it. A component exposes its own `Write`, `WriteDelta`, `Read`, and `ReadDelta`, and it is these that generated code fills in per member.

Full versus delta

A member simply carries the value as of the last write. There is no separate late-joiner payload to author.

When a new observer starts watching a system, that system serves a full snapshot of every component to that observer; every serialization after that is a delta of what changed. This applies uniformly: a full, a recovery serve, and an interest serve all ride the same absolute encoding, while an ordinary tick's serialize rides a delta. A member's author never writes two versions of anything - the full path and the delta path both read the same current value, just at different granularity.

One write pass, one read pass, per tick

Each tick raises `NetworkComponent.OnMembersChanged(ulong memberFlags, MemberChangeDirection memberChangeDirection)` up to twice per component: once for what this peer is sending, once for what it received.

- `MemberChangeDirection.Write` fires before any of the tick's writing runs, on the peer that made the change. It reports that the members were committed to the outbound pass, not that bytes left the process - a system with no observers, or one withheld from an observer, still raises this.
- `MemberChangeDirection.Read` fires after every packet of the inbound pass has landed, before reconciliation. A full apply (spawn, resync, reconcile, whole-component recovery) reports every member changed; a member-granular recovery reports only the members it repaired.

A host raises both directions in the same frame, write first, without waiting for its own packet to come back. Code that only cares about a remote change - anything calling `NetworkMemberBase.TryGetWritingClient` - must check for `MemberChangeDirection.Read`; that claim only resolves under a read.

Who may write

By default, `StateWriteAccess.Controller` (the enum's `0` value): the server, or the single controlling client, may write a system's state. Letting other observers write as well (`StateWriteAccess.AnyClient`) is a Pro opt-in, covered on its own page. A free build has no `StateWriteAccess` type, so there only the controller writes.

What a change costs

Four independent axes decide the price of a member changing, each covered on its own page:

- **Accuracy** - how finely a value's change is compared and encoded.
- **Compression level** - how tightly the wire encoding packs the value.
- **Transmission mode** - how the value's changes travel across ticks (e.g. absolute versus projected).
- **Send interval** - how often a changed member is actually allowed to send.

Vocabulary

Nucleus has its own terms; do not substitute names from another engine's netcode.

- **System**, not object: `NetworkSystem` is the networked entity.
- **Component**, not behaviour: `NetworkComponent` is a container of members within a system.
- **Member**, not variable: `NetworkMemberBase` is the individual replicated value.
- **Controller**, not owner: the client permitted to write a system's state by default.

Unity Types and System.Numerics

Declare members in System.Numerics, not UnityEngine

The engine core replicates `System.Numerics` types. A networked member is declared as `NetworkMember<System.Numerics.Vector3>`, never `NetworkMember<UnityEngine.Vector3>`. Nucleus registers its vector and rotation handling, such as interpolation and Divine projection, for the `System.Numerics` types only. A member declared with a UnityEngine type still builds, but the generator treats the type like any struct of your own: it writes a plain serializer from the type's public fields and settable properties, and none of that handling applies. For `UnityEngine.Quaternion` that plain serializer also picks up the settable `eulerAngles` property, so every value carries the rotation twice.

The shipped `UnityNavMeshAgentComponent` follows this exactly:

```
using NumericsVector3 = System.Numerics.Vector3;

public partial class UnityNavMeshAgentComponent : NetworkComponent
{
    public readonly NetworkMember<NumericsVector3> NextPoint = new(CompressionLevel.Aggres
sive);
    public readonly NetworkMember<NumericsVector3> PreviousPoint = new(CompressionLevel.Ag
gressive);
}
```

The `NumericsVector3` alias is just a using directive disambiguating from `UnityEngine.Vector3` in the same file. It's the pattern to copy: alias `System.Numerics` types at the top of a Unity-facing script rather than writing the fully qualified name at every member declaration.

Convert at the Unity boundary

`UnityConversionExtensions` converts between the native type a member stores and the Unity type a `Transform` (or other Unity API) expects: `ToUnity()` for native to Unity, `ToNative()` for Unity to native. It covers every type both libraries define: `Vector2`, `Vector3`, `Vector4`, `Quaternion`, `Plane`, and `Matrix4x4`.

Call them at the moment you read a member into a `Transform` or write a `Transform` into a member, not before. `UnityNavMeshAgentComponent` does exactly this in its Unity-facing properties:

```
public Vector3 ReplicatedNextPoint
{
    get => NextPoint.Value.ToUnity();
    set => NextPoint.Value = value.ToNative();
}
```

Everything above that boundary — the `NetworkComponent`, its `NetworkMember<T0>` fields — stays native. Everything below it — the `MonoBehaviour`, the `Transform`, `NavMeshAgent` — stays Unity. Nothing in between sees both.

Types with generated serializers already available

Besides the `System.Numerics` vectors (`Vector2`, `Vector3`, `Vector4`), these CodeBoost types already have generated serializers and can be used in a `NetworkMember<T>` from Unity code without any attribute of your own: `Vector2Int`, `Vector3Int32`, `Vector4Int`, and `UnifiedColor`.

All seven are reached through `ExternalNetworkTypes`, a marker struct in `Nucleus.Serializers`

carrying `[NetworkType]` on behalf of types declared in assemblies nobody here can annotate. Nothing reads or constructs the struct at runtime; the generator only walks its field types to know which external types need serializers.

Your own structs need no attribute

A struct you declare yourself, made entirely of natively-serializable fields, needs no `[NetworkType]` attribute at all as long as a `NetworkMember<T>` somewhere reaches it. The generator discovers it through that member, the same way it discovers any other networked type.

The error you get instead of a crash

An unserializable member type is caught at build time, not at runtime. The generator reports a `SERIALIZERS`-series diagnostic (`SERIALIZERS000` as an error, `SERIALIZERS001` as a warning) naming the member. A member declared as `NetworkMember<UnityEngine.Vector3>` is not caught this way: the generator can read the type's public fields, so it builds without a diagnostic and replicates through the plain serializer described above.

Adding a type you can't annotate

For a third-party type you don't own and can't put `[NetworkType]` on, see the `ExternalNetworkTypes` reference for how to add a marker for it.

Several systems on one object

Driving the core API directly? See System groups

Why split one object into several systems

`NetworkSystemObject` is a marker: it links a GameObject to one or more `NetworkSystem` instances, not to exactly one. Splitting an object's networked state across several systems lets each one replicate on its own terms. Interest can stop one member for a distant observer — its transform, say — while another member on the same GameObject, one whose changes matter at any range, keeps replicating. The marker only reclaims the GameObject once every member has left it, so the object itself does not flicker in and out while its parts come and go independently.

How it looks on the marker

`NetworkSystemObject` tracks every system linked to it:

- `Systems` — every linked system, in the order each one linked.
- `LinkedSystemCount` — how many are currently linked.
- `System` — the first-linked system, promoted to the next-oldest member if it leaves while others remain.
- `TryGetFirstSystem(out NetworkSystem networkSystem)` — the same first-linked system, as a bool-returning lookup.

`SystemsLinked` fires once the marker's whole set is assembled: an ungrouped object is a batch of one and fires on its first link, while a grouped object fires when its linked count reaches the group's replicated member count. A handler that subscribes after the set is already complete is invoked immediately rather than waiting for a completion that already happened, so a script that shows up late still runs.

```
void Awake()
{
    GetComponent<NetworkSystemObject>().SystemsLinked += OnSystemsLinked;
}

void OnSystemsLinked(NetworkSystemObject networkSystemObject)
{
    if (networkSystemObject.TryGetFirstSystem(out NetworkSystem primary))
    {
        // Every member the group carries is present now — register remote-call
        // handlers or other object-wide work against the primary system here.
    }
}
```

Renting a second system from a second script

`NetworkSystemObjectPool.RequireSystem<TSystem, TComponent0>(this, acquiredHandler, releasedHandler)` is what a component on a marker calls to declare its need, rather than renting directly: it resolves what to rent from the marker's situation and is timing-tolerant, so it works whether the script's Awake runs before or after the marker links a matching member. `NetworkAnimator`, for example, declares its own requirement this way:

```
NetworkSystemObjectPool.RequireSystem<NetworkSystem, UnityAnimatorComponent>(this, OnSyste
mAcquired, OnSystemReleased);
```

A second script on the same GameObject calls `RequireSystem` for its own composition in its own `Awake`,

exactly the same way. Each script's requirement is served once: `RequireSystem` records it and resolves it from whatever the marker already links or from a fresh rent, so the script never retries or guards against a double rent itself. The raw, context-aware `TryRent<TSystem, TComponent0>` that `RequireSystem` calls into is the one-shot version, meant for a driver that wants its system back synchronously at a moment of its own choosing. It is not refused on a marker that already links a system: each call rents and links another member, so calling it twice for the same need gives you two systems.

How the spawn side ties them together

On the server, the marker lazily rents a `NetworkSystemGroup` the first time a dynamic prefab's system is rented, and every later rent against that marker adds to the same group, so every member shares one non-unset `GroupId`.

On a receiver, the Unity integration's spawn handler is what ties that GroupId back to a single GameObject. It keeps a map of the marker hosting each active group, keyed by `GroupId`. The first member of a group to arrive finds nothing registered, so it instantiates the prefab and registers its marker against the group. A later member arriving with the same GroupId finds that marker already registered and links onto it instead of instantiating a second prefab.

Despawn behaviour

`NetworkSystemObject.Despawn()` stops every system currently linked to the marker, so a grouped object's members leave together rather than one at a time. Stopping a single member with `SystemManager.EnsureStopSystem` stops only that member; the others stay linked and the GameObject stays. On a receiver, each member's despawn arrives and is unlinked in turn; the spawn handler checks `LinkedSystemCount` after each unlink and only reclaims the GameObject once it reaches zero. Until the last member leaves, the object holds together.

Reclaiming means one of two things: a scene object is destroyed outright, since it is one fixed instance rather than a poolable prefab; a dynamic prefab is returned to the configured `INetworkPrefabPool` for reuse by a later spawn. Either way, the marker's group is returned to its pool once `LinkedSystemCount` reaches zero, so a reused instance rents a fresh group on its next spawn rather than inheriting the old GroupId.

An interest stop that keeps the object visible for one peer, rather than despawning it, is handled separately: the handler retains the GameObject keyed by the leaving system's Id instead of reclaiming it, so the spawn that arrives when that peer re-enters interest links onto the same GameObject rather than instantiating a second one.

Systems, Objects and Spawning

Pages in this category:

- NetworkSystemObject component
- NucleusBehaviour: the script base class
- What a system is
- Player objects
- Prefab identity and the prefab collection
- Spawning and despawning objects
- Carrying and re-parenting objects
- Pooling spawned objects
- Several systems on one object
- Client-predicted spawning in Unity

NetworkSystemObject component

This marker is the one link between a GameObject and its systems

`NetworkSystemObject` is a `MonoBehaviour` and the single point of correlation between a GameObject and one or more `NetworkSystem` instances. It is `[DisallowMultipleComponent]`: exactly one marker per GameObject, on the root of a networked prefab or scene object.

An object normally links to one system. Several systems link to the same marker when a `NetworkSystemGroup` hosts more than one system on that GameObject; the first to link becomes `System`, and the rest are reachable through `Systems`.

Inspector fields

Required Bundle Override

- **Override Required Bundle** (`_requiredBundleOverrideEnabled`) — when off, the content bundle a client must hold before this object may spawn for it is read from the prefab. When on, `Required Bundle Id` is used instead.
- **Required Bundle Id** (`_requiredBundleId`) — the bundle to require when the override is on. An explicit `0` means "this object needs no content, whatever shard it ships in," which the toggle distinguishes from never having configured an override at all.

Controller Retention Override

- **Override On Disconnect** (`_controllerRetentionOverrideEnabled`) — when off, the object takes its controller-retention behavior from the Unity System Manager's default. When on, `On Controller Disconnect` decides it for this object instead.
- **On Controller Disconnect** (`_controllerRetentionPolicy`) — `ControllerRetentionPolicy`, defaulting to `Retain`. Control is surrendered when the controller disconnects either way; this only decides whether the object is held so a returning player can be handed it back. The override is pushed onto the object's `NetworkSystemGroup.ControllerRetentionPolicyOverride` when the group is rented, not read continuously.

Predicted Spawn Policy

`_predictedSpawnPolicy`, a `PredictedSpawnPolicy`, defaulting to `None`. It is authored on the prefab rather than set in code so every peer reads the same answer with nothing going on the wire for it. It is applied to each system as it links to this marker (`NetworkSystem.PredictedSpawnPolicy`).

Stamped identity fields

Three fields are stamped by editor tooling and shown read-only in the inspector: a scene object's `Scene Object Id`, or a prefab's `Prefab Id` and `Prefab Bundle Id` pair. The one hand edit is clearing a duplicated prefab's `_prefabId` to `0` in its `.prefab` file; see Prefab identity and the prefab collection. The inspector shows one or the other depending on `IsSceneObject`, never all three at once.

Runtime read surface

Member	Type	Meaning

`System`	`NetworkSystem`	The first system linked to this object, or `null`.
`Systems`	`IReadOnlyCollection<NetworkSystem>`	Every system currently linked to this object.
`LinkedSystemCount`	`int`	The number of systems currently linked.
`TryGetFirstSystem(out NetworkSystem)`	`bool`	Resolves `System`; the canonical member to scope object-wide work against, such as an `IRpcScope` handler registration.
`PrefabId`	`ushort`	The prefab identifier assigned by the prefab collection build, or zero when unset.
`PrefabBundleId`	`ushort`	The content bundle whose collection stamped this prefab.
`RequiredBundleId`	`ushort`	The override when `_requiredBundleOverrideEnabled` is set, otherwise `PrefabBundleId`.
`SceneObjectId`	`uint`	This object's identifier within its scene, or zero when it is not a scene object.
`IsSceneObject`	`bool`	True when `SceneObjectId != 0`.
`PredictedSpawnPolicy`	`Nucleus.Systems.PredictedSpawnPolicy`	What a client may do to this object ahead of the server agreeing, as authored on the prefab.

Events this component raises

- **`SystemLinked(NetworkSystem)`** — raised once each time a system links to this marker.
- **`SystemUnlinked(NetworkSystem)`** — raised when a linked system is cleared on despawn.
- **`SystemsLinked(NetworkSystemObject)`** — raised once the object's whole set of members has linked: an ungrouped object fires on its first link, a grouped object fires when its linked count reaches the group's replicated member count. It re-fires each time a completed set is assembled, including after the group is later grown by a further spawn. A handler added after the object is already complete is invoked immediately on subscription, so a late subscriber is not missed.
- **`SystemsUnlinked(NetworkSystemObject)`** — raised once the object's last member unlinks.

Despawn and reverse lookup

`Despawn()` stops every system linked to this object on the server, replicating the despawn to observers; a call with no linked system is a no-op. It is the pooling counterpart to destroying the GameObject, which despawns network-wide through `OnDestroy`.

`NetworkSystemObject.TryGetLinked(NetworkSystem, out NetworkSystemObject)` is the reverse of `System`: given a `NetworkSystem` handed to game code by the engine, it resolves the marker linked to it, or returns false if none is linked.

RPC scope

The marker links to one or more `NetworkSystems`, but it is the `NetworkSystem` itself, not `NetworkSystemObject`, that implements `IRpcScope`. Register remote-call handlers against a system resolved through `System`, `Systems`, or `TryGetFirstSystem`, never against the marker.

NucleusBehaviour: the script base class

Driving the core API directly? See Renting and finding systems in code.

The base class requires a system for you on Awake

`NucleusBehaviour<TComponent0>` is the base class for a gameplay `MonoBehaviour` that belongs to one networked system. On `Awake` it requires that system for you, resolving the `CoreManager` and calling `NetworkSystemObjectPool.RequireSystem`, and hands back a typed `Component` property once the system is acquired. A script inheriting it never writes the manager lookup, the `RequireSystem` call, or the unlink/unsubscribe that a hand-rolled version would otherwise repeat.

The class splits in two:

- `NucleusBehaviourBase` is the non-generic half: the `NetworkSystem` reference, role and control checks, the loud guards, and the eleven per-step virtuals. It never declares a required system, so it is not the type a script inherits directly.
- `NucleusBehaviour<TComponent0>` requires a `NetworkSystem` carrying `TComponent0` and exposes it as `Component`.

A script that only needs the role and control checks and the hooks, on an object where another component (a `NetworkTransform` or a `ProjectedRigidbody`, say) already declares the system, inherits the non-generic `NucleusBehaviour` instead. It declares nothing and adopts whichever system the object's `NetworkSystemObject` links, so something else on the object must declare one or it waits for the object's whole life. If the system it adopted is stopped on its own while the object keeps another, it re-adopts the one that remains at the end of that frame; a whole-object despawn unlinks it once and adopts nothing.

A generated family widens this to more component types: `NucleusBehaviour<TSystem, TComponent0>` through `NucleusBehaviour<TSystem, TComponent0, TComponent1, ..., TComponent31>`, one overload per arity from 1 to 32 component types, each naming the system type explicitly and each calling the matching `NetworkSystemObjectPool.RequireSystem<TSystem, TComponent0, ...>` overload. Use the generated form when a script needs a system that isn't just `NetworkSystem`, or needs more than one component off it.

Inherited properties

Property	Type	Meaning
`NetworkSystem`	`NetworkSystem`	The linked system, or null before linking and after unlinking.
`CoreManager`	`CoreManager`	The manager resolved in `Awake`, or null when none was bound.
`IsHostStarted`	`bool`	True when this peer runs both server and client roles at once, for a linked system.
`Component`	`TComponent0`	The typed component this behaviour required, or null before `OnSystemLinked` and after `OnSystemUnlinked`.

Role and control checks

- `IsController(ControllerType controllerType)` returns `NetworkSystem.IsController(controllerType)`, or false while no system is linked.
- `IsStarted(Invoker invoker)` returns whether this peer's server or client role is started, for a linked system; false until one links.

Both have loud counterparts that log instead of failing silently:

- `EnsureIsController(ControllerType controllerType, [CallerMemberName] string callerMemberName = null)`
- `EnsureIsStarted(Invoker invoker, [CallerMemberName] string callerMemberName = null)`

Each returns the same answer as its plain counterpart, and when that answer is false, logs a warning naming the calling member (supplied automatically via `[CallerMemberName]`). Each behaviour warns once per calling member and argument, so a guard that keeps failing from the same place warns the first time and then stays quiet. Reach for the loud form where a failure means a real bug worth surfacing loudly: a one-off action like handling a button press or an RPC, not a per-tick callback. A per-tick write like `OnEarlySerialize` runs every tick on every peer, so a non-controlling peer failing the check is the normal, expected case, not a bug. Use the plain form there so an expected outcome logs no warning at all:

```
protected override void OnEarlySerialize(StepDelta stepDelta)
{
    if (!IsController(ControllerType.AnyController))
        return;

    // Safe to write controlled state here.
}
```

Lifecycle hooks

Override these `protected virtual` methods on `NucleusBehaviourBase`:

Hook	Guarantee
`OnSystemLinked()`	Called once `NetworkSystem` is linked.
`OnSystemUnlinked()`	Called as `NetworkSystem` unlinks, while it is still readable for the duration of the call, including when this behaviour is destroyed while still linked. A pooled, recycled instance resets its per-life state here.
`OnServerStarted()`	Called once this peer's server role starts, with a system already linked.
`OnServerStopped()`	Called once this peer's server role stops while a system is still linked.
`OnClientStarted()`	The client-role counterpart of `OnServerStarted`.
`OnClientStopped()`	The client-role counterpart of `OnServerStopped`.

`OnControllerChanged(Connection previousControllerConnection, Connection currentControllerConnection)`	Called on every peer when control of the linked system moves.

Linking adopts whichever roles are already started rather than waiting for an edge: `LinkSystem` reseeds its tracked server/client flags from the live state and immediately raises `OnServerStarted`/`OnClientStarted` for any role already up, instead of only firing on a later start/stop transition. A scene object's `Awake` runs on every peer regardless of which of them already has a role running, so a script that only reacted to edges would miss a role that started before it linked.

Overriding Awake, OnEnable, OnDisable and OnDestroy

The base classes do required setup in their own `Awake`, `OnEnable`, `OnDisable` and `OnDestroy`: `Awake` resolves the `CoreManager` and, in `NucleusBehaviour<TComponent0>`, requires the system, `OnEnable` and `OnDisable` take and drop the per-step registration, and `OnDestroy` unlinks the system. Unity calls only the most derived method of each name, so a script that needs one of these must override it and call the base method:

```
protected override void Awake()
{
    base.Awake();

    _renderer = GetComponent<Renderer>();
}
```

Call the base method first, unless the work must be ready before the system links: `base.Awake()` can link a system that is already present and raise `OnSystemLinked` on the spot, so a reference that hook reads is cached before the call. Setup that wants the system anyway belongs in `OnSystemLinked`.

The build enforces both halves of this rule, as errors:

Id	Reported when
`NUCLEUS001`	A class deriving, directly or indirectly, from `NucleusBehaviourBase` declares `Awake`, `OnEnable`, `OnDisable` or `OnDestroy` without `override`, hiding the base method.
`NUCLEUS002`	An override of one of those four methods never calls the base method.

Either mistake would otherwise leave the behaviour unlinked with nothing logged. The lifecycle hooks above and the per-step virtuals below are empty in the base class, so an override of those may skip the base call and is never reported.

Per-step virtuals

`NucleusBehaviourBase` declares eleven `protected virtual void On...(StepDelta stepDelta)` methods, one per loop step, in loop order:

```
OnEarlyVariableUpdate
OnEarlyStateUpdate
OnLateStateUpdate
OnReconcile
OnEarlyFixedUpdate
OnLateFixedUpdate
OnVariableUpdate
OnEarlySerialize
OnLateSerialize
OnTickAdvance
OnLateVariableUpdate
```

`OnEarlyVariableUpdate`, `OnVariableUpdate` and `OnLateVariableUpdate` run every frame. `OnEarlyStateUpdate`, `OnLateStateUpdate`, `OnReconcile`, `OnEarlySerialize`, `OnLateSerialize` and `OnTickAdvance` run once per tick, on the frame that ticks. `OnEarlyFixedUpdate` and `OnLateFixedUpdate` run once per elapsed tick interval, so they can run twice on one frame after a long frame. They follow the linked system's simulation rather than the loop alone: each takes a `SimulationContext` naming the pass, and on a client that predicts the object they run again for every tick a reconcile replays (see Replaying a correction in Unity).

`Awake` reflects over these once to find which ones this instance's concrete type overrides, and only those steps are ever dispatched to. Registration for the loop steps a type overrides is taken and dropped alongside `OnEnable`/`OnDisable` (and tied to whether a system is linked), so a disabled or pooled-and-despawned behaviour costs nothing per tick.

See the tick-loop pages for what each step means and when it runs.

To move an object every frame, override `OnVariableUpdate` and move it by `stepDelta.NormalizedDelta` rather than moving it in Unity's `Update` by `Time.deltaTime`. The normalized delta keeps the distance moved in each tick even, which costs far less to replicate. See Tick rate and the loop in Unity.

CanStartSystem

`protected virtual bool CanStartSystem => true` controls whether the server may start the required system as soon as it is rented. It's read once, when the required system is declared, so a later change to it has no effect, and it's ignored on a receiving client, which is handed a system that already started elsewhere. Override it to return false for an object that must write its opening state before the spawn is announced, then start the system itself.

A script that needs an undeclared system

`NucleusBehaviour<TComponent0>` (and its generated overloads) only give you the system(s) named in the type parameters. If a script needs a different system it didn't declare, rent or find it separately through the core API rather than adding it to the type parameter list for a type you don't otherwise need.

Player objects

NetworkPlayerSpawner gives every client one object of its own

`NetworkPlayerSpawner` is a `MonoBehaviour` that gives every authenticated client one object of its own. On the server it spawns one prefab per client, hands that client control of the spawned object, and keeps a connection-to-object relationship so the rest of the game can ask which object belongs to whom.

The engine itself has no player concept, and this component does not add one. A player object is an ordinary networked object that happens to be controlled by one client, so everything the engine already does for a controlled object applies unchanged: replication, interest, client state routing, write permission, controller retention. What this component adds is the one decision the engine deliberately leaves out: whether an object needs creating for a given client at all.

Only the server spawns. The component observes its role from the transport (`IsServerStarted`) rather than assuming it, because a server start can fail.

Inspector fields

- **Player Prefab** (`GameObject`) — the networked prefab spawned for each client. It needs a `NetworkSystemObject` marker; without one, `Awake` logs an error and disables the component.
- **Spawn Points** (`Transform[]`) — poses new objects are placed on, taken in turn and wrapping when exhausted. Leave this empty to spawn every player on the spawner's own transform.
- **Disconnect Mode** (`NetworkPlayerDisconnectMode`) — what happens to a player's object when that player's connection leaves. Defaults to `Despawn`.

Disconnect modes

`NetworkPlayerDisconnectMode` has two values:

- **`Despawn`** (the default) — the player's object leaves the world with them, and a returning player is always given a new one. Needs no identity worth trusting, since every join is a first spawn.
- **`RetainForReturn`** — the player's object stays in the world, uncontrolled, and is handed back to the same player if they return before the retention record lapses. The spawner marks every system on the object `ControllerRetentionPolicy.Retain`, stores the token the departure issues against `Connection.Identity`, and redeems it when that identity returns. An object whose record expires, is evicted, or empties is despawned at that point.

`RetainForReturn` is only as trustworthy as the identity behind it. The default `AddressClientAuthenticator` identifies a client by the address it connected from, which two players behind one NAT can share — under it, this mode can hand the second player the first player's object. Use an `IClientAuthenticator` that issues a real per-player identity if you turn this on.

The public members

```
public bool IsServerStarted { get; }
public int PlayerObjectCount { get; }

public bool EnsurePlayerObject(Connection connection);

public bool TryGetPlayerObject(uint connectionId, out NetworkSystemObject playerNetworkSys
temObject);
public bool TryGetPlayerObject(Connection connection, out NetworkSystemObject playerNetwor
kSystemObject);
```

`EnsurePlayerObject` is the only route to an object, and it is idempotent: a client that already holds one is a no-op, not a second spawn. It resolves in order — this spawner's own record, then whatever the connection already controls (the case after a world adoption), then a retention record redeemed by identity, and only then a fresh spawn. Call it as often as you want; asking twice for the same connection never produces a second object.

`TryGetPlayerObject` has two overloads: by connection identifier (`Connection.Id`) or by the `Connection` itself. Both only answer for objects this spawner created or took responsibility for.

`PlayerObjectCount` is how many clients this spawner currently holds an object for.

Events this component raises

```
public event PlayerObjectSpawnedHandler PlayerObjectSpawned;
public event PlayerObjectReclaimedHandler PlayerObjectReclaimed;
public event PlayerObjectDespawnedHandler PlayerObjectDespawned;
```

- **`PlayerObjectSpawned(Connection connection, NetworkSystemObject playerNetworkSystemObject)`** — raised on the server when a client is given a newly created object, after that client has been made its controller.
- **`PlayerObjectReclaimed(Connection connection, NetworkSystemObject playerNetworkSystemObject)`** — raised when a client is given an object that already existed, whether redeemed from a retention record or found already under that client's control after this peer adopted a world. The object carries whatever state it had when its previous controller left, so anything a game resets for a fresh player has to run here too, and has to be safe to run on a used object.
- **`PlayerObjectDespawned(NetworkSystemObject playerNetworkSystemObject)`** — raised when a player object leaves the world, whether because its player left under `NetworkPlayerDisconnectMode.Despawn` or because the retention record holding it lapsed unredeemed.

The adopt pass

`Awake` resolves the component's managers from the bound `CoreManager`, subscribes to `ClientAuthenticated` and the connection state-change events, and then — only if `IsServerStarted` is already true — sweeps every currently active, authenticated client and calls `EnsurePlayerObject` on each.

That sweep exists because an event does not fire retroactively. A peer that was a client before it became the server already holds connections that authenticated before this component's subscription existed, and an emulated or in-memory connection is authenticated the moment it connects, never passed to the authenticator, so it raises no event at all. Without the adopt pass those clients would never be given an object. The same sweep runs again whenever the local connection's server role turns on, and losing that role clears every relationship this spawner is holding, since the objects belonged to the session that just ended.

Writing your own spawner

`NetworkPlayerSpawner` adds nothing the engine doesn't already support — it just makes the per-joining-client decision for you. Writing your own is the same two calls:

```
private void OnClientAuthenticated(Connection connection)
{
    // spawn your prefab, get its NetworkSystemObject, then:
    playerNetworkSystem.SetController(connection);
}
```

plus the adopt-pass sweep over already-authenticated connections at startup, for the same reason: an event you subscribe to in `Awake` never fires for a client that authenticated before your component existed.

Client-predicted spawning in Unity

Driving the core API directly? See Predicted spawn and despawn requests.

Predicted spawning is a Pro feature. It lets a client create a prefab instance the instant it decides to, instead of waiting a round trip for the server to spawn it. The server still decides whether the object survives: it can confirm the client's guess or reject it, and the client finds out which.

Marking the prefab

A `NetworkSystemObject` carries a **Predicted Spawn Policy** field: what a client may do to this object ahead of the server agreeing to it. The values are flags and can combine:

- `None` — the server alone creates and removes the object. This is the default.
- `Spawn` — a client may create the object at once and ask the server for it afterwards.
- `Despawn` — a client may remove the object at once and ask the server to remove it afterwards. Independent of `Spawn`: an object the server created can still be despawned predictively.
- `SpawnAndDespawn` — both of the above.
- `AllowUndeclaredComposition` — permits a predicted spawn whose component composition no call site in this build declares. Every composition a build can produce is normally named at some call site, so refusing an undeclared one is the safe default; this flag opts an object out of that check.

The policy is authored on the prefab, not passed in code, because it is never replicated. Every peer reads its own copy of the same prefab, so every peer answers the same question the same way without a bit of it going on the wire. That matters most on the server: it judges a predicted request against its own copy of the object, never against what the asking client claims.

Asking for one

The common case is renting and asking in one step, through the normal pool rent. For an object whose opening state matters — the shot's origin and velocity, for example — rent it unstarted, write that state, then ask:

```
GameObject projectileObject = Instantiate(_projectilePrefab, _muzzle.position, Quaternion.
LookRotation(direction));
NetworkSystemObject marker = projectileObject.GetComponent<NetworkSystemObject>();

// Rented unstarted, so the shot can be written into the object before the request that ca
rries it is sent.
if (!NetworkSystemObjectPool.TryRent<NetworkSystem, PredictedProjectileComponent>(marker,
out NetworkSystem projectileSystem, canStartSystem: false))
{
    Destroy(projectileObject);
    return;
}

if (projectileSystem.TryGetComponent(out PredictedProjectileComponent projectileComponent)
)
{
    projectileComponent.Origin.Value = _muzzle.position.ToNative();
    projectileComponent.Velocity.Value = (direction * _projectileSpeed).ToNative();
}

if (NetworkSystemObjectPool.EnsureStartPredicted(marker, projectileSystem, out PredictedSp
awnRefusal predictedSpawnRefusal))
{
    // Started and sent. Wait for PredictedSpawnConfirmed or PredictedSpawnRejected.
    return;
}

// Refused locally, before anything was sent to the server.
Destroy(projectileObject);
```

`EnsureStartPredicted` sends the request carrying a snapshot of the object taken as it is called, so anything the server needs to know about the object's birth has to be written before this runs. A request travels only once `predictedNetworkSystem` was rented with `canStartSystem: false` (or a caller otherwise holds an unstarted, grouped system) and `networkSystemObject.PredictedSpawnPolicy` includes `Spawn`.

A `false` return with a `PredictedSpawnRefusal` is this client's own framework declining before anything was sent — for example, its lease of group identifiers from the server has run dry. It is not the server's answer; nothing reached the server yet, and the object was never created.

Watching the outcome

The server's answer arrives later, on the system the marker linked:

- `PredictedSpawnConfirmed` — raised when the server's snapshot for this object arrives, meaning the server agrees the object exists. It fires before the snapshot's own state is applied, so a handler still reads the values the client predicted.
- `PredictedSpawnRejected(PredictedSpawnRejectReason reason)` — raised when the server refuses the object, immediately before the local copy is torn down. Subscribe here to play whatever stands in for the thing that did not happen.
- `IsPredictedSpawnOrigin` — true on the client that created this object ahead of the server. It stays true after confirmation; it answers *which* peer predicted the object, not whether the prediction is still outstanding.
- `IsPredictedSpawnPending` — true while a predicted spawn is still waiting on the server.

```
projectileSystem.PredictedSpawnConfirmed += OnShotConfirmed;
projectileSystem.PredictedSpawnRejected += reason => OnShotRejected(projectileObject, reas
on);
```

When the server never answers

The request is sent once, and the client's upstream redundancy (`ClientRedundancy`) resends it on the ticks after, so a single lost packet does not lose it. If no answer arrives at all, the client gives up and despawns its local copy, raising `PredictedSpawnRejected` with `NotPermitted` first, as for an ordinary refusal. It waits two round trips, but never less than half a second and never more than three seconds, so on a bad link an object nobody agreed to does not stay on screen for long.

An answer can still arrive after the client gave up: the server admitted the spawn, but its confirmation was lost until after the timeout. The client then receives the server's object as an ordinary spawn, with the server's values, so both peers still end up holding the same object.

Predicted despawn

The same system exposes the mirror image for removal, gated by the `Despawn` flag:

- `IsPredictedDespawnPending` — true between a client asking the server to remove the object and hearing back. The object is not despawned locally while this is set: it stays started, routed, and replicating, and inbound state still applies to it. The game decides what to show in the meantime.
- `PredictedDespawnPending` — raised when the client's removal request goes out.
- `PredictedDespawnRejected(PredictedDespawnRejectReason reason)` — raised when the server refuses the removal. The object was never taken away, so it simply carries on; this is the same instance throughout, with nothing to restore.

What a refused prediction looks like

A rejected spawn is visible for a moment and then gone: the object existed locally from the frame it was created, and it disappears a round trip later when `PredictedSpawnRejected` fires. Left unhandled, that reads as an object popping out of existence for no reason.

Handle the rejection to make it legible instead of jarring: a small poof, a sound, a HUD counter ticking up, whatever suits the object. The demo project's predicted-projectile gun does this for a fire-rate limit — every shot appears immediately, and a shot the server rejects for firing too fast destroys itself with a log line explaining why, rather than just vanishing.

Authority side

Confirming or rejecting a predicted spawn, granting the identifier lease it rides on, and choosing the refusal reason are all decided on the server. See Predicted spawn and despawn requests for the validators, the lease, and the full set of refusal reasons.

Prefab identity and the prefab collection

The prefab collection

`NetworkPrefabCollection` is a `ScriptableObject` that lists the networked prefabs a client can instantiate for a dynamic spawn before the prefab is otherwise in hand. Create one from **Assets > Create > Nucleus > Network Prefab Collection**.

Each collection is one content shard:

- **Prefabs** — the networked prefabs this shard ships, in deterministic order.
- **PrefabBundleId** — the bundle identifier this shard registers its prefabs under. Zero is the base build's default shard.
- **BundleAssemblyName** — the simple name of the assembly whose `NetworkBundle` attribute supplies this shard's bundle identifier. Empty means the base build's default shard.

`RegisterAll()` claims the shard's bundle identifier and registers every listed prefab into the runtime registry; call it when the shard's content loads. `UnregisterAll()` removes those prefabs and releases the bundle claim; call it when the content unloads. A prefab with an unset prefab identifier logs an error and is skipped instead of registering.

Rebuilding the collection

Prefab identity is not something you assign by hand. Run **Nucleus > Rebuild Network Prefab Collection** and the editor does it for you.

The build step (`NetworkPrefabCollectionBuilder`) scans every prefab in the project for a `NetworkSystemObject`, groups the networked ones by the asset bundle Unity has them assigned to, and writes one shard's `NetworkPrefabCollection` per group. The collection with no asset bundle assignment (the one under `Resources`) owns the base build's shard; a collection assigned to asset bundle `x` owns the prefabs assigned to `x`. Nothing new has to be authored to express that membership — asset bundle assignment already decides it.

Within a shard, prefabs are ordered by asset GUID so every machine that scans the same assets produces the same ordering, then each prefab's `NetworkSystemObject` marker is stamped with a local prefab identifier and the shard's bundle identifier. Stamping is deterministic and does **not** renumber existing entries: an identifier a prefab already carries is kept as long as it is still unique within its shard, and only a new or colliding marker gets `highestPrefabId + 1`. Renumbering on every scan would mean adding one prefab silently repoints every prefab after it, so a content bundle built before the change would resolve to the wrong prefab against a client that rebuilt after it.

Run the rebuild whenever you:

- Add a new networked prefab.
- Delete a networked prefab.
- Change a prefab's asset-bundle assignment (moving it between shards).

The runtime registry

`NetworkPrefabRegistry` is the runtime lookup from a prefab's wire identity to the prefab a client instantiates for a dynamic spawn. It is populated as each shard's collection loads and cleared as it unloads.

- `EnsureClaimBundle(bundleId, collection)` claims a bundle identifier for a collection before its

prefabs register. A second collection trying to claim a bundle identifier already owned by a different loaded collection is an error.

- `Register(bundleId, localId, prefab)` adds a prefab under its `(bundle id, local id)` identity.
- `TryGet(bundleId, localId, out prefab)` resolves the prefab registered for an identity.
- `TryGetPredictedSpawnPolicy(bundleId, localId, out predictedSpawnPolicy)` resolves what a client may do to that prefab ahead of the server, from the policy recorded at registration.
- `ReleaseBundle(bundleId, collection)` releases a bundle identifier when its collection unloads.

The base build ships bundle zero. Each content shard claims and registers its own bundle identifier when its content loads, so separately built content never collides — as long as every shard declares a unique `NetworkBundle` identifier on its assembly.

The duplicated-id trap

A prefab's identity is stamped onto the prefab asset itself, in the `NetworkSystemObject` marker's serialized fields. Duplicating a prefab in the Project window duplicates that stamp along with everything else, so the copy carries the *same* prefab identifier as the original.

Two prefabs sharing one identity means a spawn of one can build the other. At runtime `NetworkPrefabRegistry.Register` keeps the first live prefab registered under an identity and logs an error naming both prefabs when a different one claims it, so every peer that registers the same collection at least resolves the identity to the same prefab, and the clash shows in the Console instead of being settled silently.

You are told well before that, though. A player build fails while any networked prefab carries an identity a rebuild would change: a missing base collection, two collections resolving to the same prefab bundle, a prefab missing from its collection, a prefab with no id or with an id another prefab holds, or a prefab stamped with a different bundle than its collection. The build error lists each problem. To find the same problems without building, run **Nucleus > Validate Network Prefabs** (see Developer tools in the editor).

Running **Nucleus > Rebuild Network Prefab Collection** fixes it: the rebuild keeps an id that is set and unique and gives a duplicate the next free local id for its shard, logging a warning that names the renumbered prefab, the prefab that already held the id, and the new id. When two prefabs share an id, the one the collection already lists keeps it, so an original keeps its id over a new copy, which matters once a build or content bundle has shipped with it. Among prefabs the collection does not list yet, a base keeps the id over its own variant, and asset GUID order settles the rest. Then rebuild both the server and the client builds so every peer agrees.

Wire identity

A prefab's wire identity is the `(bundle id, local id)` pair stamped onto its `NetworkSystemObject` marker and used as the key into `NetworkPrefabRegistry`.

Pooling spawned objects

The default cost

Every dynamic spawn instantiates a GameObject, and every despawn destroys one. For a busy spawn loop - projectiles, hit effects, anything that comes and goes at rate - that is a GC allocation and a destroy call per instance, on top of whatever the object's own `Awake` does. The integration lets you swap that behavior out without touching the spawn code that triggers it.

The swap point: INetworkPrefabPool

`INetworkPrefabPool` is the interface the integration calls into on every dynamic spawn and despawn:

```
GameObject Rent(GameObject prefabGameObject, ushort prefabBundleId, ushort prefabLocalId,
Vector3 position, Quaternion rotation, Vector3 scale);

void Return(GameObject pooledGameObject, ushort prefabBundleId, ushort prefabLocalId);
```

`Rent` is called when a dynamic spawn needs an instance, `Return` when a despawn releases one. `prefabBundleId` and `prefabLocalId` are the prefab's wire identity - the same identity a `NetworkSystemObject` exposes as `PrefabBundleId` and `PrefabId` - and they key the pool: instances of different prefabs are never interchangeable.

The rent contract takes the pose, not just the prefab. `Rent` must apply `position`, `rotation` and `scale` before the instance runs any of its own code, because the integration already knows where a received spawn belongs before it asks for the instance. A script reading its transform from `Awake` or `OnEnable` needs to see that pose, not the prefab's authored one (which for most networked prefabs is the origin).

DefaultNetworkPrefabPool vs NetworkPrefabPool

`DefaultNetworkPrefabPool` is the baseline: `Rent` instantiates at the given pose, `Return` destroys. No pooling happens; this is what the integration uses until you assign something else.

`NetworkPrefabPool` recycles instead. A returned instance is deactivated and parked under a shared, inactive pool root, keyed by prefab identity. A later `Rent` for that identity reactivates the longest-parked instance instead of instantiating a new one; only when nothing is eligible does it fall back to instantiating. It exposes three members for watching and tuning that behavior:

- `InstantiatedCount` - how many instances this pool has had to instantiate because nothing was parked. This is the cold-miss count.
- `ReusedCount` - how many rents were satisfied by reactivating a parked instance instead.
- `ReturnCooldownSeconds` - how long a returned instance must sit parked before a rent may reuse it. Zero reuses immediately; a positive value keeps just-returned instances visible and settled for a beat before they come back.

Reactivating a parked instance does not re-run `Awake`. A script that needs to re-establish per-life state on reuse should do it through `NetworkSystemObject.SystemLinked` / `SystemUnlinked`, not `Awake`.

Installing a pool

Assign `NucleusUnity.PrefabPool`:

```
NucleusUnity.PrefabPool = new NetworkPrefabPool { ReturnCooldownSeconds = 1f };
```

Assigning `null` restores the default (`DefaultNetworkPrefabPool`). The assignment takes effect

whether the integration has been initialized yet or not, and applies for as long as it's set - a game that wants the default back later just assigns `null` again.

What survives a pooled life, and what doesn't

A GameObject recycled by `NetworkPrefabPool` is not a fresh object as far as `NetworkSystemObject` is concerned. Its `SystemLinked` and `SystemUnlinked` subscriptions, and any standing `NetworkSystemRequests` scripts on it made, are held across pooled lives and cleared only on true destruction (`OnDestroy`), never on a despawn that returns the object to the pool. That's deliberate: a recycled instance never runs a second `Awake`, so a co-located script that declared what it needs once keeps that declaration, and the next `Rent`'s link serves it again without re-declaring anything.

What does not survive a recycle is the `NetworkSystem` itself. The marker (`NetworkSystemObject`) is the stable GameObject-side identity across pooled lives; the `NetworkSystem` is what despawns and is pooled underneath, and it's also the RPC scope. When it's returned, its RPC handlers go with it - a script that registered a handler against that system needs to re-register against whatever system the next `Rent` links in, through `SystemLinked` rather than assuming the old registration still applies.

Reading the plateau

`InstantiatedCount` climbs only on a cold miss - a rent with nothing eligible parked - so it settles at your peak concurrent instance count and stops growing once the pool has been through that many outstanding at once. `ReusedCount` keeps climbing after that, since every further rent is satisfied from the parked queue instead.

`ReturnCooldownSeconds` changes where that peak lands, not whether reuse happens. A longer cooldown holds returned instances out of circulation for longer, so a spawn burst arriving inside that window finds fewer eligible parked instances and instantiates more - pushing `InstantiatedCount`'s plateau higher. Zero lets a returned instance be reused on the very next rent, which minimizes the plateau at the cost of a just-despawned instance reappearing immediately if you wanted a beat of visible downtime instead.

Two pools, one page

This page covers the GameObject pool - `INetworkPrefabPool`, the thing that decides whether a spawn gets a fresh `Instantiate` or a recycled instance. Underneath it, the `NetworkSystem` each GameObject links to is drawn from its own, separate pool on the engine side, keyed off the same despawn-return path described above. That pool has no Unity surface and no inspector assignment; see the API reference for `NetworkSystem` pooling for how it's tuned.

Carrying and re-parenting objects

Driving the core API directly? See Parents, children and hierarchy.

The call

One server-only call parents one networked object to another:

```
CoreManager.SystemManager.EnsureSetSystemParent(crateSystem, carrierSystem);
```

Pass `null` as the parent to detach the object and return it to the root of its scene:

```
CoreManager.SystemManager.EnsureSetSystemParent(crateSystem, parentNetworkSystem: null);
```

Both arguments are `NetworkSystem` references, not GameObjects. `EnsureSetSystemParent` returns `true` when the parent actually changed, `false` if the caller is not the server, either system is not started, or the pair fails one of the eligibility checks described below. Only the server may call it.

What re-parenting does on each peer

Nothing in your code moves the child's GameObject. As the replicated parent identifier lands on each peer, the Unity integration re-seats the linked GameObject under the new parent's transform (or, on detach, back under its scene root). This runs identically on the server and on every client, including the frame the server itself makes the call, so you never write `transform.SetParent` yourself. The server additionally refreshes the object's interest as part of the same call, since interest is only ever computed there.

Why carrying is free

Whether the ride costs anything to replicate depends on the transform's space, set per object on `NetworkTransform`'s `Transform Space` field:

- **Local** (the default): the object replicates the offset it holds inside its parent. Once parented, that offset stops changing while the ride continues, so a carried child costs nothing to send for as long as it rides — only the carrier's own transform is written per tick.
- **World**: the object replicates the pose it holds in the scene. A re-parent moves nothing and the child keeps paying full pose replication while it rides, regardless of what it's attached to.

A re-parent under Local space carries the child's *local* values through rather than holding its world pose still, so the child lands wherever its old world pose happens to read as once it's measured inside the new parent. If you want it to land at a specific spot — a deck slot, a socket — set its world position and rotation explicitly right after the call, the same way you'd place any other object:

```
if (!CoreManager.SystemManager.EnsureSetSystemParent(crateSystem, carrierSystem))
    return;

crate.transform.SetPositionAndRotation(carrierTransform.TransformPoint(slotPosition), carr
ierTransform.rotation);
```

The same applies in reverse on detach: read the world pose before calling `EnsureSetSystemParent` with a `null` parent, then write it back afterward, or the object will jump to wherever its local offset reads as from the scene root.

Reading the parent link from a script

`NetworkSystem` exposes the link for code that needs to react to it rather than drive it:

- `ParentId` — the server's `Id` for the current parent, or unset when there is none.
- `HasParent` — true whenever `ParentId` is set, whether or not the parent object has arrived on this peer yet.
- `Parent` — the resolved `NetworkSystem` reference, or `null` when unparented *or* when the parent hasn't resolved locally yet.
- `IsParentResolved` — true only once `Parent` is non-null.
- `ParentChanged` — an event of type `NetworkSystem.ParentChangedHandler(uint previousParentId, uint currentParentId)`, raised on the server as it makes the change and on each client as it applies the replicated one.

`ParentId` can be set before `Parent` resolves: the identifier arrives with the spawn or the re-parent entry, but the live reference is only wired up once every read for that tick has landed. Code that reacts to a parent change should read `ParentId`/`HasParent` rather than assume `Parent` is already populated in the same callback:

```
private void OnSystemLinked(NetworkSystem system)
{
    _crateSystem = system;
    _crateSystem.ParentChanged += OnParentChanged;
}

private void OnParentChanged(uint previousParentId, uint currentParentId)
{
    // React to _crateSystem.HasParent here; _crateSystem.Parent may not be resolved yet.
}
```

Limits to design around

- **Only a dynamically spawned prefab can be a child.** A scene object's placement is authored, and a pure-code system has no engine object to re-seat, so neither is eligible as a child. Either kind of system can still be a parent.
- **Hierarchies nest at most 8 deep**, counted from a root. This bound also caps the despawn and interest walks that follow the parent link.
- **Despawning a parent cascades to its children.** Stopping a system stops every started descendant beneath it, deepest first, so a peer is never left holding a child whose parent is already gone. Objects not parented to the one being despawned are untouched.

For the full model — the resolved-versus-arrived distinction, the acyclic and same-scene checks the server enforces, and how the child's Id, controller and access survive the move — see Parents, children and hierarchy.

Spawning and despawning objects

Driving the core API directly? See Starting and stopping systems.

Declaring what an object needs

`NetworkSystemObjectPool.RequireSystem<TSystem, TComponent0...>` is the form to reach for from a component sitting on a marker. It declares what the object needs once, and hands the system over the moment one exists: replayed immediately if the object already holds a matching system, rented on the spot if this peer creates it, or delivered through the marker's link event when the wire brings it instead.

```
NetworkSystemObjectPool.RequireSystem<NetworkSystem, ColorPulseComponent>(
    this,
    OnSystemAcquired,
    OnSystemReleased,
    canStartSystem: true);

private void OnSystemAcquired(NetworkSystem networkSystem)
{
    // Read/write the linked system's components here.
}

private void OnSystemReleased(NetworkSystem networkSystem)
{
    // Reset per-life state; the system has unlinked.
}
```

`OnSystemReleased` is optional — pass `null` when there is nothing to reset. Both handlers are typed as `NetworkSystemAcquiredHandler` and `NetworkSystemReleasedHandler`.

A direct call to `TryRent` asks too early on a transport that has not finished starting, and misses a system that already linked before the call ran. `RequireSystem` owns both: it retries while the transport comes up, and adopts a system that linked before the subscription existed. Reach for `TryRent` instead only when a script wants the system back synchronously, such as a driver creating an object's system at a moment of its own choosing.

Renting directly

`NetworkSystemObjectPool.TryRent<TSystem, TComponent0...>(contextComponent, out networkSystem, canStartSystem = true)` is the one-shot form. It returns `true` once a system is linked to the object, and `false` on an error, which it logs. One call is correct on every peer:

```
if (!NetworkSystemObjectPool.TryRent<NetworkSystem, UnityLocalTransformComponent>(this, ou
t NetworkSystem system))
    return;
```

The call resolves its `CoreManager` from `NucleusUnity`, so nothing is passed in. Internally it branches on the calling peer's situation:

- **A wire spawn is staging.** The integration is mid-`Instantiate` for an incoming spawn, and the rent hands back the system the wire already built rather than constructing a new one.
- **A client scene object.** The system is built locally, unstarted, and parked by scene identifier until the server's spawn binds it.
- **The server.** The system is rented and started with the marker's platform identity, which is what

replicates it to observers.

The marker, and renting more than once

The context component supplies the marker: the rent locates a `NetworkSystemObject` on the component itself, or walks up through its parents. A second rent against the same marker is supported — the systems auto-group, sharing one `GroupId` so a receiver hosts every member on a single GameObject instead of instantiating a second prefab:

```
// Both calls target `this`, so both systems land on the one marker and share a GroupId.
NetworkSystemObjectPool.TryRent<NetworkSystem, UnityLocalTransformComponent>(this, out _);
NetworkSystemObjectPool.TryRent<NetworkSystem, ColorPulseComponent>(this, out _);
```

What is refused is a rent whose composition does not match a staged wire spawn: on a receiving client, a rent inside a freshly instantiated prefab's `Awake` must ask for the same `TSystem` and the same NetworkComponent types the wire actually sent for that member, or the call logs an error and returns `false`. Up to 31 NetworkComponent type arguments are generated for both `TryRent` and `RequireSystem` (`TComponent0` through `TComponent30`), and both also have a form that takes none, for a system that carries only a spawn payload.

When the number of components comes from data rather than from your code, both also take a count: `TryRent<TSystem, TComponent0>(this, componentCount, out networkSystem)` and `RequireSystem<TSystem, TComponent0>(this, componentCount, onAcquired)` give one system carrying `TComponent0` that many times. A system satisfies the request only when it holds exactly that many components, all of that type, and each one is reached by its component Id with `NetworkSystem.TryGetComponent(componentId, out component)`. This is how `NetworkTransform` puts every part it lists on one system.

Despawning

`NetworkSystemObject.Despawn()` despawns the object on the server. It stops every system linked to the marker, which replicates the despawn to observers, and requests the systems return to their pool once the stop completes:

```
_networkSystemObject.Despawn();
```

A call with no linked system is a no-op. Only the server may despawn a replicated system — a client call is refused. On the receiving side, a dynamic prefab's GameObject returns to the prefab pool for reuse; a scene object's GameObject is destroyed, since it is a fixed instance rather than a poolable one.

Where the object lands

A spawn arrives as a full serialization that is deserialized before the prefab instantiates, so the framework reads the spawned system's pose before creating the GameObject rather than moving it into place afterward. If the system carries a `UnityTransformComponentBase`, its `ReplicatedPosition`, `ReplicatedRotation`, and `LocalScale` supply the instantiate pose. If the system carries no transform component, the prefab's own authored transform is used instead, unless the marker's **Auto Transform Payload** is on, in which case the object arrives at the server's parent and pose. Either way, the instance is produced already positioned — there is no frame where it sits at the origin waiting to be moved.

A system whose position is driven by a controller — a `ProjectedRigidbody` another peer simulates — is not placed by `Instantiate` at all. The controlling peer builds its own copy from the prefab and asserts the prefab's pose back within a tick, so a server-chosen spawn point set through `Instantiate` is overwritten almost immediately. Derive or publish the spawn point so the controlling peer can place itself.

A spawn can carry one-time data

Data an object needs only when it arrives, such as a random seed or a loot table, goes in a spawn payload rather than a member. Pick a payload type in the marker's **Spawn Payload** list, or declare one from code with `NetworkSystem.SetSpawnPayload`. Spawning systems explains how payloads work and who may declare them. On a client, a scene object is bound after it has already linked, so read a scene object's payload where it arrives, in the payload's own `ReadSpawnPayload`, or from `SystemManager.SystemStarted` or a system watch, which announce a scene object on a client once the server's spawn has bound it, rather than in `OnSystemLinked`.

An object that needs nothing but a payload can require a system with no components: `NetworkSystemObjectPool.RequireSystem<NetworkSystem>(this, OnSystemAcquired)`.

A scene object needs no separate spawn call

There is no separate `Spawn` call for an object already sitting in a loaded scene — the same `TryRent` or `RequireSystem` call that spawns a dynamic prefab also resolves a scene object, built locally and bound once the server's spawn arrives.

What a system is

The replicated unit

A `NetworkSystem` is the thing Nucleus replicates. It is not a data container itself; the data lives in the `NetworkComponent` instances attached to it. A system can hold up to 31 components (`NetworkSystem.MaximumComponentCount`), and you read them back with `TryGetComponent<T0>` or check how many are attached with `ComponentCount`.

```
if (system.TryGetComponent<TransformComponent>(out TransformComponent transform))
{
    // transform is one of the components attached to this system.
}

int attached = system.ComponentCount;
```

Think of the system as the identity and lifecycle; the components are the fields that get serialized onto the wire.

Each component also has a component Id: the order it was added in, starting at zero. Every peer builds the system's components in the same order, so an Id names the same component everywhere. `TryGetComponent<T0>(out T0)` returns the first component of a type, which is all you need while each type appears once. A system can carry one type several times, though (a `NetworkTransform` replicating several parts does exactly that), and then you reach each instance by its Id:

```
for (byte componentId = 0; componentId < system.ComponentCount; componentId++)
{
    if (system.TryGetComponent(componentId, out TransformComponent part))
    {
        // part is the component added at position componentId, the same one on every peer
.
    }
}
```

Lifecycle

Every system moves through the same four states, exposed as `NetworkSystemState`:

- `Starting` — accepted but not yet live.
- `Started` — networked and usable.
- `Stopping` — a stop has been accepted but the tick has not ended yet.
- `Stopped` — not networked.

Read the current state through `NetworkSystem.State`, or use the shortcut `IsStarted`, which is true only while `State` is `Started`.

```
if (system.IsStarted)
{
    // safe to read replicated component data here.
}
```

A start accepted after the tick's state flush stays at `Starting` until the next tick begins; during that window `Id` is still unset.

Identifiers

A system carries several identifiers, and each answers a different question:

- **`Id`** — the network identity: which system this is, unique across the network. `UnsetId` is 0, and a system reads this as unset until it actually starts.
- **`PlatformId`** — which prefab, or which scene object, this system is linked to on the running platform. `UnsetPlatformId` is 0, which is also what a system with no engine object carries — a plain system never sets this.
- **`PrefabBundleId`** — the content bundle that owns the prefab named by `PlatformId`. Only meaningful for a dynamically spawned prefab; a scene object doesn't use it.
- **`GroupId`** — the `NetworkSystemGroup` this system belongs to, or unset if it isn't grouped.
- **`ParentId`** — the `Id` of the system this one is parented to (a crate riding a vehicle, a weapon in a hand), or `UnsetParentId` if it sits at the root of its scene.
- **`SceneHandle`** — the live scene instance the system belongs to, or `UnsetSceneHandle` if it belongs to whatever scene the peer booted into rather than one the server opened.
- **`IsSceneObject`** — true when `PlatformId` identifies a scene object that already exists on every peer, rather than a prefab a receiver must instantiate.

Role reads

Every system carries three reads for what role the local peer is playing: `IsServerStarted`, `IsClientStarted`, and `IsHostStarted`. These answer "is the server/client/host running here", not "do I control this object" — for control questions, use `IsController(ControllerType)` instead.

Systems and engine objects

A `NetworkSystem` does not need an engine object at all — a purely logical system (matchmaking state, a game rule tracker) can start, replicate, and stop without ever touching a GameObject or any other engine construct.

In an engine integration, one engine object can link to several systems at once, each with its own `Id` and lifecycle, all driven by the same underlying object.

Vocabulary

A `NetworkSystem` is a system, not a "networked GameObject." In the Unity integration, the marker component that links an engine object to its system is a separate thing — a small bridge, not the system itself.

Blitz Relay Transport component

Driving the core API directly? See The relay transport.

`BlitzRelayTransport` is a `NetworkTransport` component that carries a session through a relay, so two peers behind routers or NAT can reach each other without either one listening on a port. Add it to the `UnityCoreManager`'s GameObject; the `UnityTransportManager` adds it to the `CoreManager` with the inspector's settings applied.

The component and its custom editor compile only behind the `BLITZ_RELAY` scripting define. A project that has not added the define never loads anything relay-related.

Blitz Relay needs Pro. The relay assembly references the Newfarm Migration assembly, whose host-migration coordinator calls `ServerManager.TryAdoptRetainedWorld`, and that method exists only in a Pro build. Against a free `Nucleus.dll` the relay integration does not compile, whether or not Host Migration is on.

What must already be running

This component does not start a relay. It dials one. Something else has to be running the Blitz Relay service at the address and port configured below, and, if host migration is on, a directory service too. See Hosting and joining a relay session for standing those up.

Relay

Field	Default	Purpose
Address	`127.0.0.1`	Address of the Blitz Relay carrying this session.
Port	`7770` (`RelayTransport.DefaultRelayPort`)	Port the relay listens on.
Connection Key	empty	Key the relay admits peers with. Set by whoever runs the relay, never by a player.

Room

Field	Default	Purpose
Room Code	empty	Room to join. Leave empty to host, or when joining by session id instead.
Maximum Clients	`16` (`RelayTransport.DefaultMaximumClients`)	How many clients a room created by this peer will hold. The relay refuses a room without a real number, so this cannot mean unlimited.
Transmission Unit	`1200`	Largest datagram to build before it is split.

The relay reports a larger `DatagramOverheadBytes` than the Synapse transport does: on top of the usual

per-packet overhead, every datagram also carries the relay's own framing so it can be routed to the right room. That overhead is deducted automatically when Nucleus reserves header space, so it does not require any adjustment here.

Surviving a lost host

Field	Default	Purpose
Host Migration	`true`	Register the session with a directory so it survives losing whoever is hosting it.
Directory Address	`127.0.0.1`	Address of the directory the session registers with. Shown only while Host Migration is on.
Directory Port	`47778`	Port the directory listens on. Shown only while Host Migration is on.

Turning Host Migration on sets `ClientManager.DisconnectResetMode` to `RetainReceivedWorld`, so a client keeps its received world across the moment its link drops instead of being reset to an empty one. That wiring, and the creation of the migration coordinator, lives in a partial of its own (`BlitzRelayTransport.Migration.Pro.cs`).

This is the relay-and-directory handover this component ships: a client can keep its world across a host loss, and a peer can rejoin a moved session by session id instead of a room code. It is not a separate, broader engine-level host migration feature — nothing beyond what is described on this page and on Hosting and joining a relay session exists yet.

Live fields

Two read-only fields appear while the game is playing: **Current Room**, the room this peer is hosting or has joined, and **Session Id**, shown in hex once a session has registered with a directory. Both are blank until play starts, and Session Id stays blank unless Host Migration is on and a session has actually opened.

Connecting across networks

The three hops

Get a session working in this order. Each hop changes exactly one thing, so when it breaks you know which layer to blame.

Loopback, one process. Both peers run in the same process on `127.0.0.1`. This is the default: `RemoteHost` starts at `IPAddress.Loopback` in C#, and the Unity `SynapseTransport` component defaults its Remote Host field to `127.0.0.1`. Nothing about ports or addresses can be wrong here, so if this fails the bug is in your game logic, not the network.

LAN, two machines. Point the client's Remote Host at the server machine's LAN IP (e.g. `192.168.x.x`) instead of loopback. The server's own configuration does not change. This hop tests that the port is actually open on the server machine and that nothing between the two machines (a firewall) is dropping the traffic.

Open internet, peer outside your network. The client needs the server's public IP or DNS name. This hop adds a router in the path, which introduces port forwarding as a new failure point on top of the previous two.

What each side needs

The client needs the server's reachable address and the port it is listening on. The server needs that port open to it.

- Unity: set **Remote Host** on the `SynapseTransport` component to the server's address, and **Port** to match on both sides.
- C#: set `Synapse.RemoteHost` (an `IPAddress`) before connecting the client, and `Configuration.Port` the same on both peers.

See the Unity and C# transport pages for the full field lists.

Why the server ignores the address

`Synapse`'s server side always binds every local address on the machine; it never looks at `RemoteHost`, and it reads only `Configuration.Port`. That is why the server-side address field doesn't exist at all — there's nothing to set. A server listening on port 7777 answers on its loopback address, its LAN address, and its public address simultaneously, all on that same port. What has to be right is which of those addresses the client is told to dial, and whether traffic to that address and port actually reaches the machine.

The failure ladder

Work through these in order; each one only matters once the ones above it are ruled out.

Wrong port. The client's port doesn't match the server's `Configuration.Port`. Cheapest thing to check, check it first.

Firewall. The OS or network firewall on the server machine is dropping inbound packets to that port before the socket ever sees them.

Router with no forward. On a home/office network, the router won't route an inbound packet from the internet to the server machine's LAN address unless a port forward (or UPnP) sends it there. LAN-to-LAN traffic never hits this; it only shows up on the third hop.

Carrier-grade NAT. Some ISPs (and most mobile carriers) put the server behind a NAT you don't

control and can't forward through, no matter how correctly the router is configured. If forwarding is set up correctly and still doesn't work, this is usually why — and no amount of retrying fixes it from your end.

Reading which layer failed

`LocalConnectionState` alone can't tell you where it stopped. A `SynapseTransport` client reports `Connected` as soon as it has sent its handshake, before the server has answered, and a link that fails for any reason ends at `Disconnected`; the shipped transports never set `Error` or `TimedOut`. The sign that the server actually answered and accepted the client is `LocalClientAuthenticated` on `UnityClientManager.NucleusClientManager`, so read the two together:

- **Drops to `Disconnected` without `LocalClientAuthenticated` ever firing.** The client's packets aren't reaching the server at all. The client waits out the `SynapseTransport` component's **Idle Timeout Seconds** (15 by default) and then drops. This is a routing/reachability problem: wrong address, wrong port, firewall, or missing port forward. A server that refuses the client drops it straight away instead, and `ClientManager.LastAuthenticationDenialReason` holds the reason it gave.
- **`LocalClientAuthenticated` fired, then the link dropped to `Disconnected`.** The initial handshake worked, so the path was open at least once, but traffic stopped flowing after that. This points at something intermittent: a NAT mapping that expired, a firewall rule that only blocks certain packet patterns, or a real network interruption. It's a different problem than never connecting at all, so don't go back to checking the port or the forward.

When to stop fighting NAT

If you've confirmed the port is right, the firewall is open, and the router has a working forward, and it still won't connect over the open internet, you're likely behind (or connecting to a peer behind) carrier-grade NAT or a similarly locked-down network you have no access to. At that point, stop trying to forward a port that can't be forwarded and use `RelayTransport` instead: one peer hosts a room and gets back a `RoomCode`, and every other peer joins with just that code — no address, no port, no forwarding required on either side.

No discovery, no punchthrough

Nucleus does not scan the LAN for servers and does not attempt NAT punchthrough on its own. An address always has to come from somewhere you provide — typed in, read from a config file, fetched from your own matchmaking service, or (for the relay path) a room code exchanged out of band.

Building a headless Unity server

Driving the core API directly? See Running a dedicated .NET server.

The scene

Build with Unity's Dedicated Server target. The scene that loads on launch needs a `UnityCoreManager` and, on the same GameObject hierarchy, a `UnityTransportManager` with its Automatic Start Mode set to Server.

`UnityCoreManager.Awake` runs before any other script (it carries `[DefaultExecutionOrder(-10000)]`), builds the `CoreManager`, and adds every missing Unity manager component - `UnityTransportManager` among them - so a scene with only a `UnityCoreManager` still ends up with one.

Reading launch arguments yourself

Nothing in the Unity integration calls `Transport.ApplyCommandLineArguments` for you. `UnityTransportManager` adds each `NetworkTransport` component it finds to the core manager and can start them, but it never reads the process's launch arguments on your behalf. You need your own bootstrap script that calls `ApplyCommandLineArguments()` on each transport, and it has to run before that transport connects - the call only has anything to change if it runs first.

```
using Nucleus.Transports;

public class ServerBootstrap : MonoBehaviour
{
    private async void Start()
    {
        UnityTransportManager transportManager = GetComponent<UnityTransportManager>();

        await transportManager.EnsureAddedAsync();

        foreach (Transport transport in transportManager.Transports)
            transport.ApplyCommandLineArguments();

        await transportManager.StartServerAsync();
    }
}
```

`ApplyCommandLineArguments()` with no arguments reads `CommandLineArguments.Process`, which parses `Environment.GetCommandLineArgs()` once and caches the result for the process. The port flag is `-port <value>`; any flag your transport recognizes works in both `-flag value` and `-flag=value` form.

Why Automatic Start Mode alone isn't enough

`UnityTransportManager` starts itself from its own `Start()`, and Unity gives no ordering guarantee between two components' `Start()` calls on the same frame. Set Automatic Start Mode to Server and add your own argument-applying script beside it, and you have a race: whichever `Start()` runs first decides whether the server came up with your arguments applied or not.

Leave Automatic Start Mode at None and drive the start yourself once arguments are applied, as in the bootstrap above - call `StartServerAsync()` after your loop of `ApplyCommandLineArguments()` calls, not before. `StartServerAsync` calls `EnsureAddedAsync` itself, so the explicit call in the example above

is only there to get at `Transports` before the transports connect.

What a headless build has and doesn't have

There's no `NetworkDiagnosticsHud` in a build made this way - it's demo code that lives under the integration's `Demos` folder and isn't meant to ship. A headless server has no window to read it from regardless. Route Nucleus's own logging somewhere you can actually read: the Unity integration's logger writes through `UnityEngine.Debug`, whose output still lands in the standalone player's log file even with no console attached, so check that file rather than expecting a console window.

Shutting down cleanly

Call `UnityTransportManager.ShutdownAll()` on process exit to remove and shut down every transport the manager added. It already runs from the component's own `OnDestroy`, so a normal scene teardown covers it; call it directly only if your bootstrap needs to shut the transports down before that, for example on a signal handler that also tears down other state.

When to skip Unity entirely

If the server doesn't need Unity's engine loop, physics, or any other engine-side feature, a plain .NET host avoids the editor, the Dedicated Server build pipeline, and this bootstrap ordering altogether. See Running a dedicated .NET server.

Choosing a hosting topology

The three shapes

Nucleus does not pick a topology for you. A `CoreManager` can run its server half, its client half, or both, and that one choice is what separates a dedicated server, a player-hosted session, and a relayed room.

Dedicated server. You run a process with only the server half started. It has no gameplay of its own; every player connects as a client. You pay for the machine, and it keeps running whether or not any particular player is online.

Host. One player's process starts both halves. That player is also a client of their own server. Nobody else pays for a machine or keeps a process alive on your behalf; the session lives exactly as long as that player's process does.

Relayed room. A host, plus a `RelayTransport` in front of it. Neither the host nor any joining peer listens for inbound connections. Both sides are clients of the relay server, and the relay hands them to each other under a room code.

What each costs

Shape	Who runs a machine	Reachability	Session lifetime
Dedicated server	You	Server needs an open, reachable port	Independent of any one player
Host	The hosting player	Both server and client halves need an open, reachable port	Dies when the host's process does
Relayed room	The relay operator	Neither peer needs to be reachable	Dies when the host's process does, unless paired with a directory handover

A host is not a special mode

There is no host-only code path in the engine. A host is a `CoreManager` with a server started and a client started in the same process, talking to each other. `HostPairing` describes exactly this: two independent sockets, one process, each computed from the other's address. Under `HostPairing.Endpoint` the two halves exchange real datagrams over loopback like any other server/client pair; under `HostPairing.Local` they're paired in process without ever putting a packet on the wire. Either way, it's the same server and the same client code every other topology runs.

That has one direct consequence for trust: the host's `CoreManager` is the server. It runs the same server logic a dedicated server would, with the same say over what's true. Hosting doesn't relax that — it just means the server is sitting in a player's process instead of yours.

Every topology has its own reachability problem

A dedicated server needs an open, forwarded port people can reach. A player host needs the same thing on the player's own connection — behind carrier-grade NAT or a restrictive firewall, that often doesn't work.

A relayed room needs neither. `RelayTransport` carries the session through a Blitz Relay server: "nothing here listens. Both sides are clients of the relay, one holding the host role in a room and the rest joining it."

The host connects out to the relay, the relay names a room, and every joining peer connects out to the same relay and gives it the room code. No peer accepts inbound connections, so there's no port to forward and no NAT to fight.

The relay doesn't change who's in charge

Relaying usually gets blamed for handing control to whoever the relay trusts, or for adding a man-in-the-middle that could tamper with state. That doesn't happen here, because the relay only replaces the transport's addressing — it's still one peer's `CoreManager` acting as the server. The host authenticates and simulates exactly as it would over a direct connection; the relay just forwards its packets under a room code instead of a socket address. Nothing on the wire becomes client-authoritative by being relayed, and no other peer gains any capability it wouldn't have had directly against that same host.

Session lifetime

A plain host's session lives and dies with that player's process. Close the game, and the server half — and everyone's connection to it — goes with it.

What this repo ships for surviving that is a relay-plus-directory pairing: `ISessionHost` is the seam a directory implementation uses to publish where a session is hosted (`HostedCredential`) and to join wherever the directory currently points (`JoinAsync`). `NewfarmHostMigration` drives that against the directory on a poll loop, so a session can hand its hosting role from one peer to another without every client needing to be told a new address by hand. World-state retention across that handover (keeping the game state alive rather than starting the new host cold) is a Pro feature, and so is the pairing itself, because `NewfarmHostMigration` adopts the retained world when it promotes a peer. A broader, engine-level host migration feature is a separate, larger undertaking documented elsewhere and not yet built; this pairing is not that.

Trade-off axes

Weigh a topology on these, not on habit:

- **Cost.** A dedicated server is a machine you run and pay for continuously. A host costs you nothing directly — the player's machine and connection carry it. A relay costs whoever runs the relay server, continuously, whether or not it's you.
- **Trust.** All three have exactly one `CoreManager`, and it is the server. What differs is whose process it runs in, and whether you trust that process's operator (yourself, or a player) to run honest server logic.
- **Latency.** A dedicated server sits somewhere fixed, generally central. A host's latency is whatever every other peer's connection to that one player looks like — good for the host, variable for everyone else. A relay adds the extra hop through the relay server on top of the host's own connection.
- **Persistence.** A dedicated server's world persists independent of players. A plain host's world dies with the host's process. A relayed room with the directory pairing can hand the session to a new host, and Pro can carry the world state across that handover.
- **Who keeps a machine running.** A dedicated server asks you to. A host asks a player to, for as long as the session needs to exist. A relay asks whoever operates the relay to keep a lightweight forwarding service up, plus (for a survivable session) a directory.

What Nucleus does not provide

Nucleus does not run matchmaking, does not run a lobby service, and does not operate a hosted relay for you. `RelayTransport` and the directory seam (`ISessionHost`) are building blocks you point at infrastructure you run or a service you choose — the engine doesn't supply that infrastructure itself.

All three transports Nucleus ships — Synapse, Yak, and the Blitz Relay integration — are UDP. Pick a topology for its reachability and trust trade-offs, not for a transport-level constraint; none of the three shapes changes which transport you're allowed to use.

How transports work

What a transport does

A `Transport` moves bytes between two peers, and nothing else. It connects, disconnects, sends and receives datagrams, and tells the engine whether a given connection is its own client talking to its own server (`IsOwnClientConnection`). It does not know about identity, interest, reliability semantics, acknowledgment, or recovery — those all live in the engine above it, in `Connection` and the managers that use it.

That division is what lets you swap a transport without touching anything else. A game built against Synapse, Yak, or Blitz Relay replicates, spawns, and reconciles the same way regardless of which one is moving the bytes, because none of that logic ever reaches down into the transport.

Channels

Every send picks a `Channel`:

- `Channel.Reliable` — ordered and reliable.
- `Channel.Unreliable` — unreliable.

Nucleus is unreliable-first: state, RPCs, and messages default to `Channel.Unreliable` and only move to `Channel.Reliable` when something requires it.

What happens past the transmission unit differs by what is being sent:

- A reliable message or remote call that will not fit is handed to the transport whole, and the transport splits it.
- An unreliable message or remote call that will not fit is split by the engine into pieces that each fit one datagram, so it stays unreliable on every transport, the relay included. It arrives whole or not at all.
- A state write that will not fit is sent across several packets on whichever channel it was already using.

Maximum Transmission Unit

`Configuration.MaximumTransmissionUnit` is 1200 bytes by default — the ceiling on what the engine packs into one packet before it fragments or promotes.

The engine never gets the full 1200 bytes of payload to work with. Each transport reports `DatagramOverheadBytes`, the header it prepends to every datagram, and the engine reserves that (plus its own packet header) out of the configured unit before writing anything. So the number you set is a limit on the datagram, not a guarantee of payload space, and how much space you actually lose depends on the transport:

- Synapse reports its socket library's own header size as overhead.
- Blitz Relay reports more: the relay's own framing on top of the same underlying socket header, because the relay is a hop the packet passes through, not the endpoint reading it. A relayed session always has less usable payload per packet than a direct Synapse connection at the same MTU.

The three shipped transports

Transport	What it is	When to use it

Synapse	Real UDP sockets	Direct connections, dedicated servers, anywhere both peers (or the server) have a reachable address and port.
Yak	Offline, in-process, nothing bound	Local single-player or same-process testing where no socket should exist at all.
Blitz Relay	A third machine carries the traffic	Peers that can't reach each other directly — no port forwarding, both behind NAT — at the cost of relay overhead and a dependency on that third machine.

Adding transports

A `CoreManager` can hold several transports at once — add each with `TransportManager.AddTransportAsync` or `TryAddTransportAsync`. A transport is added and initialized before it is connected; `ConnectAsync` is a separate, later call.

Where to go next

Driving this from the Unity integration: see the transport component pages. Driving it from plain C#: see the API pages for `Transport`, `Configuration`, and `TransportManager`.

Transports, Connectivity and Hosting

Pages in this category:

- Synapse Transport component
- Transport Manager component
- How transports work
- Blitz Relay Transport component
- Choosing a hosting topology
- Building a headless Unity server
- Connecting across networks
- Simulating a bad connection in the editor
- Hosting and joining a relay session

Hosting and joining a relay session

Driving the core API directly? See The relay transport.

Hosting a session

Add `BlitzRelayTransport` to the `UnityCoreManager`'s GameObject and call `StartHostingSessionAsync()`. It opens a session, hosts it, and registers the room the relay names.

```
bool isHosting = await blitzRelayTransport.StartHostingSessionAsync();
```

Once that returns `true`, read `RoomCode` and hand it to the players who will join directly:

```
string roomCode = blitzRelayTransport.RoomCode;
```

`RoomCode` reads through to `RelayTransport.HostedRoomCode` on the underlying core transport once one exists, and falls back to the inspector-configured room otherwise. Either way, it is a room, not a session: see below for why that distinction matters past the first host.

Joining a session

A joining peer can go one of two ways.

By room code. Set `RoomCode` before connecting, the same field a host leaves empty. This puts the joining peer straight into that room and nothing else.

By session id. Call `JoinSessionAsync(sessionId)`. It finds the session by its identifier and joins whatever room it currently lives in, wherever that is.

```
bool isJoined = await blitzRelayTransport.JoinSessionAsync(sessionId);
```

`SessionId` gives you the id to distribute in the first place:

```
ulong sessionId = blitzRelayTransport.SessionId;
```

It reads `SessionDirectoryUnset` (`0`) when this peer holds no session, which is the case whenever `IsHostMigrationEnabled` is off or hosting hasn't been started yet.

Room code vs. session id

The room code names one room on the relay. Every handover moves the session to a new room, so a room code a player was handed goes stale the moment the host changes. The session id is a bearer token registered with the newfarm directory: it survives a handover because the directory, not the relay, is what a rejoining peer asks.

Give players a room code for a session that will never change host. Give them a session id for anything that might.

Handing the room on

A host that means to leave without ending the match calls `TrySurrenderHostingSession()`:

```
bool isSurrendered = blitzRelayTransport.TrySurrenderHostingSession();
```

This gives the session up to whoever the directory elects next, without leaving it and without stopping play. The component owns the coordinator that does this work, exposed as `Migration` (a `NewfarmHostMigration`). It drives the directory from the component's own `Update`, so nothing beyond starting a hosting session or joining one is required of the game; election, adoption and finding the session's

new location all follow from that.

A surviving peer that is elected next adopts the world it had been receiving as its own and starts hosting it. A peer that only rejoins picks up the new host's room the same way it joined originally, through the session id.

What the host-migration toggle does

`IsHostMigrationEnabled` (on by default) is what makes `Migration` exist at all; without it, `StartHostingSessionAsync`, `JoinSessionAsync` and `TrySurrenderHostingSession` all fail and log an error telling you to drive the CoreManager directly instead.

Turning it on also sets `ClientManager.DisconnectResetMode` to `RetainReceivedWorld` for you. That's a client-side setting: it decides, at the moment this peer's link to the server drops, whether the world it received is thrown away or kept. Migration needs it kept, because the peer that gets elected next has to adopt what it was already receiving rather than starting from nothing.

`RetainReceivedWorld` is settable in a free build, but adopting the retained world is Pro-only, and so is this component: the coordinator adopts the retained world when it promotes a peer, so the Blitz Relay and Newfarm Migration assemblies do not compile against a free `Nucleus.dll`.

Watch `SystemManager.RespawnWorldOnReconnectEnabled` too. It changes only what happens when a client reconnects: the world is still kept when the link drops, and a peer elected to host still adopts it, but a peer that rejoins drops what it kept and rebuilds it from the new server's own account instead of being relinked to it. That costs a spawn per object, in exchange for never keeping anything the new server never heard of.

Services you have to run yourself

Neither ships in this repository:

- **The relay**, listening on UDP 7770, which needs a connection key (`BlitzRelayTransport`'s `_connectionKey` field/inspector setting) matching whatever the relay was configured with.
- **The newfarm directory**, listening on UDP 47778, which is what session ids and elections are resolved against.

Both addresses and ports are inspector fields on `BlitzRelayTransport` (`_relayAddress`/`_relayPort` and `_directoryAddress`/`_directoryPort`), defaulting to `127.0.0.1` and the ports above.

Simulating a bad connection in the editor

> **Driving the core API directly?** See Simulating latency, jitter and loss.

Playing in the editor runs over loopback, which is instant and lossless. You can make it behave like a real connection by giving the transports a set of network conditions.

Describe the connection with NetworkConditions

`NetworkConditions` is a struct in `Nucleus.Transports`. Each field describes one way a real connection goes wrong, and the default value simulates nothing.

- `LatencyMilliseconds` is a one-way delay added to every packet sent.
- `JitterMilliseconds` is extra delay on top of the latency, a fresh random amount below this figure for each packet, so packets do not all arrive shifted by the same amount.
- `PacketLossChance` is the chance, from 0 to 1, that a packet is dropped. `0.2f` drops one packet in five.
- `OutOfOrderChance` is the chance, from 0 to 1, that a packet is held back long enough to arrive after packets sent later.
- `DuplicateChance` is the chance, from 0 to 1, that a packet is also sent a second time.
- `Seed` fixes the random choices behind the other fields, so a test can be run again with the same choices for the same packets. `0`, the default, leaves them different on every run. The same seed repeats the choices, but a networked run can still differ, because how many packets are sent and when depends on timing. To replay a failing run, pick a seed yourself, log it, and set it.

The Synapse transport supports every field except `DuplicateChance`, which it ignores. A transport that supports none of them logs a warning when you set them.

Synapse sends its connection setup packets, the handshake and the NAT traversal probes, without any simulated delay or loss, so a connection always opens however bad the conditions are. A connect that succeeds under simulated conditions does not show that the handshake copes with that link.

Set them in the inspector

`UnityTransportManager` has a **Network Conditions** section in the inspector. Tick **Network Conditions Enabled** and fill in the values. When the transports are added, just before they start, it sets those conditions on every transport, or clears them when the box is unticked. This works with any **Automatic Start Mode**.

Each editor reads its own inspector, so in a ParrelSync pair you can make only the clone's connection bad.

Set them from code

Call `SetNetworkConditions` on `UnityTransportManager` to set every transport, or `SetNetworkConditions<Synapse>` to set only the Synapse ones. Passing null goes back to a clean connection. Conditions set from code replace the inspector's.

The transports have to exist before they can take the conditions, and Synapse decides whether to simulate at all when it connects. So set **Automatic Start Mode** on `UnityTransportManager` to `None`, and start the network yourself: add the transports, set the conditions, then start.

```
using Nucleus.Integrations.Unity.Managers.Transports;
using Nucleus.Transports;
using UnityEngine;

public class ConnectionSimulator : MonoBehaviour
{
    [SerializeField]
    private UnityTransportManager _unityTransportManager;

    private async void Start()
    {
        await _unityTransportManager.EnsureAddedAsync();

        _unityTransportManager.SetNetworkConditions(new NetworkConditions
        {
            LatencyMilliseconds = 100,
            JitterMilliseconds = 40,
            PacketLossChance = 0.05f,
        });

        await _unityTransportManager.StartHostAsync();
    }
}
```

Use `StartServerAsync` or `StartClientAsync` in place of `StartHostAsync` for the other roles.

The conditions can change while playing

A Synapse transport that connected with conditions takes new values straight away, so you can raise or lower them mid-session. One that connected with no conditions ignores them until it next connects. If you plan to change them while playing, start with a small condition, such as a latency of 1 ms, so the simulation is running from the start.

To see what is set, call `GetNetworkConditions<Synapse>()`, or `GetNetworkConditions()` for a pooled list with one entry per transport that you return to `ListPool` when done.

Each editor sets its own

Conditions belong to the transports in one editor. In a ParrelSync pair, each editor is its own process with its own transports, so setting conditions in the host editor does nothing to the clone.

Each peer applies loss to the packets it sends. A request and its reply each roll independently, so a round trip only survives when both directions do.

Read the effect back from the Connection

While playing, read the measured effect off `Connection`:

- `Connection.RoundTripTimeMilliseconds` is the smoothed round trip time.
- `Connection.RoundTripTimeDeviationMilliseconds` is the jitter, meaning how far samples stray from the smoothed average.
- `Connection.PacketLossPercentage` is the share of recent probes that went unanswered.

These are measured by the engine's own round trip probes, so they are the right numbers to check that the conditions are actually taking effect, not just an echo of the values you set.

Test at more than one setting

A steady 100 ms latency and a link swinging between 20 ms and 180 ms average the same but feel nothing alike, so set jitter as well as latency to catch bugs that only show up when timing varies. Try loss in the 1 to 5% range for typical broadband, and higher, around 10 to 20%, to stress recovery.

Some real network faults cannot be reproduced

The simulation delays, drops and reorders packets inside the process. It does not model a slow upload beside a fast download, or a real path that silently drops packets over some size. Loopback with conditions set tells you how your game feels on a poor connection, but it is not a substitute for testing over an actual network.

Simulating a bad connection in the editor

> **Driving the core API directly?** See Simulating latency, jitter and loss.

Playing in the editor runs over loopback, which is instant and lossless. You can make it behave like a real connection by giving the transports a set of network conditions.

Describe the connection with NetworkConditions

`NetworkConditions` is a struct in `Nucleus.Transports`. Each field describes one way a real connection goes wrong, and the default value simulates nothing.

- `LatencyMilliseconds` is a one-way delay added to every packet sent.
- `JitterMilliseconds` is extra delay on top of the latency, a fresh random amount below this figure for each packet, so packets do not all arrive shifted by the same amount.
- `PacketLossChance` is the chance, from 0 to 1, that a packet is dropped. `0.2f` drops one packet in five.
- `OutOfOrderChance` is the chance, from 0 to 1, that a packet is held back long enough to arrive after packets sent later.
- `DuplicateChance` is the chance, from 0 to 1, that a packet is also sent a second time.
- `Seed` fixes the random choices behind the other fields, so a test can be run again with the same choices for the same packets. `0`, the default, leaves them different on every run. The same seed repeats the choices, but a networked run can still differ, because how many packets are sent and when depends on timing. To replay a failing run, pick a seed yourself, log it, and set it.

The Synapse transport supports every field except `DuplicateChance`, which it ignores. A transport that supports none of them logs a warning when you set them.

Synapse sends its connection setup packets, the handshake and the NAT traversal probes, without any simulated delay or loss, so a connection always opens however bad the conditions are. A connect that succeeds under simulated conditions does not show that the handshake copes with that link.

Set them in the inspector

`UnityTransportManager` has a **Network Conditions** section in the inspector. Tick **Network Conditions Enabled** and fill in the values. When the transports are added, just before they start, it sets those conditions on every transport, or clears them when the box is unticked. This works with any **Automatic Start Mode**.

Each editor reads its own inspector, so in a ParrelSync pair you can make only the clone's connection bad.

Set them from code

Call `SetNetworkConditions` on `UnityTransportManager` to set every transport, or `SetNetworkConditions<Synapse>` to set only the Synapse ones. Passing null goes back to a clean connection. Conditions set from code replace the inspector's.

The transports have to exist before they can take the conditions, and Synapse decides whether to simulate at all when it connects. So set **Automatic Start Mode** on `UnityTransportManager` to `None`, and start the network yourself: add the transports, set the conditions, then start.

```
using Nucleus.Integrations.Unity.Managers.Transports;
using Nucleus.Transports;
using UnityEngine;

public class ConnectionSimulator : MonoBehaviour
{
    [SerializeField]
    private UnityTransportManager _unityTransportManager;

    private async void Start()
    {
        await _unityTransportManager.EnsureAddedAsync();

        _unityTransportManager.SetNetworkConditions(new NetworkConditions
        {
            LatencyMilliseconds = 100,
            JitterMilliseconds = 40,
            PacketLossChance = 0.05f,
        });

        await _unityTransportManager.StartHostAsync();
    }
}
```

Use `StartServerAsync` or `StartClientAsync` in place of `StartHostAsync` for the other roles.

The conditions can change while playing

A Synapse transport that connected with conditions takes new values straight away, so you can raise or lower them mid-session. One that connected with no conditions ignores them until it next connects. If you plan to change them while playing, start with a small condition, such as a latency of 1 ms, so the simulation is running from the start.

To see what is set, call `GetNetworkConditions<Synapse>()`, or `GetNetworkConditions()` for a pooled list with one entry per transport that you return to `ListPool` when done.

Each editor sets its own

Conditions belong to the transports in one editor. In a ParrelSync pair, each editor is its own process with its own transports, so setting conditions in the host editor does nothing to the clone.

Each peer applies loss to the packets it sends. A request and its reply each roll independently, so a round trip only survives when both directions do.

Read the effect back from the Connection

While playing, read the measured effect off `Connection`:

- `Connection.RoundTripTimeMilliseconds` is the smoothed round trip time.
- `Connection.RoundTripTimeDeviationMilliseconds` is the jitter, meaning how far samples stray from the smoothed average.
- `Connection.PacketLossPercentage` is the share of recent probes that went unanswered.

These are measured by the engine's own round trip probes, so they are the right numbers to check that the conditions are actually taking effect, not just an echo of the values you set.

Test at more than one setting

A steady 100 ms latency and a link swinging between 20 ms and 180 ms average the same but feel nothing alike, so set jitter as well as latency to catch bugs that only show up when timing varies. Try loss in the 1 to 5% range for typical broadband, and higher, around 10 to 20%, to stress recovery.

Some real network faults cannot be reproduced

The simulation delays, drops and reorders packets inside the process. It does not model a slow upload beside a fast download, or a real path that silently drops packets over some size. Loopback with conditions set tells you how your game feels on a poor connection, but it is not a substitute for testing over an actual network.

Synapse Transport component

Driving the core API directly? See The Synapse transport.

`SynapseTransport` is the Inspector-configurable `NetworkTransport` for the built-in UDP transport, Synapse. Place it on the `UnityCoreManager`'s GameObject and edit its fields there; `UnityTransportManager` reads every `NetworkTransport` component on that GameObject and its children and adds each to the `CoreManager`. When it finds none, it adds a `SynapseTransport` itself, so a project with no transport component configured still gets Synapse with its defaults.

A custom editor (`SynapseTransportEditor`) groups the fields into three labeled sections in the Inspector: Connection, Timeouts, and Hosting.

Inspector fields

Connection

Port (`ushort`, default `7777`): the port the server listens on and the client connects to.

Remote Host (`string`, default `"127.0.0.1"`): the address a client connects to. The server ignores it: a host's server half never reads `RemoteHost`, since a server just listens on `Port` rather than dialing anywhere.

Maximum Transmission Unit (`ushort`, default `1200`): the largest packet built before it is split into multiple packets.

Timeouts

Idle Timeout Seconds (`ushort`, default `15`): how long a connected peer waits without hearing from the other side before it drops the link. One second is the least value applied.

Connecting Timeout Seconds (`ushort`, default `10`): how long a client waits for the server to answer its connect before giving up and returning to `Disconnected`. Zero falls back to **Idle Timeout Seconds**.

Hosting

Host Pairing (`HostPairing`, default `HostPairing.Local` on this component): how this transport reaches its own client half when the peer running it is also the host.

- `HostPairing.Local`: the server and client halves are paired in process and never exchange a datagram. The link is named the instant it's made, and a host's own traffic never crosses the socket to be discarded on arrival. The cost is that a host no longer proves anything about the transport through its own half — a genuinely remote client of the same host is unaffected and still goes over the real socket.
- `HostPairing.Endpoint`: the two halves talk over the socket like any other pair, and the host recognizes its own client the same way it recognizes any client. This is what the underlying `Synapse` transport defaults to in the engine.

The component ships the opposite default from the engine (`Local` here, `Endpoint` in `Synapse` itself) because a game hosting a session wants its own half answered at once, without a frame of its own traffic risking discard on the wire. Point `Remote Host` at another machine and this setting changes nothing — the pairing only engages for a loopback address.

Reading and writing values from a script

`SynapseTransport` exposes `Port`, `RemoteHost`, and `HostPairing` as properties over its serialized fields:

```
public ushort Port { get => _port; set => _port = value; }
public string RemoteHost { get => _remoteHost; set => _remoteHost = value; }
public HostPairing HostPairing { get => _hostPairing; set => _hostPairing = value; }
```

`Port` here is a `ushort`, matching the engine's `Configuration.Port`. `RemoteHost` is a `string` on this component — it gets parsed into an `IPAddress` only when the transport is added to the `CoreManager`, unlike the underlying `Synapse.RemoteHost` property in the engine, which is already an `IPAddress`. An invalid string will fail to parse at that point, so keep it a valid address literal.

Synapse Transport component

> **Driving the core API directly?** See The Synapse transport.

`SynapseTransport` is the Inspector-configurable `NetworkTransport` for the built-in UDP transport, Synapse. Place it on the `UnityCoreManager`'s GameObject and edit its fields there; `UnityTransportManager` reads every `NetworkTransport` component on that GameObject and adds each to the `CoreManager`. When it finds none, it adds a `SynapseTransport` itself, so a project with no transport component configured still gets Synapse with its defaults.

A custom editor (`SynapseTransportEditor`) groups the fields into three labeled sections in the Inspector: Connection, Timeouts, and Hosting.

Inspector fields

Connection

Port (`ushort`, default `7777`): the port the server listens on and the client connects to.

Remote Host (`string`, default `"127.0.0.1"`): the address a client connects to. The server ignores it: a host's server half never reads `RemoteHost`, since a server just listens on `Port` rather than dialing anywhere.

Maximum Transmission Unit (`ushort`, default `1200`): the largest packet built before it is split into multiple packets.

Timeouts

Idle Timeout Seconds (`ushort`, default `15`): how long a connected peer waits without hearing from the other side before it drops the link. One second is the least value applied.

Connecting Timeout Seconds (`ushort`, default `10`): how long a client waits for the server to answer its connect before giving up and returning to `Disconnected`. Zero falls back to **Idle Timeout Seconds**.

Hosting

Host Pairing (`HostPairing`, default `HostPairing.Local` on this component): how this transport reaches its own client half when the peer running it is also the host.

- `HostPairing.Local`: the server and client halves are paired in process and never exchange a datagram. The link is named the instant it's made, and a host's own traffic never crosses the socket to be discarded on arrival. The cost is that a host no longer proves anything about the transport through its own half — a genuinely remote client of the same host is unaffected and still goes over the real socket.
- `HostPairing.Endpoint`: the two halves talk over the socket like any other pair, and the host recognizes its own client the same way it recognizes any client. This is what the underlying `Synapse` transport defaults to in the engine.

The component ships the opposite default from the engine (`Local` here, `Endpoint` in `Synapse` itself) because a game hosting a session wants its own half answered at once, without a frame of its own traffic risking discard on the wire. Point `Remote Host` at another machine and this setting changes nothing — the pairing only engages for a loopback address.

Reading and writing values from a script

`SynapseTransport` exposes `Port`, `RemoteHost`, and `HostPairing` as properties over its serialized fields:

```
public ushort Port { get => _port; set => _port = value; }
public string RemoteHost { get => _remoteHost; set => _remoteHost = value; }
public HostPairing HostPairing { get => _hostPairing; set => _hostPairing = value; }
```

`Port` here is a `ushort`, matching the engine's `Configuration.Port`. `RemoteHost` is a `string` on this component — it gets parsed into an `IPAddress` only when the transport is added to the `CoreManager`, unlike the underlying `Synapse.RemoteHost` property in the engine, which is already an `IPAddress`. An invalid string will fail to parse at that point, so keep it a valid address literal.

Transport Manager component

Driving the core API directly? See TransportManager.

`UnityTransportManager` is the Inspector-side driver for the core transport manager. On `ManagersInstantiated` it grabs the core `TransportManager` off the `CoreManager` and applies its `Default Channel` field to it. If you have not added a `UnityTransportManager` yourself, `UnityCoreManager` adds one for you.

Inspector fields

Field	Type	Default	Description
Default Channel	`Channel`	`Channel.Unreliable`	The channel a send takes when it names none.
Automatic Start Mode	`NetworkStartMode`	`None`	What to start as when play begins: `None`, `Server`, `Client`, or `Host`. There is no core-manager equivalent of this field; it exists only on the Unity component.

Transport discovery

`UnityTransportManager` does not configure transports itself. It looks for `NetworkTransport` components on its own GameObject and its children, in the order Unity finds them, and adds each one to the core manager. If it finds none, it adds a `SynapseTransport` component so the object always has a usable transport.

Each `NetworkTransport` is an Inspector-configurable wrapper around a pure-code `Transport`; it is responsible for adding and configuring that underlying transport when asked.

Driving it from a script

```
UnityTransportManager transportManager = GetComponent<UnityTransportManager>();

await transportManager.StartServerAsync();
await transportManager.StartClientAsync();
await transportManager.StartHostAsync();

// Adds every discovered transport to the core manager without connecting.
// StartServerAsync/StartClientAsync/StartHostAsync call this for you.
await transportManager.EnsureAddedAsync();

// Removes and shuts down every transport this component added.
transportManager.ShutdownAll();

IReadOnlyList<Transport> transports = transportManager.Transports;
```

`EnsureAddedAsync` only adds transports once; later calls return immediately if the core manager already

has transports. `ShutdownAll` also runs automatically from `OnDestroy`, unless the core manager has already torn itself down.

The core manager

`UnityTransportManager.NucleusTransportManager` is the underlying `Nucleus.Managers.Transports.TransportManager`. Anything this component does not expose — settings and behavior beyond the default channel and the transport list — is on that core manager instead.

Bundle Ids for Unity Assemblies

Driving the core API directly? See Type Identity on the Wire: Bundles and the Registry.

Declare the bundle

Every networked type is registered under a bundle id and a local id. Two separately compiled assemblies that both declare no bundle collide, because an assembly declaring nothing defaults to bundle zero and the local id numbering restarts inside it. If you build a content shard as its own asmdef, give that asmdef its own bundle id before you ship it alongside anything else.

Put the attribute in a C# file inside the asmdef whose types you are numbering, at assembly scope:

```
using Nucleus.Serializers;

[assembly: NetworkBundle(ContentBundleShard.BundleId)]
```

The gated-prefabs content shard demo does exactly this. Its `ContentBundleShard` class exists only to carry a named constant for the id:

```
public static class ContentBundleShard
{
    public const ushort BundleId = 1;
}
```

The assembly that carries the attribute has to be the asmdef whose types you're numbering, not a shared assembly the shard happens to reference. The editor resolves a prefab collection's bundle id by reflecting over the assembly the collection names, so folding the attribute into a shared demos assembly would pull every prefab in that assembly into the same bundle.

Ids already taken

- `NetworkTypeRegistry.FrameworkBundleId` — the framework's own bundle, `ushort.MaxValue`.
- `Nucleus.Integrations.Unity`'s `AssemblyInfo.cs` — `FrameworkBundleId - 1`, one below the framework so it can't collide with either the framework or a default user assembly.
- The integration's test assembly — `FrameworkBundleId - 2`, one below that, for the same reason: the tests and the demos both load in the editor, and without a distinct id the first of them to register takes bundle zero and the other's components are refused.
- A Unity assembly that declares no `[assembly: NetworkBundle(...)]` at all defaults to bundle zero.

Pick an id outside this set for every shard you add.

A different bundle number on the same component

`NetworkPrefabCollection` carries its own `PrefabBundleId` and `BundleAssemblyName` serialized fields. These drive `NetworkPrefabRegistry`, the separate registry that resolves which prefab a spawn packet names — not `NetworkTypeRegistry`, the registry the `[assembly: NetworkBundle(...)]` attribute feeds. The two numbers look alike and both call themselves a bundle id, but they govern different registries and are populated independently: `NetworkPrefabCollection`'s fields are filled in by the editor scan of the collection's own assembly, not by anything you set directly.

What a collision looks like

If two types land on the same `(bundleId, localId)` pair, `NetworkTypeRegistry` refuses the second

registration outright and logs an error naming the bundle and local id involved. The type is not registered under the wrong class — it is simply absent from the registry, and any attempt to spawn or replicate it fails. If a component or system silently never spawns and the console has a registration error naming a bundle and local id, check for a missing or duplicated `[assembly: NetworkBundle(...)]` first.

Assigning ids

Assign shard ids from the low end and keep going: 1, 2, 3. This isn't just convention — it's cheaper on the wire. `BundleIdWireFormat`, the internal format the registry ids are packed with, is tuned so both ends of the id space cost the fewest bits, with the cost rising the further an id sits from either end. A low id picked from the start of the range costs a few bits; an id picked from the middle of the range costs about twenty. There's no reason to burn wire budget by picking an arbitrary or high id when the low end is open.

See the core API page for `NetworkTypeRegistry` itself, and for declaring the same attribute in a csproj-built (non-Unity) assembly.

Code Generation in Unity

Driving the core API directly? See Wiring the Source Generator into a .NET Project.

What has to be in Assets/Nucleus

Generation happens inside the Unity compiler, driven by a Roslyn source generator that ships as a DLL. For it to run, `Assets/Nucleus` needs four files side by side:

- `Nucleus.dll` - the engine
- `CodeBoost.dll` - pooling and collection support the engine depends on
- `SynapseSocket.dll` - the transport
- `Nucleus.CodeAnalysis.SourceGenerators.dll` - the generator itself

Unity only treats a DLL as a source generator when its `.meta` file carries the `RoslynAnalyzer` label:

```
labels:
- RoslynAnalyzer
```

That label, and the asset GUID Unity uses to track the file, live in the `.meta`, not the DLL. When you update the generator, replace `Nucleus.CodeAnalysis.SourceGenerators.dll` and leave its `.meta` alone. Replacing the `.meta` too (or deleting and re-adding the DLL) drops the label, and Unity silently goes back to compiling without it - no error, just a generator that never runs.

Which assembly your component needs to be in

The generator inspects each assembly Unity compiles and skips any assembly that doesn't reference `Nucleus.dll` - it has nothing to generate there. Whether that reference exists depends on where your `NetworkComponent` lives:

- **Scripts with no asmdef** compile into `Assembly-CSharp`, which references `Nucleus.dll` automatically. `Nucleus.Integrations.Unity.asmdef` is marked `autoReferenced: true`, so any assembly Unity auto-references picks it up without you adding anything.
- **Scripts under your own `.asmdef`** need an explicit reference to `Nucleus.Integrations.Unity` (or another assembly that itself references `Nucleus.dll`) added to that asmdef's `references` array.

`Nucleus.asmdef`, the core engine's own assembly definition, sets `noEngineReferences: true`: the engine assembly does not automatically reference `UnityEngine`/`UnityEditor`, keeping it free of a direct engine dependency. In the shipped Unity workflow the engine reaches your project as the precompiled `Nucleus.dll` in `Assets/Nucleus`, not as this asmdef's source.

The C# 9 ceiling

Unity's generated project pins `LangVersion` to 9. Anything Unity compiles - your components, the integration source, generated partials - is limited to C# 9 syntax: namespaces need braces (no `namespace Foo;`), and there's no collection-expression syntax (`[1, 2, 3]`). This ceiling applies only to source Unity compiles. It doesn't touch the prebuilt DLLs in `Assets/Nucleus`, which are built against a newer language version outside Unity.

Symptom: a component that compiles but replicates nothing

If the generator didn't run against your assembly, a `partial NetworkComponent` still compiles - the class is valid C#, it just has no generated half. Both the serializer (the Write/Read pair) and the

`[ModuleInitializer]` registration that wires the type into the network system are missing. Nothing throws. The component sits in the scene, does nothing on the wire, and the only sign is silence: no errors, no replicated state, no console warning pointing at the cause.

Check first: is the DLL's `.meta` still carrying `RoslynAnalyzer`, and does the component's assembly reference `Nucleus.dll`?

Where generator errors surface

Generator diagnostics appear in the Unity console, using the same diagnostic ids the .NET build reports for the same problems. See the diagnostics reference for what each id means.

See also

For what the generator emits, and for wiring it into a plain .NET project outside Unity, see Wiring the Source Generator into a .NET Project.

Generator Errors and Warnings

Diagnostics appear in your build output or the Unity console

The source generator runs as part of the normal C# compile. In a plain .NET project it reports through the ordinary build output or your IDE's error list, same as any other compiler diagnostic. In Unity, the generator DLL is imported with the `RoslynAnalyzer` label so Unity runs it as part of scripting compilation; its diagnostics land in the Unity console alongside regular compile errors.

Every diagnostic carries an id in the form `FAMILY###`, a severity (error or warning), and a message built from a plain format string rather than a fixed sentence per id. Reading the id tells you which stage raised it; reading the message tells you what to fix. Look up the page by the message text you actually see, not by the id alone.

The id families

Id	Severity	Stage
`NETWORKTYPEFINDER00`	Warning	Type discovery (`NetworkTypeFinder` walking your networked types and members)
`NETWORKTYPEFINDER01`	Error	Type discovery
`GENERATION000`	Error	Generation - invalid type
`GENERATION001`	Error	Generation - invalid Roslyn type
`GENERATION002`	Error	Generation - comparer/operation stage
`GENERATION003`	Warning	Generation - comparer/operation stage (declared, not currently raised)
`GENERATION004`	Warning	Generation - wire hash collision between two networked messages or calls
`SERIALIZERS000`	Error	Serializer generation
`SERIALIZERS001`	Warning	Serializer generation

`NETWORKTYPEFINDER*` fires first, while the generator is still deciding which types and members are networked at all. `GENERATION*` and `SERIALIZERS*` fire afterward, while it builds comparers, readers, and writers for what discovery found. `GENERATION004` is the exception: it is a late, non-fatal check that two networked messages (or two networked calls) hashed to the same wire id. It only reports; it never stops the rest of the assembly from generating, because renaming one of the colliding types is a fix to make, not a reason to withhold everything else the build would otherwise emit. While two types collide, the engine sends neither, registers neither, and hands neither to a handler.

The commonest cause: a member with no serializer

By far the most common diagnostic is `SERIALIZERS001`, with a message shaped like:

`[Type.Member] Type [SomeType] for [Write/Read] SerializeMethod could not be found; this value will not be networked.`

This means the generator could not find a serializer for that member's type. It has two fixes:

- **Marker struct** - if the type lives in an assembly you don't control (a `System.Numerics` type, a third-party math type), add it as a field on a marker struct carrying `[NetworkType]` so the generator can reach it. See Reaching Types You Cannot Annotate.
- **Hand-written serializer** - if the type's shape genuinely needs custom read/write logic rather than just visibility. See Writing a Custom Serializer.

A `UnityEngine` type (`Vector3`, `Quaternion`, and so on) does not raise this warning: the generator reaches the type's public fields and writes a plain serializer for it. Declare the `System.Numerics` equivalent anyway. See Unity Types and System.Numerics.

`SERIALIZERS000` is the error-level sibling of the same stage: a required serializer is missing or malformed in a way the generator cannot route around at all, for example `A [Write/Read] serializer could not be found for [Type].` or a type exceeding the maximum networked member count. Treat it the same way as `SERIALIZERS001`, just with a build-breaking severity.

The second commonest: a type the generator can't fill in

`NETWORKTYPEFINDER01` covers a family of structural problems in how a networked type is declared, including type arguments that aren't currently supported and a member type discovery doesn't recognize. The one worth calling out specifically: a networked component or system that is not declared `partial`, or that is nested inside another type in a way the generator cannot reach, fails as an ordinary Roslyn compile error (for example a missing-partial-modifier error) rather than a Nucleus-specific diagnostic, because the generator is trying to emit a partial declaration into a type it does not control. If your networked type is producing raw C# compiler errors about a generated file rather than a `NETWORKTYPEFINDER`/`GENERATION`/`SERIALIZERS` id, check that the type and every type containing it is `partial` and check its nesting.

Read the top diagnostic, not the longest list

The generator's stages run in order, and an early error aborts the run for that type before later stages get a chance to run. A discovery-stage error on `NETWORKTYPEFINDER01` commonly produces a long tail of `SERIALIZERS001` warnings underneath it, because every member the generator never got to walk correctly also has no serializer. Fixing the first error in the list is usually enough to make the rest disappear on the next build. Scroll to the top of the diagnostics for a type, fix that one, and rebuild before working through the rest.

A generator diagnostic is a compile-time problem, never a runtime one

A generator diagnostic is a compile-time problem: the generator could not produce correct code for your types. It has nothing to do with runtime behavior. If your build is clean but replicated values are wrong, systems never converge, or packets look malformed at runtime, that is a wire or convergence problem, not a generator diagnostic, and belongs on a different page.

How a Value Reaches the Wire

> New to Nucleus? Start with How replication works first — this page assumes you already have the pipeline overview and goes one level deeper, into the encoding itself.

Setup: Reference the source generator (plain .NET) or Installing the Unity integration (Unity).

The path, end to end

A `NetworkMember<T0>`'s `Value` setter is the start of the path. It mutates the member's live ring slot and notifies the owning system that something changed — nothing is written yet.

At serialize time, the component's generated writer encodes each changed member against the shared baseline both peers hold, producing a delta payload. That payload becomes the body of a framed subpacket: a `StatePacketType` (`Delta`, `Full`, `Controller`, and the rest), the subpacket's bit length, then the body. Multiple kinds get packed into one combined stream for the tick this way — a spawn, a controller change, and a batch of deltas can all ride together.

`PacketSegmentWriter` takes that combined stream and fits it into one or more `StatePackets`, splitting at the maximum transmission unit when the tick's data doesn't fit in one. Each `StatePacket` carries a `SegmentNumber` and `ExpectedSegmentCount` so the receiver can rejoin a tick's segments before demuxing the frames inside. The segmented packets are then handed to the transport for the peer.

On the receiving end this runs in reverse: rejoin segments into one reader, walk the frames in order (`Full`, then `Controller`, then `Delta` — so a spawn exists before anything addresses it), and each component's generated reader decodes its members back out.

Values are packed at bit granularity, not byte granularity

Values are packed at bit granularity, not byte granularity. A member carries a length indicator alongside its data, so the wire cost of "a float" or "a Vector3" has no fixed answer — it depends on how far the value moved since the last tick it was sent. A value that barely changed writes short; one that changed a lot writes long. (How that length indicator itself is encoded isn't covered here — the granularity and the cost trade-off are what matter at this level.)

Every serializer is generated at build time, never by reflection

Every `Write`/`Read` method your components use is emitted by the Nucleus source generator at build time, and each one registers itself at assembly load through a module initializer. There is no reflection-based serialization path, and no runtime type inspection to find or build a serializer — a type either has generated code for it or it doesn't compile against the network path. That's what keeps the engine AOT-friendly: nothing here depends on `System.Reflection.Emit` or similar runtime code generation that AOT platforms disallow.

Three things you can tune

- **Compression level and accuracy — how tightly.** `CompressionLevel` (`Tight` by default, or `Aggressive`) is fixed at a `NetworkMember<T0>`'s construction. `Accuracy` sets the wire accuracy used for delta encoding, loss recovery, and lossy serialization; it defaults per-type (tighter for a `Quaternion` than a general float) and can be coarsened for a value that tolerates more error, like a velocity.
- **Transmission mode and send interval — how often, or whether at all.** `TransmissionMode.Interval` (the default) sends ordinary deltas paced to a `SendInterval`;

`SendInterval.Normal` sends every changed tick, and a wider interval accumulates changes and rides at most one delta per span. `TransmissionMode.Divine` projects the value instead of sending ordinary deltas, correcting only when needed, so it can go quiet for stretches where `Interval` would still be sending — the mechanism itself isn't something you need to reason about to use it, and it is Pro-only.

- **`[ReplicationIgnore]` — whether the code exists at all.** A member declared with `ReplicationIgnoreAttribute` gets no generated serialization: the generator skips it entirely rather than serializing-and-discarding it, so it's not a runtime cost you pay and choose not to use — it's code that was never emitted.

Most state rides an unreliable channel by design

Most state rides an unreliable channel. Correctness doesn't come from retransmitting a lost packet — it comes from acknowledgment, redundant resends, and targeted recovery for a peer that's fallen behind. See the reliability and recovery section for how that works; this page only needs you to know that "unreliable" is the normal case for state, not an edge case to work around.

Glossary

- **Full** — a complete serialization of every member, used for spawns and resyncs. Reconstructs a system from nothing.
- **Delta** — a member encoded against the shared baseline, carrying only what changed.
- **Baseline** — the last value both peers agree the member held, which a delta is encoded relative to.
- **Framed subpacket** — a `StatePacketType`, its bit length, then its body; the unit the combined per-tick stream is built from.
- **Segment** — one `StatePacket` produced by splitting the tick's combined stream to fit the maximum transmission unit.
- **Bundle id** — part of how a spawned system's type is addressed on the wire; not explained further here.
- **Projection** — the mechanism behind `TransmissionMode.Divine`: a value rides a predicted path between corrections instead of being resent every interval.

Serialization, Codegen and the Wire

Pages in this category:

- How a Value Reaches the Wire
- Code Generation in Unity
- Generator Errors and Warnings
- The Shapes on the Wire
- Bundle Ids for Unity Assemblies
- Unity Types and System.Numerics

The Shapes on the Wire

Packet types

Every packet the transport hands off carries a `PacketType` in its header.

- **Message** — one-shot single-packet messages, deserialized immediately.
- **State** — tick-aligned data that may span several segments.
- **Reconcile** — correction data sent to clients; only ever received on a client, which replays forward from that point after applying the correction.
- **Raw** *(DEBUG builds only)* — raw data with no special formatting.
- **Rpc** — system-addressed remote calls, deserialized as they arrive rather than on a tick boundary.

`Raw` exists only under `#if DEBUG` and sits at value 3. `Rpc` is declared past it and held at value 4 whether or not `Raw` is compiled in, so a debug peer and a release peer read the same header regardless of build configuration.

State subpacket types

A `State` packet's body is not one thing — it is a sequence of framed subpackets, each tagged with a `StatePacketType`. Seventeen kinds exist:

Kind	Carries
`Delta`	Delta-serialized system state: members encoded against their shared baseline, for a change the sender made itself. The default, cheapest kind — a world that never grants write access emits only this one.
`Full`	Full-serialized system state, used for spawns and resyncs.
`Controller`	Controller changes, as a list of `(SystemId, ControllerConnectionId, ControllerRetentionToken)` entries.
`ClientState`	Tick-aligned client state: upstream from the controlling client to the server, and from the server to the system's observers, so an observer predicting the system can simulate from it.
`Reconcile`	Server?controller state correction carrying the server's full state for a system at the current tick.
`Ack`	A standalone acknowledgment carrying the receiver's highest fully-applied tick that carried state. Written only when that value has advanced and there's no other outbound traffic to piggyback it on.

`Recovery`	Targeted loss recovery: the served-through tick followed by absolute full values for each recovered system's changed components. Systems spawned since the acknowledged tick ride `Full` instead.
`Despawn`	System despawns, as a count followed by the stopped systems' Ids. Server-only; written after that tick's state subpackets.
`WriteAccess`	Who may write a system's state, as `(SystemId, access, may-this-connection-write)` entries. Server?client only.
`RpcAccess`	Who may send a system a remote call, as `(SystemId, access, may-this-connection-send)` entries. Server?client only.
`DeltaRelayed`	Delta-serialized state the sender is passing on from another peer, rather than a change it made itself.
`DeltaAbsolute`	Delta-serialized state whose changed members carry self-contained absolute values (not baseline-encoded), for a change the sender made itself.
`DeltaAbsoluteRelayed`	Same as `DeltaAbsolute`, but relayed from another peer.
`RosterAdd`	Peer-roster additions, as a count followed by added peers' `Connection.Id`s. Server-only; written ahead of every other kind.
`RosterRemove`	Peer-roster removals, as a count followed by departed peers' `Connection.Id`s. Server-only; written after that tick's despawns.
`SceneMove`	A system's scene changing, as a count followed by `(SystemId, SceneHandle)` entries. Server-only, to established observers only.
`Reparent`	A system's parent changing, as a count followed by `(SystemId, ParentId)` entries. Server-only, to established observers only.

`DeltaAbsolute`/`DeltaAbsoluteRelayed` exist because a plain `Delta` can't say two things a receiver needs and can't re-derive: whether the payload is baseline-encoded or self-contained, and whether the sender authored the change or is relaying somebody else's. Carrying those as the subpacket kind rather than as per-system bits pays for itself once enough systems share the same answer in a tick.

Framing and apply order

Each subpacket is written as its `StatePacketType`, then its bit length, then its body. This makes the stream self-describing: a receiver that doesn't recognize or can't bind an entry steps over it whole using the declared length, instead of losing alignment for everything after it.

A tick's segments are rejoined into one `Reader` before any of this is read. Within that combined stream the declared write order is `RosterAdd`, spawns (`Full`), `Controller`, `SceneMove`, `Reparent`, then the delta kinds, then `Despawn`, then `RosterRemove`. The receiver applies frames in that order — `Full` before `Controller` before `Delta` — so a spawn exists before control or state addresses it, and a despawn is applied only after that system's final state has landed.

RpcRoute

An RPC packet carries an `RpcRoute` naming where the call is headed, packed into two bits by the generated serializer:

- **Server** — a client's call to the server; it stops there.
- **Observers** — a call to every connection observing the system. Legal in both directions: from the server it *is* the fan-out; from a client it's a request for one, admitted and judged by the server's handlers before going out to the other observers.
- **Target** — the server's call to a single observing connection. Server-only; one arriving from a client is discarded and logged, since no engine path produces it.
- **TargetRelay** — a client's call to a single connection it named, passed on once the server admits it. The server rewrites the route to `Target` before forwarding, so the recipient can't tell a routed call from one addressed to it directly.

Direction is validated against the sender's role on receipt, not inferred from which link the packet arrived on — a host holds connections in both directions, so the role has to be checked explicitly.

Segmentation

A `State` packet's combined stream can be too large for one transport unit, so it's split across segments. `StatePacket` carries `SegmentNumber` and `ExpectedSegmentCount` for this: segments span the *whole* combined stream for a tick — every subpacket kind is packed and numbered together — so when everything fits in one transmission unit, `ExpectedSegmentCount` is 1.

`PacketHeader.MaximumOutboundHeaderBytes` (48) is the space reserved ahead of a segment's body for the packet header and `StatePacket` wrapper fields (packet type, tick deltas, acknowledgment ticks, segment number and count) before the transport decides how much room is left for payload. Reserving it up front means the transport never has to run a second segmentation pass after the header grows into space it thought was payload.

`SegmentMode` names how a segment is filled — `Strict` (data must fit the maximum segment size), `Overfill` (exceed it rather than split), `Split` (always fill to the maximum, splitting the data across multiple segments) — but it's an internal enum, not something reachable from outside the engine.

IncomingPacket and OutgoingPacket

`IncomingPacket` is what a received packet looks like once it's on its way into the engine: `Payload` (the raw bytes to read), `Channel`, `Sender` (the `Connection` that produced it), `Invoker` (which role sent it — Server or Client, derived from `Sender.IsServer`), and `Transport` (the transport it arrived on). `ReturnPayloadArray()` returns `Payload`'s backing array to the shared `ArrayPool`.

`OutgoingPacket` is the send-side counterpart: `Payload` (the bytes to send), `Channel`, `Sender`, `Receiver` (the connection to route to), `Writer` (the `Writer` that produced the payload), and `Return()`, which returns the `Writer` to its pool and, if the payload was copied into a pooled array of its own, returns that array too.

The convention across both types is one owner copies once, and one owner returns: a payload lives in its

writer's buffer unless a later stage copies it out, and whichever struct ends up holding that copy is the one responsible for returning it.

This page is for reading the wire, not for writing packets

`PacketHeader` is a public type, but every member on it is `internal`, and `SegmentMode` is `internal` as well. Nothing here is a public API for writing packets — this page is for reading what's on the wire (a log line, a capture, a framing error), not for constructing packets yourself.

Unity Types and System.Numerics

Declare members in System.Numerics, not UnityEngine

The engine core replicates `System.Numerics` types. A networked member is declared as `NetworkMember<System.Numerics.Vector3>`, never `NetworkMember<UnityEngine.Vector3>`. Nucleus registers its vector and rotation handling, such as interpolation and Divine projection, for the `System.Numerics` types only. A member declared with a UnityEngine type still builds, but the generator treats the type like any struct of your own: it writes a plain serializer from the type's public fields and settable properties, and none of that handling applies. For `UnityEngine.Quaternion` that plain serializer also picks up the settable `eulerAngles` property, so every value carries the rotation twice.

The shipped `UnityNavMeshAgentComponent` follows this exactly:

```
using NumericsVector3 = System.Numerics.Vector3;

public partial class UnityNavMeshAgentComponent : NetworkComponent
{
    public readonly NetworkMember<NumericsVector3> NextPoint = new(CompressionLevel.Aggres
sive);
    public readonly NetworkMember<NumericsVector3> PreviousPoint = new(CompressionLevel.Ag
gressive);
}
```

The `NumericsVector3` alias is just a using directive disambiguating from `UnityEngine.Vector3` in the same file. It's the pattern to copy: alias `System.Numerics` types at the top of a Unity-facing script rather than writing the fully qualified name at every member declaration.

Convert at the Unity boundary

`UnityConversionExtensions` converts between the native type a member stores and the Unity type a `Transform` (or other Unity API) expects: `ToUnity()` for native to Unity, `ToNative()` for Unity to native. It covers every type both libraries define: `Vector2`, `Vector3`, `Vector4`, `Quaternion`, `Plane`, and `Matrix4x4`.

Call them at the moment you read a member into a `Transform` or write a `Transform` into a member, not before. `UnityNavMeshAgentComponent` does exactly this in its Unity-facing properties:

```
public Vector3 ReplicatedNextPoint
{
    get => NextPoint.Value.ToUnity();
    set => NextPoint.Value = value.ToNative();
}
```

Everything above that boundary — the `NetworkComponent`, its `NetworkMember<T0>` fields — stays native. Everything below it — the `MonoBehaviour`, the `Transform`, `NavMeshAgent` — stays Unity. Nothing in between sees both.

Types with generated serializers already available

Besides the `System.Numerics` vectors (`Vector2`, `Vector3`, `Vector4`), these CodeBoost types already have generated serializers and can be used in a `NetworkMember<T>` from Unity code without any attribute of your own: `Vector2Int`, `Vector3Int32`, `Vector4Int`, and `UnifiedColor`.

All seven are reached through `ExternalNetworkTypes`, a marker struct in `Nucleus.Serializers`

carrying `[NetworkType]` on behalf of types declared in assemblies nobody here can annotate. Nothing reads or constructs the struct at runtime; the generator only walks its field types to know which external types need serializers.

Your own structs need no attribute

A struct you declare yourself, made entirely of natively-serializable fields, needs no `[NetworkType]` attribute at all as long as a `NetworkMember<T>` somewhere reaches it. The generator discovers it through that member, the same way it discovers any other networked type.

The error you get instead of a crash

An unserializable member type is caught at build time, not at runtime. The generator reports a `SERIALIZERS`-series diagnostic (`SERIALIZERS000` as an error, `SERIALIZERS001` as a warning) naming the member. A member declared as `NetworkMember<UnityEngine.Vector3>` is not caught this way: the generator can read the type's public fields, so it builds without a diagnostic and replicates through the plain serializer described above.

Adding a type you can't annotate

For a third-party type you don't own and can't put `[NetworkType]` on, see the `ExternalNetworkTypes` reference for how to add a marker for it.