PHOTON FUSION

FOR UNREAL ENGINE 5

Build a scalable, shared-authority multiplayer using the power of C++

VICTOR CHANET

Founder of Victor Game Studio and software engineering instructor

PHOTON FUSION

FOR UNREAL ENGINE 5

Build a scalable, shared-authority multiplayer
using the power of C++

VICTOR CHANET

Photon Fusion for Unreal Engine 5
First edition — 2026

Set in Audiowide and Inter. Code set in JetBrains Mono.
Written, designed, and typeset by Victor Chanet.

CONTENTS

PREFACE

Multiplayer is one of the most worthwhile things you can add to a game. It's also one of the hardest, and this book exists because of the gap between those two facts.

The appeal is straightforward. Single-player content is finite. You build it, players consume it, and eventually they're done. Other people aren't. They cooperate, compete, and surprise each other in ways no script can produce, which is why multiplayer is what pushes a game to its highest level of interactivity. It's a goal worth setting.

But it comes with constraints, and they're worth understanding before you commit, because the hard part of multiplayer often isn't the gameplay code. It's everything around it.

The first constraint is cost. Multiplayer can be expensive. Servers, bandwidth, and matchmaking generally bill you every month, and much of that cost is fixed whether ten people are playing or ten thousand. You take on an ongoing expense before you ship anything, and more than one promising game has been sunk not by bad design but by an infrastructure bill that outpaced its revenue. You need to know what you are signing up for.

The second constraint is your workflow. The solution you pick can make day-to-day development much heavier. Dedicated servers are the clearest example. Choose that path and you cannot use the prebuilt Unreal Engine that Epic provides. You have to compile the engine from source yourself, package it, and distribute that custom build to your whole team, and you repeat that work every time the engine changes. Every step is one more thing to maintain and one more thing that can break, on top of the game you set out to build.

None of this is a reason to avoid multiplayer. It's a reason to choose your approach deliberately, and it's the problem Photon Fusion is designed around.

Fusion's advantage is that it doesn't change how you work. You stay on the same Unreal Engine you'd use for a solo project; the build from Epic is all you need. There's no engine to compile from source and no custom binary to push to your team. The workflow that already works for single-player keeps working.

The cost model is just as practical. It starts low and scales with your game. If a title doesn't find an audience, you aren't paying for players who never arrived. If it succeeds, your costs rise, but proportionally and predictably. Spending tracks revenue instead of running ahead of it, which makes multiplayer something you can budget for. And because you aren't running the infrastructure yourself, you don't need an ops team to keep it alive. That is what puts scalable multiplayer within reach of a small studio, or a single developer.

Why I wrote this book

Most Unreal multiplayer is built around a single idea: one server is in charge of the game, and everything checks back with it to know what's true. That central authority works, but it shapes how you write every piece of code. Fusion takes a different approach. It spreads that responsibility across the individual objects in your game, so each one looks after itself. Once that shift in thinking clicks, the rest of the framework follows naturally.

This book is built to make that click happen, whether or not you've written a line of networked code before. It starts from the ground up and builds each idea on the one before it, staying close to real C++ examples the whole way. Everything in it reflects how Fusion behaves in a project that actually shipped, and wherever there's a common mistake waiting, I've marked a Pitfall so you can step around it.

A note on the tools we use

Now and then I'll mention USourceControl, the version-control system I built for game teams and that we use at Victor Game Studio every day. I bring it up because multiplayer projects are merge-heavy and binary-heavy, and the way you manage your source has a real effect on

how painful the work becomes. If you build games, the problems it solves are probably already on your desk. I'll keep these notes short and clearly labeled, and you can skip them entirely if you like.

Let's build something that scales.

Victor Chanet

HOW TO READ THIS BOOK

This book serves two readers at once. If you're new to networked gameplay, read it front to back: each part builds on the one before, and the early chapters assume nothing beyond comfort with Unreal C++. If you're an experienced engineer, treat it as a reference. Every chapter is self-contained enough to open on its own, and the tables and code listings are meant to be the thing you actually copy from.

What you need

- **Unreal Engine 5.4 or newer.** Fusion ships as an engine plugin and targets 5.4+.
- **A C++ project.** Every example is C++. Where a workflow also exists in Blueprint, an In Blueprint note explains the equivalent.
- **A Photon account.** The free tier is enough to follow along; you'll paste an App ID into Project Settings in Chapter 2.
- **Chaos physics enabled.** This is the Unreal 5 default, and it's required for the Forecast physics module in Chapter 6.

The marginal notes

Five kinds of callout appear throughout the book. Each one is a different promise about the information inside it:

> IN BLUEPRINT **In Blueprint**
>
> Every code example in this book is C++. When the same thing is done differently, or exposed under a different name, in the Blueprint editor, this note tells you how, so a mixed C++/BP team stays on the same page.

> PITFALL **Pitfall**
>
> A mistake I've made, watched someone make, or that the framework actively invites. If you read only one kind of callout, read these.

> UNDER THE HOOD **Under the Hood**
>
> How Fusion implements something beneath the public API. Safe to skip on a first read; useful when you're debugging at 2 a.m.

> NOTE **Note**
>
> A clarification or cross-reference worth knowing that does not change what you type.

> TIP **Tip**
>
> A small recommendation that tends to make production life easier.

On code style

Listings are deliberately short. They show the line that matters, not a compilable file. Member variables such as `OnlineSubsystem` and `FusionActorComponent` are assumed to be cached on the owning class unless a listing shows otherwise. When a function name ends in `_Receive` or `_Implementation`, that's the body you write; the framework generates the matching send side.

PART ONE

FOUNDATIONS

What shared authority is, why it fits the games most teams actually ship, and how to get from an App ID to a live room.

CHAPTER 01

THE SHARED-AUTHORITY MODEL

Fusion replaces the single authoritative server with many small owners. Understand that one shift, and every other difference falls into place.

IN THIS CHAPTER

- From One Server to Many Owners
- Three Ways to Run a Multiplayer Game
- Owners, Authority, and Roles
- The Master Client
- Rooms and Sessions
- Coming from Built-in Unreal Netcode

Photon Fusion is a multiplayer networking SDK for Unreal Engine 5.4 and later. Before we write any code, there is one idea to settle, because everything else in the book builds on it: in a Fusion game, no single machine is in charge.

That needs unpacking, because it is easy to misread. In most multiplayer games, one computer acts as the server. It holds the official version of the world, decides what is true, and tells every player's machine what to show. Fusion does not work this way, and it is not peer-to-peer either, where one player's machine quietly runs the match for everyone. Instead, each player's game simulates the part of the world it owns, and a service on the Photon Cloud passes those results between players. This is called shared authority, and it is the foundation the rest of the book rests on.

If you are coming from Unreal's built-in networking, this is a real shift in how you think, not just a new set of function names. This chapter is here to make that shift early and on purpose, so the rest of the book feels obvious instead of surprising.

FROM ONE SERVER TO MANY OWNERS

Unreal's built-in networking uses a client-server model. One machine, either a dedicated server or a player acting as host, is the authority: it owns every networked object in the game, decides its state, and sends that state out to everyone else. (In Unreal's terms, that machine holds `ROLE_Authority`.) Every other player runs a copy that only observes. It can display and predict what the authority owns and send input back, but it cannot change the official state itself. There is exactly one source of truth, and it lives on that one machine.

Fusion removes that single source of truth. Every networked object has its own owner, which is simply a regular player's game, and that owner is the authority for that one object. The owner reads and writes the object's state; everyone else sees it but cannot change it. There is no global authority and no central server to defer to. A room on the Photon

Cloud holds the shared state and forwards updates to the players who need them, but it never runs your gameplay.

Each play session lives in exactly one room: a service on the Photon Cloud that holds the shared, room-wide state and sends property updates and RPCs to the players that need them. The Photon Cloud is the same proven infrastructure that powers thousands of live games. You do not set up or maintain any servers yourself; Photon handles hosting, scaling, and global routing across 21+ regions, with DDoS protection and uptime SLAs. Pricing is based on how many players are online at once, with a free tier for development.

THREE WAYS TO RUN A MULTIPLAYER GAME

It helps to put shared authority next to the two setups you have probably seen before. Each one suits a different kind of game, and the differences come down to cost and operation as much as code.

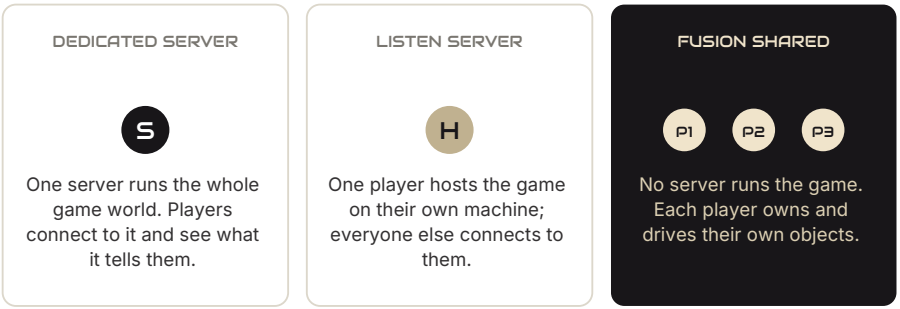


FIG 1.1 The three setups. A dedicated server and a listen host both keep authority in one place; Fusion spreads it across the players and relays through the room.

A dedicated server is a copy of your game running with no graphics on rented infrastructure, doing nothing but hosting the match. It gives the strongest protection against cheating, because players never write the official state themselves, but it is expensive to run and a lot to operate: you have to launch it, monitor it, and roll it out across regions. That cost usually only makes sense for competitive games.

A listen server is a normal player whose machine also hosts the match. It saves the hosting bill, but that machine becomes a single point of failure. Speed, uptime, and regional reach are only as good as whatever the host happens to have, and if the host disconnects, crashes, or sends the app to the background, the session ends for everyone.

Shared authority avoids both costs. The Photon Cloud handles regions, DDoS protection, and uptime, and the running cost per session stays low enough to ship a free-to-play game. The trade-off is cheat resistance: because each player writes their own state, ranked competitive PvP is still better served by a dedicated server. But co-op, casual, party, sandbox, social, persistent worlds, mobile, web, and XR are exactly where shared authority is strongest.

How the three setups compare on the things that actually decide a project.

	Fusion (shared)	Listen host	Dedicated
Reach	Cross-anything; F2P, mobile, XR, web	Cross-anything	PC/Console only in practice
Cost	Low and scalable	'Free' on Steam / EOS relay	Premium, 10x-50x
Operations	Simple	Relay; matchmaking and SLA on you	Advanced
Authority	Distributed; server-side code optional	Uncontrolled client host	Full server simulation
Host migration	Not necessary	Not available	Not necessary
Quality of service	21+ regions, DDoS, SLA	Whatever the host has	Orchestrated by you

OWNERS, AUTHORITY, AND ROLES

Every networked object in Fusion has an owner: a player id (an `int32`) that identifies which player is its authority. The owner reads and writes that object's state; everyone else can see it but not change it. In Fusion, "having authority" always means owning that specific object. There is no global authority and no server to ask.

In gameplay code, two checks do almost all the work: `IsOwner(Actor)` and `CanModify(Actor)`. Both look up the object's owner, and either one is the right thing to guard any change to networked state.

```
if (OnlineSubsystem->IsOwner(this))
{
    // Only the owning player may write networked state.
    Health -= IncomingDamage;
}
```

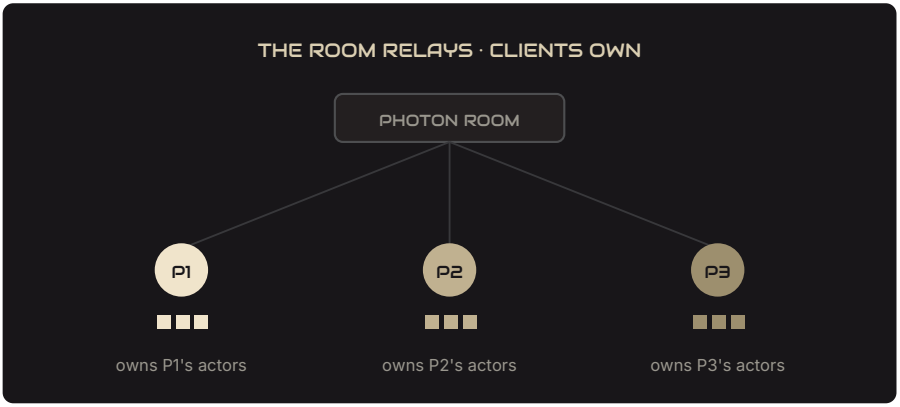


FIG 1.2 Under shared authority, each player owns and drives their own objects. The room relays the resulting state; it never runs the game.

This idea drives every other difference in the book. Cheat prevention, lag compensation, where your game logic lives, and how RPCs are aimed are all handled differently once you accept that authority belongs to objects, not to a single process. We will come back to the full set of

ownership modes (Transaction, PlayerAttached, Dynamic, MasterClient, and GameGlobal) and the runtime transfer API in Chapter 5.

> PITFALL **HasAuthority() is not your friend here**
>
> Fusion does not override `AActor::HasAuthority()`. On a Fusion-owned object it often happens to return true on the owner and false everywhere else, but that is a coincidence, not a guarantee. Guard your writes with `IsOwner` or `CanModify`, which ask the real Fusion owner directly.

THE MASTER CLIENT

Some decisions need exactly one player to make them. Who advances the match state? Who picks the next map? Who owns the shared objects that no single player should control? Fusion handles this by electing one player in every room as the Master Client.

Any player can check whether they are it with `UFusionOnlineSubsystem::IsMasterClient()`, and you run Master-only logic only when that returns true.

```
UFusionOnlineSubsystem* Subsystem =
    GetGameInstance()->GetSubsystem<UFusionOnlineSubsystem>();

if (Subsystem->IsMasterClient())
{
    AdvanceMatchState();  // exactly one player runs this
}
```

It is important to be clear about what the Master Client is not. It is not a server. It has no extra simulation power, no special trust, and no separate process. It is a normal player chosen to play a coordinator role. Every player still simulates their own objects and writes their own state, no matter who the master is. When the current Master Client leaves the room, Photon promotes another player automatically, and objects set to

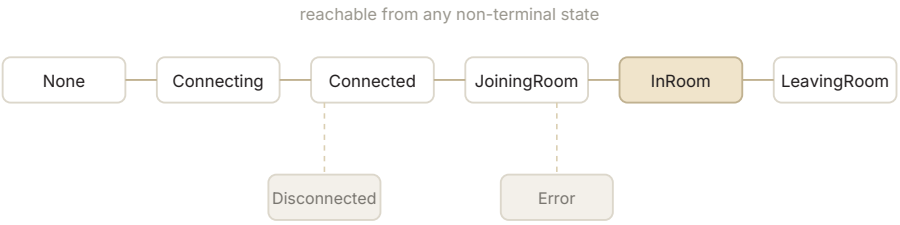
Master Client ownership move to the new master with no code from you.

> NOTE **Use it for coordination, not trust**
>
> Reach for the Master Client when exactly one player must act: spawning shared NPCs, advancing match state, picking the next map. Do not treat it as a trusted server. It cannot validate another player's state any more than that player can.

ROOMS AND SESSIONS

A room is one multiplayer session, and every session lives in exactly one room. Players create or join a room through `UFusionOnlineSubsystem`, passing an `FFusionRoomOptions` struct that describes it. We will cover the full connect-and-join flow in the next chapter; here we only need the lifecycle.

A session moves through a small set of states, listed in the `EFusionStatus` enum. Your gameplay code rarely needs the raw enum. Instead it uses the helper checks `IsConnected()`, `IsInRoom()`, and `IsJoiningOrInRoom()` to decide when networked systems are safe to read or write.



EFusionStatus: the states a session passes through.

FIG 1.3 The session lifecycle. Networking is only live in the InRoom state. Disconnected and Error are dead ends you can reach from any other state.

EFusionStatus	Reached from	Meaning
None	initial	Subsystem initialised; no Photon connection started.

Connecting	None	ConnectToPhoton in flight.
Connected	Connecting	Cloud connection up; not in a room yet.
JoiningRoom	Connected	A join/create-room action is in flight.
InRoom	JoiningRoom	Fully in a room; networking is live.
LeavingRoom	InRoom	LeaveRoom in flight; returns to Connected.
Disconnected	any non-terminal	Cloud connection dropped. Terminal.
Error	any non-terminal	Unrecoverable failure. Terminal.

Once in a room, each player has a stable player id (an `int32`) from `GetLocalPlayerId()`. That id lasts for the whole session, and it is exactly what the ownership calls (`GetOwner`, `IsOwner`, `CanModify`) compare against under the hood. When you read "the owner is player 3," this id is what that means.

COMING FROM BUILT-IN UNREAL NETCODE

If you have shipped a game with built-in netcode, a few of its concepts simply do not carry over, and trying to force them is the most common source of early confusion. `AGameModeBase` is not a server-only authority: every player constructs its own `AGameMode` locally, and Master-only flow is gated explicitly with `IsMasterClient`. The `ENetRole` family (`GetLocalRole`, `GetRemoteRole`, `GetNetOwningPlayer`) plays no part in Fusion replication. Authority is per-object and queried through `UFusionOnlineSubsystem`, not through actor roles.

Built-in Unreal	Fusion equivalent
`HasAuthority()`	`IsOwner(Actor)` or `CanModify(Actor)`
`GetNetOwningPlayer()`	`GetOwner(Actor)`
Dedicated server process	Master Client + per-object ownership
Server-only `AGameMode` logic	MasterClient or GameGlobal objects, gated on `IsMasterClient`
Replicated `UPROPERTY`	Same syntax, see Chapter 4
`SetReplicates` / `bReplicates`	Add a `UFusionActorComponent`; networking is opt-in via the component

The last row is the most important shift in day-to-day code, so it is worth stating on its own: an actor becomes networked by adding a `UFusionActorComponent`, not by being spawned on the authority or marked `bReplicates`. That component is the bridge between your actor and the Fusion client.

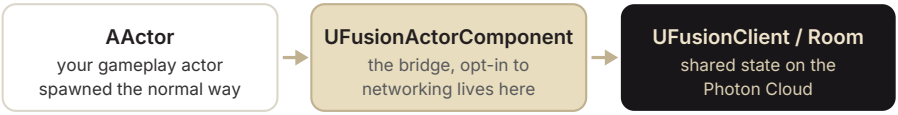


FIG 1.4 The Fusion Actor Component is the bridge: your ordinary actor opts into networking through it, and it talks to the client and the room on your behalf.

> IN BLUEPRINT **The same model in Blueprint**
>
> None of this is C++ only. A Blueprint actor becomes networked the same way, by adding the Fusion Actor Component in the Components panel, and `IsOwner`, `CanModify`, and `IsMasterClient` are all pure Blueprint nodes. The model in this chapter is the model for both languages. Only the syntax in the listings is C++.

With the model in place, we can connect to the cloud and get into a room, which is exactly what the next chapter does.

CHAPTER 02

CONNECTING TO THE PHOTON CLOUD

From an App ID to a live room: configuration, the async connect-and-join flow, regions, and the matchmaking surface, and where its limits are.

IN THIS CHAPTER

Everything in this book happens inside a room, and getting into one takes two steps: connect to the Photon Cloud, then join or create a room. This chapter walks the whole path and the settings around it.

APP ID AND PROJECT SETTINGS

Project-wide Fusion settings live in Project Settings > Fusion Settings. The two fields you cannot ship without are `AppId` and `AppVersion`. The App ID identifies your project on the Photon Cloud; get one from the Photon Dashboard and paste it into the App ID field. `AppVersion` groups players by client build, so two incompatible versions of your game can never end up in the same room. Bump it whenever the data your game sends over the network changes shape.

The same settings asset holds a few fields that belong to later chapters: `LoadMapAutomatically` shapes the map-load callback sequence (Chapter 10), and `DefaultArraySize` sets the default capacity for replicated `TArray` properties (Chapter 4). It is worth knowing they live here even before you need them.

> TIP **Treat AppVersion as part of your build**
>
> Wire it into your build pipeline so every shipped build carries a version that matches its replication format. The day you change a replicated struct and forget to bump it is the day two builds quietly corrupt each other's state in the wild.

CONNECTING

Every connection and room operation goes through `UFusionOnlineSubsystem`, a `UGameInstanceSubsystem` you reach the standard Unreal way: `GameInstance->GetSubsystem<UFusionOnlineSubsystem>()`. The subsystem owns the underlying `UFusionClient` and the life of the session.

The entry point is `ConnectToPhoton(FFusionConnectOptions, WorldContextObject)`. It returns a `UFusionConnectToPhotonAsync`, a latent action: a node that starts now and finishes later, exposing `OnSuccess` and `OnFailure` pins. In Blueprint the call appears as the node Connect To Photon; in C++ you bind the delegates and activate the node yourself.

```
UFusionConnectToPhotonAsync* Connect =
    OnlineSubsystem->ConnectToPhoton(ConnectOpts, this);

Connect->OnSuccess.AddDynamic(
    this, &UMyGameInstance::HandleConnected);
Connect->OnFailure.AddDynamic(
    this, &UMyGameInstance::HandleConnectFailed);
Connect->Activate();
```

IN BLUEPRINT **Latent nodes instead of Activate()**

In Blueprint you do not call `Activate()`. You drop the Connect To Photon node into the graph and wire its exec output to whatever happens next, with separate exec pins for On Success and On Failure. The C++ pattern above (bind, then Activate) is the manual equivalent of those pins.

JOINING OR CREATING A ROOM

Once connected, the subsystem exposes five room entry points. `CreateRoom`, `JoinOrCreateRoom`, and `JoinOrCreateRandomRoom` all take an `FFusionRoomOptions` that describes the room to create. `JoinRoom(RoomName)` and `JoinRandomRoom` join an existing room without creating one: the by-name variant needs the exact room name, and the random variant picks any room that is open and visible.

```
FFusionRoomOptions RoomOpts;
RoomOpts.RoomName     = TEXT("Match-42");
RoomOpts.MaxPlayers   = 8;
RoomOpts.InitialWorld = LobbyWorldPtr;

UFusionJoinOrCreateRoomAsync* Join =
    OnlineSubsystem->JoinOrCreateRoom(RoomOpts, this);
Join->OnSuccess.AddDynamic(
    this, &UMyGameInstance::HandleJoined);
Join->Activate();
```

For the common case of connecting and then joining in one step, the convenience call `ConnectAndJoinRoom(ConnectOptions, RoomOptions, WorldContextObject)` chains a connect and a join internally, so your code binds to a single OnSuccess/OnFailure pair.

Which entry point you choose follows from your matchmaking shape. Prefer `JoinOrCreateRoom` for invite-style flows where a known room name is shared between players, such as a friend code or a lobby id from an external service. Prefer `JoinOrCreateRandomRoom` for quick-match flows where the player does not care which room they land in.

ASYNC ACTIONS

Every connect and room call returns a subclass of `UFusionRoomActionBase`, a latent action that exposes `OnSuccess` and `OnFailure` as exec pins in Blueprint and as multicast delegates in C++. `OnFailure` carries an `EFusionActionFailureCodes` value that describes the reason.

Async class	Subsystem method(s)	Wraps
`UFusionConnectToPhotonAsync`	ConnectToPhoton	Initial cloud connection
`UFusionDisconnectFromPhotonAsync`	DisconnectFromPhoton	Full disconnect from the cloud

`UFusionJoinOrCreateRoomAsync`	JoinOrCreateRoom / ...Random	Join a matching room or create one
`UFusionCreateRoomAsync`	CreateRoom	Create a new room unconditionally
`UFusionJoinRoomAsync`	JoinRoom / JoinRandomRoom	Join an existing room
`UFusionLeaveRoomAsync`	LeaveRoom	Leave the room, stay connected
`UFusionConnectAndJoinRoomAsync`	ConnectAndJoinRoom	Connect plus join in one action

The failure codes cover the practical cases: `TimeOut`, `InvalidRegion`, `ClientAlreadyExists`, `Disconnected`, and the in-progress states `Connecting`, `JoiningRoom`, and `InRoom`, which fire when you start an action while another action of the same kind is still running. Handle that last group by disabling your UI button until the previous action's delegate returns; it is the most common avoidable failure.

CHOOSING A REGION

`FFusionConnectOptions` carries two fields. `Region` is a region code string such as "us" or "eu". `RegionSelectionMode` decides how that field is used.

Mode	Behaviour
Default	Use the first available region. The Region string is ignored.
Select	Connect to the region named in the Region string.
Best	Ping every region, pick the lowest round-trip one, and cache it.

```
FFusionConnectOptions ConnectOpts;
ConnectOpts.Region = TEXT("us");
ConnectOpts.RegionSelectionMode =
    EFusionRegionSelectionMode::Select;
```

> **TIP** **Best once, then Select**
>
> Ship with Best on the first launch so the player lands on the lowest-latency region with no UI. Save the chosen region, then switch to Select with that value on later launches to skip the ping pass and connect faster.

ROOM OPTIONS

`FFusionRoomOptions` describes the room being created or joined. Most fields are self-explanatory; two are worth a closer look.

Field	Type	Effect
RoomName	`FString`	Unique name within the App ID. Required for invite flows; auto-generated for random joins.
MaxPlayers	`uint8`	Maximum concurrent players in the room.
bIsOpen	`bool`	If false, the room rejects new joins. Set false to seal a match in progress.
bIsVisible	`bool`	If false, the room is hidden from random-join results. Invite-only rooms set this false.
EmptyTTL	`uint8` (s)	How long the room survives after the last player leaves.

InitialWorld	`TSoftObjectPtr<UWorld>`	World to load when the local player joins.

Set `InitialWorld` whenever the room creator already knows the starting map. The joining client loads that world automatically as it joins, and, usefully, the Master Client does not need to make a separate `ChangeWorld` call after OnSuccess. We will see the full map-change machinery in Chapter 10; for now, `InitialWorld` is the shortcut that covers the common case.

MATCHMAKING

`FFusionRoomOptions` keeps its surface deliberately small. It exposes `RoomName`, `MaxPlayers`, `bIsOpen`, `bIsVisible`, `EmptyTTL`, and `InitialWorld`, and nothing for custom properties or match filters. The built-in random-join entry points, `JoinRandomRoom` and `JoinOrCreateRandomRoom`, pick any room that is open and visible, with no further criteria.

When you need richer matchmaking, such as skill brackets, game modes, or region filters, drop down to the Photon Realtime client that Fusion runs on. `FFusionRoomOptions::ToCreateRoomOptions()` converts the wrapper into a `PhotonMatchmaking::CreateRoomOptions`, and from there you have the full Realtime matchmaking surface: custom room properties, the subset of those properties published to the lobby, and SQL lobby filters (the `C0` through `C9` columns) for filtered random joins. The simple struct covers the common case; the Realtime client underneath covers everything past it.

> NOTE **Plan matchmaking early**
>
> If your game needs filtered matchmaking, decide that before you build your lobby UI. The Realtime client beneath Fusion gives you custom properties and lobby filters, but wiring them in reaches deeper into your connection flow than the plain room options do, and it is far cheaper to design in than to bolt on.

LEAVING, RECONNECTING, AND STATUS

`LeaveRoom(WorldContextObject)` returns a `UFusionLeaveRoomAsync`; after OnSuccess the client is back in Connected and can call `JoinOrCreateRoom` again without re-running `ConnectToPhoton`. To tear down the connection completely, `DisconnectFromPhoton` returns the client to None.

Gameplay code reads the session state through a small set of queries rather than the raw enum:

Query	Returns
`IsConnected()`	True once the client has reached at least Connected.
`IsInRoom()`	True when fully in a room (InRoom).
`IsJoiningOrInRoom()`	True for both JoiningRoom and InRoom.
`IsMasterClient()`	True when this client is the elected master.
`PlayerCount()`	Current number of players in the room.
`GetLocalPlayerId()`	Stable int32 id of the local player.
`GetRtt()`	Round-trip time to the Photon relay, in ms.

There is exactly one global event in this area: `OnConnectedToPhoton`, a multicast that fires once the initial cloud connection is up. Per-action success and failure come through the OnSuccess/OnFailure pins on the async nodes. There is deliberately no global OnRoomJoined multicast, because the async-action pins already cover that case.

```
void UMyGameInstance::Init()
{
    Super::Init();
    OnlineSubsystem = GetSubsystem<UFusionOnlineSubsystem>();
    OnlineSubsystem->OnConnectedToPhoton.AddDynamic(
        this, &UMyGameInstance::HandleConnected);
}
```

PITFALL **Don't poll status in Tick**

It is tempting to check `Status()` every frame to drive your UI. Prefer the async-action delegates and `OnConnectedToPhoton`: they fire exactly when the transition happens, avoid race conditions, and keep your connection logic in one place instead of smeared across Tick.

You are now connected and in a room. From here on, every chapter assumes a live room, and the first thing most games do in one is spawn something.

PART TWO

REPLICATING THE WORLD

Spawning networked actors, syncing their properties, and deciding who owns what, the everyday mechanics of putting state on the wire.

CHAPTER 03

SPAWNING NETWORKED ACTORS

There is no special spawn API. You spawn the normal way, add one component, and the room adopts the actor as networked.

IN THIS CHAPTER

- The Ordinary Spawn, Adopted
- Adding the Fusion Actor Component
- Persistent vs Dynamic Actors
- Lifetime and Destruction
- Ready Events

If you expected a `FusionSpawnActor` function, you will be pleasantly disappointed. Fusion piggybacks entirely on Unreal's ordinary spawn flow. The magic is a single component, not a parallel API.

THE ORDINARY SPAWN, ADOPTED

There is no networked counterpart to `UWorld::SpawnActor`. You spawn an actor with the standard `SpawnActor` family of calls, exactly as you would in a single-player game, and Fusion adopts it as networked through a `UFusionActorComponent` on the actor.

This is the spawn-time face of the shared-authority shift from Chapter 1. Built-in Unreal replication requires the spawn to originate on the authoritative server, which then replicates the spawn outward. Fusion has no authoritative server, so the spawning client itself owns the new actor; the room replicates the actor's existence to other clients through a handshake on the component.

```
// In the actor's constructor:
FusionActorComponent =
    CreateDefaultSubobject<UFusionActorComponent>(TEXT("Fusion"));
SetReplicates(true);

// Anywhere in gameplay code, the ordinary spawn API:
AMyNetActor* Spawned = GetWorld()->SpawnActor<AMyNetActor>(
    AMyNetActor::StaticClass(), SpawnLocation, SpawnRotation);
```

IN BLUEPRINT **Spawn Actor From Class, unchanged**

In Blueprint you use the same Spawn Actor From Class node you already know. As long as the spawned class has a Fusion Actor Component and has Replicates checked, the room adopts it. There is no networked spawn node to hunt for.

ADDING THE FUSION ACTOR COMPONENT

The `UFusionActorComponent` is the bridge between an actor and the Fusion client. Add one to any actor, Blueprint or C++, that should be networked. Fusion adopts an actor only when it has both the component and `bReplicates` enabled (`SetReplicates(true)` in C++, or the Replicates checkbox in Blueprint). Adding the component does not set `bReplicates` for you, so an actor missing either one stays local-only. This is the rule that trips up every developer arriving from built-in netcode exactly once.

The component's `Ownership` property picks the ownership mode: `Transaction`, `PlayerAttached`, `Dynamic`, `MasterClient`, or `GameGlobal`. The mode controls who can take ownership at runtime and what happens to the actor when its owner disconnects. Chapter 5 treats every mode in detail; for now, know that you choose it here, at design time.

PERSISTENT VS DYNAMIC ACTORS

Networked actors come in two lifetimes, and the distinction matters for how late-joiners experience your world.

	Map-placed (persistent)	Runtime-spawned (dynamic)
Origin	Placed in the editor, saved with the map	`SpawnActor` at runtime
Late-joiners	Already present when they load the map	Re-spawned from the replicated room state
Lifetime	Until the level unloads	Until destroyed or owner leaves (per mode)
Typical ownership	MasterClient, GameGlobal	PlayerAttached, Transaction, Dynamic

A late-joining client does not miss runtime-spawned actors: the room re-spawns them from its replicated state as the client joins, so a player

who arrives ten minutes into a match still sees every crate, projectile, and avatar that should exist. This is automatic, and it is one of the quiet conveniences of letting the room hold shared state.

LIFETIME AND DESTRUCTION

Destruction is reported through `UFusionActorComponent::OnObjectDestroyed`, a multicast that carries an `EFusionObjectDestroyMode` reason. Reacting to the reason, rather than treating every destroy identically, is what separates clean teardown from double-fired cleanup.

Reason	Fires when
`Local`	The local client destroyed the actor.
`Remote`	The owner destroyed it on a remote peer; the destroy replicated in.
`MapChange`	The current map is unloading as part of a ChangeWorld.
`Shutdown`	The Fusion session is being torn down (leaving the room or disconnecting).

```
void AMyActor::HandleDestroyed(EFusionObjectDestroyMode Reason)
{
    switch (Reason)
    {
        case EFusionObjectDestroyMode::MapChange:
        case EFusionObjectDestroyMode::Shutdown:
            return;  // framework-driven; ignore
        default:
            ReleaseGameplayResources();
            break;
    }
}
```

Gate your cleanup on the reason. Ignore `MapChange` and `Shutdown` when your handler only cares about gameplay-driven despawns, otherwise the cleanup runs once when the player dies and again when they leave the match.

> PITFALL **Double-fired cleanup**
>
> The classic bug: a handler that frees a gameplay resource on every `OnObjectDestroyed`. At end of match it fires for the gameplay death and again for Shutdown, freeing twice. Filter on the reason and this whole class of bug disappears.

READY EVENTS

`OnObjectReady` on the component fires once the actor has completed its networked handshake. From that point the replicated state on the actor is safe to read and write. This is the hook you want for network-dependent initialisation, not `BeginPlay`, which runs before the handshake finishes, so replicated values read there may still be at their defaults.

```
void AMyActor::BeginPlay()
{
    Super::BeginPlay();
    FusionActorComponent->OnObjectReady.AddDynamic(
        this, &AMyActor::HandleReady);
    FusionActorComponent->OnObjectDestroyed.AddDynamic(
        this, &AMyActor::HandleDestroyed);
}
```

`UFusionOnlineSubsystem::OnObjectReady` is the subsystem-wide equivalent: it fires for every Fusion object as it becomes ready. Use it for systems that observe every newly ready object, such as a debug HUD or a global registry, without holding a per-actor binding.

PITFALL **Reading replicated state in BeginPlay**

This is the most common timing bug in a new Fusion project. `BeginPlay` runs before the Fusion handshake, so a health bar initialised there reads 100 (the default) even on a unit that joined at 30 health. Move network-dependent init to `OnObjectReady` and it simply works.

IN BLUEPRINT **Bind in the event graph**

`OnObjectReady` and `OnObjectDestroyed` are BlueprintAssignable. Bind them in the event graph exactly as you would any component delegate. The C++ AddDynamic calls above map one-to-one onto Assign nodes.

CHAPTER 04

REPLICATED PROPERTIES

Mark a UPROPERTY Replicated and Fusion syncs it, with the RepNotify, array, and batching behaviour you already half-expect.

IN THIS CHAPTER

- Marking a Property Replicated
- RepNotify and OnRep Handlers
- Networked Arrays
- Controlling When State Goes Out
- Replication Filters and Smoothing

The headline of this chapter is reassuring: replicated properties in Fusion use the same `UPROPERTY(Replicated)` reflection you already know. The details are where Fusion earns its keep.

MARKING A PROPERTY REPLICATED

Mark a property `Replicated` the way the Unreal documentation describes, and Fusion picks it up automatically. The opt-in is at the actor level, not the property level: adding a `UFusionActorComponent` to the actor is what brings all of its `Replicated` properties into Fusion's sync.

```
UPROPERTY(Replicated)
float Health = 100.f;
```

The types Fusion can put on the wire come from `EFusionDataTypes`: the primitive integers and floats (`Byte`, `Int`, `Float`, `Double`, `Int64`, and the unsigned and short variants), `bool`, the common math structs (`FVector`, `FRotator`, `FQuat`), `FName`, `FString`, object, class, and actor references, custom `USTRUCT`s, and `TArray` containers of any supported type. If you can express it as one of those, Fusion can replicate it.

REPNOTIFY AND ONREP HANDLERS

`UPROPERTY(ReplicatedUsing = OnRep_Foo)` works as it does in stock Unreal. Fusion resolves the RepNotify function for each property when it builds the actor's replication descriptor, and invokes it whenever an incoming replicated value differs from the local one.

```
UPROPERTY(ReplicatedUsing = OnRep_Health)
float Health = 100.f;

UFUNCTION()
void OnRep_Health(float OldHealth);
```

There is one behaviour worth committing to memory, because it makes a

class of race conditions impossible. Fusion batches its notifications per packet: when a packet updates several replicated properties, all the changed values are written first, and then every affected `OnRep_*` fires in a single pass. That means an OnRep handler can safely read other freshly-replicated properties on the same actor; they are already up to date by the time your handler runs.

> NOTE **Why batching matters**
>
> In some networking stacks, reading a sibling property inside an OnRep is a gamble, because it may not have arrived yet. Fusion's write-all-then-notify ordering removes that gamble: inside `OnRep_Health` you can read `LastHitDirection` from the same packet and trust it.

NETWORKED ARRAYS

Fusion replicates `TArray` properties through a fixed-capacity slot that is pre-sized when the descriptor is built. Declare the capacity with the `FusionArraySize` metadata key. The hard cap is 64 (`FusionMeta::FusionArrayMaxSize`); arrays without an explicit `FusionArraySize` fall back to the project-wide `DefaultArraySize` from settings.

```
USTRUCT()
struct FInventory
{
    GENERATED_BODY()
    FUSION_BODY()

    UPROPERTY(Replicated, meta = (FusionArraySize = "32"))
    TArray<FInventoryItem> Items;
};
```

For richer arrays, Fusion ships `FFusionNetworkedArray`. It supports delta replication (sending only the elements that changed) with `PreRemove`,

`PostAdd`, and `PostChange` callbacks, in the spirit of Unreal's `FFastArraySerializer`. Declare your container as a `USTRUCT` inheriting `FFusionNetworkedArray` and wire callbacks through `FFusionArrayHooks`. Reach for it when you need to react to individual element changes rather than re-scan the whole array on every update.

> PITFALL **The 64-element ceiling**
>
> Networked arrays are fixed-capacity, capped at 64. If your inventory or scoreboard can exceed that, do not replicate it as one giant array. Page it, or move the long tail into a MasterClient-owned structure. Sizing `FusionArraySize` generously also costs bandwidth, so size it to the realistic maximum, not the theoretical one.

CONTROLLING WHEN STATE GOES OUT

By default Fusion copies your live `UPROPERTY` values into the outgoing networked state every tick. `UFusionActorComponent::LocalStateCopyMode` controls this. `Auto` (the default) copies every tick; `Manual` copies only on the tick after you explicitly call `CopyLocalStateNextFrame()`.

Manual mode exists for the case where gameplay code writes intermediate values that should never stream out. A multi-step state transition touching several replicated properties is the canonical example: with `Manual`, the intermediate frames never hit the wire, only the final, consistent state does.

```
// In the constructor:
FusionActorComponent->LocalStateCopyMode =
    ELocalStateCopyMode::Manual;

// In gameplay code, after a multi-write transaction:
ApplyDamageAndUpdateHitState();  // writes several properties
FusionActorComponent->CopyLocalStateNextFrame();
```

For pausing replication entirely (while a cinematic plays out locally, say),

`bAutomaticallySendUpdates` and the `ToggleNetworkSend(bool)` method are the coarser switches. They suppress all outgoing replication on the actor until you re-enable them.

REPLICATION FILTERS AND SMOOTHING

Fusion's primary bandwidth filter is area-of-interest, driven by a `UFusionSpatialHashInterestComponent`. Because it is a substantial topic in its own right, it has its own chapter, Chapter 7. For now it is enough to know that interest filtering is how the room avoids sending every object to every client.

Smoothing splits along the physics line. For physics-driven roots, any actor whose root component simulates physics, smoothing is handled automatically by the Forecast pipeline (Chapter 6). For non-physics roots, plain `AActor` transforms or CharacterMovement-driven pawns, Fusion does not ship a public smoothing API on the Unreal plugin. Remote transforms reflect the values that arrive in the next packet, without interpolation.

> **TIP** **Smooth non-physics remotes in Tick**
>
> When a non-physics remote actor looks steppy between packets, smooth it in gameplay code: store the replicated transform as a target and interpolate the visual mesh toward it each Tick. A few lines of `FMath::VInterpTo` on the remote side buys a lot of visual polish for non-physics movement.

CHAPTER 05

OWNERSHIP

Who writes an object's state, what happens when they leave, and how authority moves at runtime. This is the most consequential design choice you make per actor.

IN THIS CHAPTER

Ownership is the beating heart of shared authority. The mode you pick on each actor decides who may write its state, what becomes of it on a disconnect, and whether authority can change hands while the game runs.

THE FIVE OWNERSHIP MODES

`EFusionObjectOwnerFlags` has five values, each a different rule for who can own an object and what happens to it when the owner disconnects. You pick the mode at design time on the `UFusionActorComponent`; it is the single most consequential property on the component.

`Transaction`	take / release; most objects
`PlayerAttached`	destroyed when owner leaves
`Dynamic`	frequent owner changes; physics
`MasterClient`	always the current master
`GameGlobal`	lifetime is yours to manage

FIG 5.1 The five ownership modes and the one-line rule that defines each.

EFusionObjectOwnerFlags, from the enum's own comments.

Mode	Behaviour
`Transaction`	Take-and-release: the owner releases via `SetWantsOwner(false)`, then another client may claim. Used for most objects.
`PlayerAttached`	Transaction semantics, plus: if a player is the owner when they leave, the object is destroyed.

`Dynamic`	Allows ownership to be overridden on every client; suited to objects that change owner frequently, like physics props.
`MasterClient`	Always owned by the current Master Client; re-targets automatically on master change.
`GameGlobal`	Lifetime and ownership are entirely the application's responsibility.

The default mapping is a good starting point: `PlayerAttached` for player avatars and per-player inventory; `Transaction` for shared pickups a player explicitly grabs and drops; `Dynamic` for physics objects and projectiles that change hands constantly; `MasterClient` for singletons and room-wide systems; and `GameGlobal` only when your project genuinely needs to manage lifetime itself.

SETTING OWNERSHIP AT DESIGN TIME

The `Ownership` property is read once, when the actor is attached as a Fusion source. Set it in the C++ constructor or the Blueprint defaults; it is not designed to flip at runtime.

```
AMyNetActor::AMyNetActor()
{
    FusionActorComponent =
        CreateDefaultSubobject<UFusionActorComponent>(
            TEXT("Fusion"));
    FusionActorComponent->Ownership =
        EFusionObjectOwnerFlags::PlayerAttached;
    SetReplicates(true);
}
```

Changing the `Ownership` property after the actor is attached is not the normal transfer path and is not supported as a runtime API. Runtime ownership changes go through `SetWantsOwner` instead, which we get to next.

> IN BLUEPRINT **Set it in Class Defaults**
>
> On a Blueprint actor, select the Fusion Actor Component and set Ownership in the Details panel. It is a design-time default in both languages; there is no runtime setter for the mode itself.

REQUESTING OWNERSHIP AT RUNTIME

`UFusionOnlineSubsystem::SetWantsOwner(Actor, bWantsOwner)` is the runtime entry point. Call it with `true` to claim ownership of an actor, or `false` to release. The call resolves through the ownership-mode rules: a `Transaction` actor responds to the claim, while a `PlayerAttached` actor does not.

```
OnlineSubsystem->SetWantsOwner(Actor, true);  // claim
OnlineSubsystem->SetWantsOwner(Actor, false); // release
```

The pure inspectors answer the runtime question "who owns this, and may I write to it?": `HasOwner`, `GetOwner`, `GetResolvedOwner`, `IsOwner`, and `CanModify`. Use `IsOwner` and `CanModify` as the predicates that gate every write to replicated state, the same predicates introduced in Chapter 1, now with the transfer machinery behind them.

AUTO DYNAMIC OWNERSHIP RANGE

`AutoDynamicOwnershipRange` (in centimetres) is an optional helper that automates `SetWantsOwner` calls based on distance. It is a separate feature from the `Dynamic` ownership mode: the two often pair, but neither requires the other.

```
FusionActorComponent->Ownership =
    EFusionObjectOwnerFlags::Dynamic;
FusionActorComponent->AutoDynamicOwnershipRange = 500.0; // cm
```

When the value is non-zero, the component enables its tick and, each

frame, measures the distance from the actor to the local player pawn, calling `SetWantsOwner(GetOwner(), distance < AutoDynamicOwnershipRange)`. In plain terms: the nearest player tends to take ownership of nearby objects automatically. A value of 0 disables the helper and leaves ownership entirely under explicit `SetWantsOwner` control.

> TIP **Pair it with Dynamic for physics props**
>
> A barrel you want whichever player is closest to simulate, handed off smoothly as players move past: that is the textbook case. Dynamic mode plus a sensible `AutoDynamicOwnershipRange` gives you smooth, hands-off ownership handoff for physics clutter.

WHAT HAPPENS ON DISCONNECT

Each mode answers the disconnect question differently, and getting this right is what keeps a room consistent when players come and go.

`PlayerAttached` objects are destroyed when the owning player leaves: this covers avatars and player-attached inventory, none of which should outlive their player. `MasterClient` objects auto-transfer to the newly promoted master, with no application code. `Transaction` and `Dynamic` objects are not destroyed; they simply become unowned, and what happens next is up to your transaction logic on top of `SetWantsOwner`. `GameGlobal` lifetime is entirely yours.

> NOTE **Persisting across a rejoin**
>
> If a value must survive a player leaving and coming back, do not store it on a `PlayerAttached` object; it will be destroyed. Put it on `APlayerState` (preserved via Unreal's InactivePlayerArray, see Chapter 9) or on a `MasterClient`-owned actor that outlives any single player.

MASTER CLIENT OWNERSHIP AND AUTO-PROMOTION

`MasterClient` ownership binds an object's owner to whoever is the current Master Client. The resolved owner is recomputed on every master change, so `IsOwner(Actor)` and `IsMasterClient()` always agree for these actors. When the current master leaves, Photon promotes a new one, every `MasterClient`-owned object re-targets in the same tick, and `OnOwnerChanged` fires on each affected component, all without a line of application code.

> PITFALL **Don't SetWantsOwner a MasterClient object**
>
> The rule for `MasterClient` ownership is "whoever is master right now," not "whoever asked." Claim requests do not change the resolved owner. If you need ownership that any client can request and release, use `Transaction` or `Dynamic` instead. Reaching for `SetWantsOwner` on a `MasterClient` object silently does nothing.

OWNERSHIP-CHANGE EVENTS

Two events let your gameplay react to authority moving. `OnOwnerChanged` fires whenever the resolved owner changes, including local-to-remote and remote-to-local transitions. `OnOwnerWasGiven` is the narrower companion: it fires only when the local client has just become the owner, the right hook for "I just took authority, start driving this actor." Use it to spin up input handling, AI behaviour, or any per-tick logic that only the owner should run.

```
void AMyActor::BeginPlay()
{
    Super::BeginPlay();
    FusionActorComponent->OnOwnerChanged.AddDynamic(
        this, &AMyActor::HandleOwnerChanged);
    FusionActorComponent->OnOwnerWasGiven.AddDynamic(
        this, &AMyActor::HandleBecameOwner);
}
```

Pairing these events with the predicates from earlier in the chapter gives you a complete ownership story: `OnOwnerWasGiven` to start owning, `OnOwnerChanged` to react to any handoff, and `CanModify` to gate the writes in between.

PART THREE

FEEL AND SCALE

Smooth physics across the network with Forecast, send only what each player needs with interest management, and target RPCs explicitly under shared authority.

CHAPTER 06

PHYSICS PREDICTION WITH FORECAST

Physics bodies cannot tolerate snapping to a value every few ticks. Forecast predicts them locally and corrects gently, so motion stays smooth at render rate.

IN THIS CHAPTER

Plain replicated properties update discretely: each packet snaps the remote value to the authoritative one. For a health bar that is fine. For a rolling barrel it looks like a teleport-and-rubber-band. Forecast is Fusion's answer.

HOW FORECAST WORKS

Physics bodies move continuously under forces, so a snap every few network ticks reads as a visual glitch. Forecast replaces the snap with **extrapolation plus continuous correction**. The owner periodically sends an `FFusionBodyState` snapshot: position, rotation as a quaternion, linear and angular velocity, and the frame it was captured on. Between snapshots, remote clients extrapolate forward from the most recent one, applying gravity and integrating velocity, and nudge the local Chaos body toward the predicted target each physics tick.

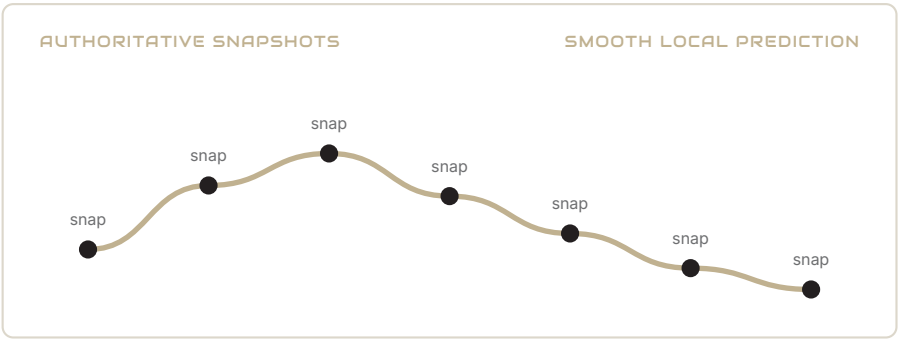


FIG 6.1 Forecast carries a smooth predicted path between sparse authoritative snapshots, then corrects toward each new snapshot as it arrives.

The extrapolation window is bounded by `MaxExtrapolationTime`. If a new snapshot does not arrive within that window, the body freezes at its last predicted state rather than drifting off into nonsense. The result is motion that stays smooth at render rate even when network updates arrive at only 20-30 Hz, which, on real networks, they often do.

ENABLING AND DISABLING FORECAST

`bForecastPhysicsEnabled` on the component is the master switch, and it defaults to true. When false, the actor falls back to plain transform replication, the same path a non-physics actor takes, and the smoothing pipeline is bypassed.

Forecast also requires that the body is actually simulating physics. A primitive with Simulate Physics off is not forecast, because there is nothing to predict. Re-enable physics on it to bring it back.

```
AMyKinematicActor::AMyKinematicActor()
{
    FusionActorComponent =
        CreateDefaultSubobject<UFusionActorComponent>(
            TEXT("Fusion"));
    // Fall back to plain transform replication:
    FusionActorComponent->bForecastPhysicsEnabled = false;
}
```

Leave Forecast on for any rigid body you want smoothed across the network. Turn it off for kinematic actors that gameplay code drives directly through ownership transfers: those want plain transform replication, not prediction fighting your scripted motion.

WHICH ACTORS ARE ELIGIBLE

Forecast engages for each primitive component on the actor that is actually simulating physics, checked once when the actor is attached. A `UPrimitiveComponent` with Simulate Physics enabled is forecast; one with physics off is not. What decides eligibility is whether a primitive is simulating, not where it sits in the component hierarchy.

```
APhysicsCrate::APhysicsCrate()
{
    Mesh = CreateDefaultSubobject<UStaticMeshComponent>(
        TEXT("Mesh");
    RootComponent = Mesh;
    Mesh->SetSimulatePhysics(true);

    FusionActorComponent =
        CreateDefaultSubobject<UFusionActorComponent>(
            TEXT("Fusion"));
    // UFusionPhysicsReplicationComponent attaches automatically.
}
```

PITFALL **A prop that won't smooth**

If a rigid body refuses to smooth across the network, check that Simulate Physics is actually enabled on the primitive. A primitive with physics off is not forecast, no matter how the Fusion settings look. Forecast follows the simulating body, so make sure the body is simulating.

ERROR CORRECTION MODES

`EFusionPhysicsCorrection` selects how the remote body is nudged toward the predicted target each physics tick.

Mode	How it corrects	Use for
`Velocity` (default)	Adds a corrective velocity toward the target	Most rigid bodies, smooth with no overshoot
`PositionRotation`	Lerps the transform toward the target	Bodies where transform fidelity beats realism

`SpringDamping`	Applies a spring-damper force/torque	Snappier convergence, accepts some overshoot

```
FusionActorComponent->ErrorCorrectionType =
    EFusionPhysicsCorrection::SpringDamping;
```

`Velocity` is the recommended starting point and the project-wide default. Switch to `SpringDamping` only when overshoot is acceptable in exchange for snappier convergence, typically vehicles or fast props where lag-induced visual stretch is worse than a small bounce on arrival.

TUNING KNOBS

Every tuning knob lives in two places. The project-wide default sits on the Fusion settings and is edited under Project Settings; the per-actor override sits on the component in the Fusion | Forecast Physics category, paired with a matching `Override*` toggle that decides whether the per-actor value or the project default applies.

Knob	Default	Raise to / Lower to
`LinearVelCorrectionMul`	4.0	Converge faster / reduce jitter on noisy nets
`AngularVelCorrectionMul`	1.0	Converge rotation faster / reduce rotational jitter
`PositionCorrectionLerp`	0.15	Snappier / smoother position lerp
`RotationCorrectionLerp`	0.15	Snappier / smoother rotation lerp
`Spring`	75	Stiffer, faster (overshoot) / softer, lag-tolerant

`Damper`	2	More damping (less overshoot) / springier
`MaxExtrapolationTime`	0.3 s	Longer extrapolation under lag / freeze sooner

> **TIP** **Tune at project scope first**
>
> Set the defaults that suit most of your props once, at project scope. Reserve per-actor overrides for the genuine exceptions: a heavy boss, a delicate vehicle. Scattering overrides across dozens of actor classes is how a tuning pass becomes unmaintainable.

GRAVITY FORECAST

`EFusionGravityForecast` decides whether gravity is folded into the extrapolation on remote clients. `Apply` always folds it in; `None` never does; `Auto` (the project default) applies gravity only when the body has `bEnableGravity` set. Choose `None` for zero-gravity environments or buoyant objects; choose `Apply` for predictable falling props like crates and debris; stick with `Auto` for mixed scenes where the same object can be in flight, grounded, or floating depending on context.

COLLISIONS AND TELEPORTS

On a hit, Fusion registers the collision and pauses correction for `ImpactStartCorrectionTime` seconds, trusting the local physics solver to handle the response, then ramps correction back over `ImpactCorrectionTimeComplete` seconds. This is what avoids the classic rubber-band-on-collision artifact, where remote prediction fights the local hit response.

For intentional snaps, a `TeleportKey` on the body state forces an instant transform change on remotes, skipping smoothing. You rarely manage it

by hand: it is bumped automatically when a move uses `ETeleportType::TeleportPhysics`, so calling `SetActorLocation(..., ETeleportType::TeleportPhysics)` on the owner is enough.

> PITFALL **A teleport that slides**
>
> If a teleport on the owner produces a smooth slide on remotes instead of an instant snap, the move almost certainly used the default `ETeleportType::None`, which Fusion reads as a continuous move and smooths like any other. Pass `ETeleportType::TeleportPhysics` to get the snap.

CHAOS INTEGRATION

Forecast integrates at the solver level, not the actor level. Fusion registers a physics-replication factory in the engine's replication-factory slot; the factory returns Fusion's implementation, which ticks each physics step and applies forces directly on Chaos bodies. That low-level integration is exactly why Forecast can run at physics frequency rather than tick frequency.

> NOTE **Chaos is required**
>
> Forecast applies forces directly on Chaos bodies, so a PhysX-only build cannot use it. Unreal 5.x uses Chaos by default, so this is rarely an issue, but a project that has deliberately switched back to PhysX gives up Forecast.

CHAPTER 07

INTERESTMANAGEMENT

Don't send every object to every client. Interest keys filter what each player receives, saving bandwidth and hiding state from players who shouldn't see it.

IN THIS CHAPTER

Interest management answers a question every networked game eventually asks: why am I paying to tell a player about an object on the far side of the map they cannot possibly see? Fusion's answer is a 64-bit key and a subscription.

HOW IT WORKS

Interest management has two goals. The obvious one is bandwidth: do not send state to players who cannot see it. The subtler one is integrity: hide state from players outside the relevant region, so that, for example, a stealth-game opponent cannot learn a hidden player's position by sniffing replicated transforms.

The mechanism is simple to state. Every replicated object carries a 64-bit interest key in the room. Every client maintains a set of subscribed keys. The room only delivers property updates and RPCs for objects whose key matches one of the client's subscriptions. Two halves drive the system: the owner stamps keys on the objects it owns (the publisher side), and the client subscribes to the keys it wants (the subscriber side).

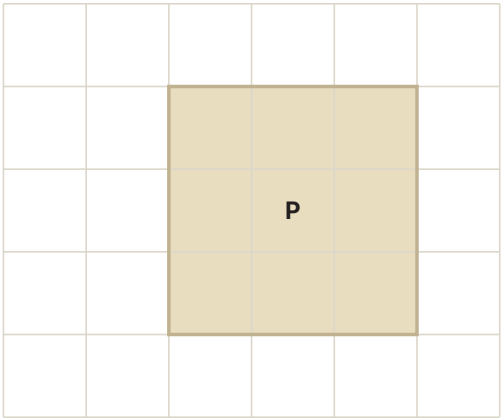
GLOBAL INTEREST

The default key is 0, which means global. Objects with the global key are delivered to every client regardless of subscriptions; they bypass the filter entirely. An object falls back to global in two cases: no interest component is present, or one is present but disabled. This fallback is deliberate: a project that has not yet adopted interest management still works, because every object is simply global.

Global is the right choice for small rooms (a four-player co-op session does not benefit from filtering), for lobby state every player must see, and for singletons such as the `AGameStateBase` that every client needs to observe.

SPATIAL-HASH AREA OF INTEREST

`UFusionSpatialHashInterestComponent` is the pre-built area-of-interest strategy. Add it to a `PlayerController` or `Pawn` and it does both halves of the work in one tick: it subscribes the local client to the cells the player is near, and it stamps the local client's owned root actors with their cell key.



subscribed cells (SubscribeRadius 1,1), the room only sends what is near

FIG 7.1 The spatial hash. The player subscribes to the cells around them; the room sends only objects whose cell key falls inside that neighbourhood.

The 64-bit key packs three 21-bit cell coordinates. Because the encoding is deterministic, two clients computing the same world position produce the same key, which is exactly what lets the room match publishers to subscribers cheaply, without any shared lookup table.

Configuring the spatial-hash component.

Property	Default	Effect
`bEnabled`	true	When false, owned actors fall back to global interest.
`CellSize`	3200 cm	Edge length of a cell. Smaller = finer filtering, more room memory.

`WorldHalfExtent`	1024 cells	Half-extent of the grid. World must fit inside it.
`SubscribeRadius`	(1,1,1)	Neighbour cells per axis. (1,1,1) is a 3×3×3 = 27-cell volume.

Shape `SubscribeRadius` to your game. The default 27-cell cube suits first- and third-person 3D play. A top-down or isometric game, where vertical visibility is irrelevant, uses something like (2, 0, 2), a flat slab that widens horizontal interest without subscribing to cells above and below the player.

> PITFALL **Every client must agree on the grid**
>
> `CellSize` and `WorldHalfExtent` must be identical on every client in the room, so a given world position maps to the same key everywhere. The server has no cell geometry of its own; it just indexes objects by key value and delivers each one to any subscriber whose keys include it. If clients disagree on these values, their keys do not line up, and objects show up as missing updates or ghosts. Ship the same configuration to every client.

CUSTOM INTEREST STRATEGIES

The spatial hash is just one strategy. Any client-side system can drive the publisher half by calling `UFusionClient::UpdateOwnedActorAreaInterestKeys` each frame with a function that computes a key per owned actor. The function can key on whatever your game needs: team affiliation, room region, gameplay state.

```
void UMyInterestSubsystem::Tick(float DeltaTime)
{
    UFusionClient* Client = OnlineSubsystem->GetClient();
    Client->UpdateOwnedActorAreaInterestKeys(
        [this](const AActor* Actor) -> uint64
        {
            return ComputeKeyForActor(Actor);
        });
}
```

For the subscription half, the pattern is to write a replacement component modelled on the spatial-hash component's tick logic, driving the same subscribe/unsubscribe surface on the client. Whatever scheme you choose, every client must produce the same 64-bit key for the same logical region. The server only matches keys by value and never interprets their meaning, so the agreement that matters is between clients: a client that disagrees will see ghost objects or miss updates.

RELEVANCY, DORMANCY, AND ENTER/EXIT EVENTS

Fusion interest keys are independent of Unreal's `bAlwaysRelevant`, `NetCullDistanceSquared`, and `NetDormancy`; those belong to Unreal's native path and have no effect on Fusion traffic. The Fusion equivalent of always relevant is the global key. The Fusion equivalent of dormancy is simply that an object whose properties never change generates no traffic; there is no separate dormancy state machine to manage. For a coarse pause-everything switch, use `ToggleNetworkSend(false)` from Chapter 4.

Two events track the local client's interest state for an actor. `OnInterestEnter` fires when the client starts receiving updates for an object; `OnInterestExit` fires when it stops. A remote object stays alive on the local client even when out of interest, it just stops receiving updates, so it can enter and exit interest many times in its life without being destroyed or re-spawned.

```
void AMyActor::BeginPlay()
{
    Super::BeginPlay();
    FusionActorComponent->OnInterestEnter.AddDynamic(
        this, &AMyActor::FadeIn);
    FusionActorComponent->OnInterestExit.AddDynamic(
        this, &AMyActor::FadeOut);
}
```

TIP **Fade, don't pop**

Interest enter/exit is the perfect hook for fade-in/fade-out visuals and for suspending per-tick work on objects you are no longer receiving. Fading on these events turns the hard pop of an object leaving interest into something the player never notices.

CHAPTER 08

REMOTE PROCEDURE CALLS

Without a canonical server, an RPC has to say where it is going. Fusion makes targets explicit, and that turns out to be clearer than the built-in three.

IN THIS CHAPTER

- Fusion RPC Targets
- Defining an RPC in C++
- Reliability and Ordering
- Behaviour Under Owner and Master Changes
- Built-in Markup and Argument Types
- Global RPCs

Built-in Unreal gives you Server, Client, and NetMulticast. Under shared authority there is no "the server" to be the implicit recipient, so Fusion makes the destination part of the call. Once you adjust, it reads more honestly than the markup it replaces.

FUSION RPC TARGETS

Every Fusion RPC specifies a target, because shared authority has no built-in "server" recipient: the call has to say what it means by destination.

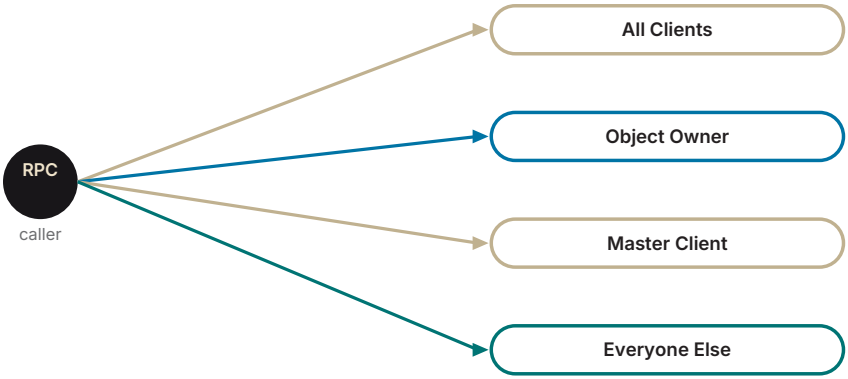


FIG 8.1 An RPC names its destination explicitly: all clients, the object's owner, the current Master Client, or everyone except the caller.

The four Fusion RPC targets.

Target	Receivers
`TargetAllClients`	Every peer in the room, including the caller
`TargetObjectOwner`	The resolved owner of the source actor
`TargetMasterClient`	The current Master Client (which can change mid-game)
`TargetEveryoneElse`	Every peer except the caller

Targets must be Fusion-networked actors, or subobjects of

Fusion-networked actors with replication enabled. An RPC is a message about a networked object, so the object is what routes it.

DEFINING AN RPC IN C++

C++ RPCs use the `SEND_FUSIONRPC` macro. Mark the call-site function with `SEND_FUSIONRPC(target)`; Fusion's build-tool plugin generates the body that serialises the parameters and sends them, and you implement the matching `_Receive` function that runs on the target.

Two pieces of boilerplate make the generation work. First, include the generated Fusion header above the usual Unreal generated header:

```
#include "MyUnrealClass.fusion.h"
#include "MyUnrealClass.generated.h"
```

Second, add `FUSION_BODY()` to the class body, just below `GENERATED_BODY()`:

```
GENERATED_BODY()
FUSION_BODY();
```

With that in place, the RPC itself is two declarations. The `_Receive` suffix marks the function that runs on the selected target; it behaves like Unreal's `_Implementation`, and it is yours to write. The send side is generated for you.

```
SEND_FUSIONRPC(TargetEveryoneElse)
void SendValue(float Amount);
void SendValue_Receive(float Amount);
```

IN BLUEPRINT **The Custom Fusion RPC node**

In Blueprint, create a custom event with your parameters, then drop a Custom Fusion RPC node and select that event in its dropdown as the target. Invoke the node anywhere in the graph. If you change the event's parameters, press Refresh Pins to regenerate the binding; the node builds an internal serialiser named after the event plus a `_Call` suffix.

RELIABILITY AND ORDERING

Fusion RPCs are reliable and in-order per (caller, target object). Two RPCs sent from the same caller to the same target object arrive in the order they were sent, and neither is silently dropped. That guarantee is per pair, not global: ordering is not promised across different callers or different target objects.

RPCs are not replayed for late joiners. A client that joins after an RPC fired never sees it. Anything that must survive a join belongs in a replicated property on a networked actor, not in an RPC; the persistent path is Chapter 4, and this distinction is one of the most important in the book.

PITFALL **Using an RPC for state that must persist**

The trap: announce "the door is now open" with a `TargetAllClients` RPC, then a player joins and sees a closed door forever. RPCs are events, not state. Replicate a `bDoorOpen` property and let the OnRep open the door; send the RPC only for the one-shot sound or particle that a late-joiner genuinely should miss.

BEHAVIOUR UNDER OWNER AND MASTER CHANGES

`TargetObjectOwner` and `TargetMasterClient` both re-resolve the recipient on the receiving side. For the master target in particular, an in-flight migration, the original master left while the RPC was on the wire, is delivered to the new host rather than bounced. Your code does not need to retry the call across master changes; the routing handles it.

BUILT-IN MARKUP AND ARGUMENT TYPES

Existing projects need not rewrite everything at once. Fusion routes Unreal's built-in NetMulticast markup through its net driver, forwarding each call into the subsystem, so stock RPC markup still compiles and runs. That said, the `SEND_FUSIONRPC` path is the recommended one for shared-authority code, because it makes the target explicit instead of leaving it implicit in markup that assumed a server.

The supported argument types mirror the replicated wire types from Chapter 4: the integer and float family, `bool`, `FVector`/`FRotator`/`FQuat`, `FName`, `FString`, class/object/actor references, `TArray` of any supported type, and custom `USTRUCT`s. Object, weak-object, and soft-object references require the encoded object to be either Fusion-networked or part of the loaded level; level objects are encoded as deterministic string paths so every client resolves them identically.

GLOBAL RPCS

Deriving a class from `UGameInstance` and defining RPCs there lets you send to everyone connected to the same session, regardless of which world or objects they currently have. This works in both C++ and Blueprint. Because the `UGameInstance` is guaranteed to exist on every connected client, it is the one RPC target that is always resolvable, which makes it the natural home for truly global, world-independent messages. We return to the networked GameInstance in Chapter 9.

PART FOUR

THE GAME FRAMEWORK

How GameMode, GameState, PlayerState, and the GameInstance behave without a server, and how to change maps and stream levels without breaking the room.

CHAPTER 09

GAME CLASSES UNDER FUSION

GameMode, GameState, PlayerState, and GameInstance all still exist. Who runs them, and when, changes once there is no single server.

IN THIS CHAPTER

- GameMode in Fusion
- GameState and Master-Client Authority
- MatchState and the Bogus-Initial-State Race
- PlayerState and the InactivePlayerArray
- The Networked GameInstance
- The Online Subsystem, PlayerController, and Pawn

The Unreal game framework assumes a server. Fusion does not have one. The classes survive intact, but the rule "only the server runs this" becomes "only the Master Client runs this," and it is now your job to enforce it.

GAMEMODE IN FUSION

Every client constructs its own `AGameMode` locally. Fusion is distributed-authority, not server-only, so there is no single peer that runs the GameMode "for real" while everyone else proxies it. Each peer's `AGameMode` is a full Unreal instance, and all the standard hooks (`HandleMatchHasStarted`, `HandleStartingNewPlayer`, `RestartGame`) fire locally on every peer.

This is the opposite of stock Unreal, where only the dedicated server runs the GameMode. The consequence is direct: any match-flow logic you do not gate runs once per client, and its effects multiply by player count. Gate it on `IsMasterClient()`.

```
void AMyGameMode::RestartGame()
{
    if (!OnlineSubsystem->IsMasterClient()) return;
    Super::RestartGame();
}
```

> PITFALL **Match logic that multiplies by player count**
>
> Leave `RestartGame`, `HandleMatchHasStarted`, or a score-award hook ungated and, in an eight-player room, it runs eight times. This is the single most common Fusion bug for developers arriving from dedicated-server projects. When in doubt, gate match flow on `IsMasterClient`.

GAMESTATE AND MASTER-CLIENT AUTHORITY

The GameState is networked automatically. Fusion auto-attaches a `UFusionActorComponent` to any `AGameStateBase` with MasterClient ownership, so only the Master Client's writes propagate, and ownership auto-transfers to the new master on migration with no application code.

Fusion also patches the replicated world-time fields on the GameState each tick from its own network time, so you can use a stock `AGameStateBase` without a Fusion-specific subclass and still get synchronised time.

MATCHSTATE AND THE BOGUS-INITIAL-STATE RACE

There is a subtle race worth understanding. Each client's local `AGameMode` sets its own initial MatchState during map load, before Fusion has replicated anything. So on a non-master client the value is briefly wrong: the local GameMode thinks the match just started fresh, while the room is already mid-match.

Once Fusion's per-tick GameState update runs, non-master clients mirror the authoritative MatchState onto their local GameMode directly. That mirror deliberately bypasses `AGameMode::SetMatchState`, precisely to avoid re-firing server-only handlers like `HandleMatchHasStarted` on every client. When you handle a match-state change, guard against the bogus initial value:

```
void AMyGameState::OnRep_MatchState(FName OldMatchState)
{
    if (OldMatchState == MatchState) return;  // initial value
    HandleMatchStateChanged(OldMatchState, MatchState);
}
```

PLAYERSTATE AND THE INACTIVEPLAYERARRAY

Fusion auto-attaches a PlayerAttached component to every `APlayerState`. The PlayerState's lifetime follows its Fusion player: created on join, destroyed on leave. Disconnected players that a client has observed land in the standard `AGameMode::InactivePlayerArray`, exactly as in stock Unreal networking, so their carryover state (score, team, any replicated property) survives a rejoin for as long as the PlayerState remains in that array.

> NOTE **Store travel-critical data on PlayerState**
>
> Because PlayerState survives in the InactivePlayerArray and pawns and controllers do not, anything that must outlive a disconnect or a map change (score, team assignment, persistent flags) belongs on `APlayerState`. We return to this in Chapter 10.

THE NETWORKED GAMEINSTANCE

`UGameInstance` is networked under Fusion, and its replicated properties persist for as long as the `UFusionClient` is alive. Because it survives `ChangeWorld`, outliving any individual map, it is the right place to store global state that must carry across maps without re-spawning a networked actor or rebuilding the GameState.

For the same reason it is a reliable place to fire global RPCs to all connected clients regardless of which world each peer is in: the GameInstance is guaranteed to exist on every connected client, so the target is always resolvable. This is the mechanism behind the global RPCs introduced at the end of Chapter 8.

THE ONLINE SUBSYSTEM, PLAYER CONTROLLER, AND PAWN

`UFusionOnlineSubsystem` is reached the standard way and owns the connection, the room flow, the status queries, and the map-load

delegates for the whole process.

```
UFusionOnlineSubsystem* Subsystem =
    GetGameInstance()->GetSubsystem<UFusionOnlineSubsystem>();
```

`APlayerController` and `APawn`, by contrast, use standard Unreal behaviour. Each client runs the normal `HandleStartingNewPlayer` / `RestartPlayer` chain locally and spawns its own controller and pawn, with no Master-Client gating needed in this path, and possession follows the usual Possess flow. The controller and pawn are local concerns; it is the room-wide singletons (GameState, match flow, map) that need the Master gate.

CHAPTER 10

MAP CHANGES AND TRAVEL

ServerTravel has nothing to anchor to under Fusion. The Master Client moves the whole room with ChangeWorld; individual peers branch off with FusionClientTravel.

IN THIS CHAPTER

- Why Map Changes Are Different
- ChangeWorld: Moving the Whole Room
- ClientTravel: Leaving on Your Own
- The Map-Load Callback Sequence
- StopFusionSession and RestartGame
- Seamless vs Non-Seamless, and Preserving PlayerState

Map changes are where built-in habits do the most damage. There is no server, so ServerTravel is meaningless, and reaching for it, or for raw SeamlessTravel, is how you put Fusion into a broken state. This chapter is the safe path.

WHY MAP CHANGES ARE DIFFERENT

Under Fusion there is no canonical server, so `ServerTravel` has nothing to anchor on. Map authority anchors to the Master Client instead: it decides the next world for the room, and every other peer either follows that decision (room-wide travel, replicated through Fusion) or branches off locally with a client travel (single-peer, not replicated).

The rule is short: never call `ServerTravel` under Fusion. Prefer the Fusion-aware entry points on `UFusionOnlineSubsystem` and `UFusionHelpers` over `UGameplayStatics::OpenLevel` or `APlayerController::ClientTravel`. On a client, `OpenLevel` and `ClientTravel` both work, for regular and seamless travel alike; what breaks Fusion is raw `UWorld::SeamlessTravel` on the Master Client that bypasses `ChangeWorld`.

CHANGEWORLD: MOVING THE WHOLE ROOM

`UFusionOnlineSubsystem::ChangeWorld(TSoftObjectPtr<UWorld>, WorldContext)` is the Master-Client-only call that schedules the next world for the entire room. It takes a soft pointer to the target world and replicates the request to every peer, so they all load the same map. `ChangeWorldByName(FString, WorldContext)` is the string-path overload it funnels into.

```
if (OnlineSubsystem->IsMasterClient())
{
    OnlineSubsystem->ChangeWorld(NextLevel, this);
}
```

CLIENTTRAVEL: LEAVING ON YOUR OWN

When a single peer wants to leave the room, returning to the main menu after a defeat, say, it uses local travel, which does not replicate. `UFusionHelpers::FusionClientTravel` is the Blueprint-friendly wrapper: it parses an Unreal URL and forwards the resolved map name to the subsystem's `ClientTravel`, which refuses to run on the Master Client (the master is expected to use `ChangeWorld`).

```
UFusionHelpers::FusionClientTravel(
    this, FName("MainMenu"), /*bAbsolute=*/true, FString());
```

> **TIP** **FusionClientTravel replaces ClientTravel**
>
> In a Fusion project, reach for `UFusionHelpers::FusionClientTravel` anywhere you would have called `APlayerController::ClientTravel`, including the path `AGameMode::RestartGame` would otherwise take on a non-master peer.

THE MAP-LOAD CALLBACK SEQUENCE

Three multicast delegates fire on the subsystem, in order, during a map change.

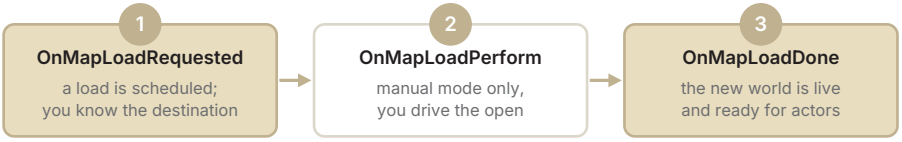


FIG 10.1 The map-load sequence. The middle step fires only in manual mode, where your handler issues the actual open.

The three map-load callbacks.

Order	Callback	Fires when
1	`OnMapLoadRequested`	Always: a load is scheduled; the destination is known.
2	`OnMapLoadPerform`	Manual mode only: your handler drives the actual open.
3	`OnMapLoadDone`	Always: the new world is live and ready for actors.

```
void UMyGameInstance::Init()
{
    Super::Init();
    UFusionOnlineSubsystem* S =
        GetSubsystem<UFusionOnlineSubsystem>();
    S->OnMapLoadRequested.AddDynamic(
        this, &UMyGameInstance::OnRequested);
    S->OnMapLoadPerform.AddDynamic(
        this, &UMyGameInstance::OnPerform);
    S->OnMapLoadDone.AddDynamic(
        this, &UMyGameInstance::OnDone);
}
```

`OnMapLoadPerform` fires only when auto-load is off: when `LoadMapAutomatically` is false in settings, or the `Fusion.LoadMapBehaviourOverride` CVar forces manual mode (Chapter 11). In manual mode the client stays in the loading state until your handler calls the actual open, which is exactly the window in which to show a loading screen or stream assets.

STOP FUSION SESSION AND RESTART GAME

`StopFusionSession` has two overloads. The parameterless form tears down the underlying client. The `ReturnWorld` overload does the same teardown and then opens a non-networked map. The async leave/disconnect/connect actions all call the teardown path internally, so

there is no separate teardown to remember.

> PITFALL **Leaving multiplayer onto a dead map**
>
> Use the `ReturnWorld` overload when returning to a main menu. Without it, the player stays on the now-detached networked map, leaving networked actors and components around in a degraded state. With `ReturnWorld` they land on a clean offline level immediately.

`AGameMode::RestartGame()` works under Fusion but only behaves correctly on the Master Client. Called on a non-master peer it triggers a local client travel that does not replicate; that peer travels alone and the others watch them vanish. For round restarts, do not call `RestartGame` directly: replicate the intent through gameplay state (typically the MatchState on the GameState) and let the Master Client drive the `ChangeWorld` from its handler.

SEAMLESS VS NON-SEAMLESS, AND PRESERVING PLAYERSTATE

Fusion detects seamless travel and filters the transition map out of networking, so the brief transition level never becomes the networked world. Seamless travel on the Master Client must originate from `ChangeWorld`; calling `UWorld::SeamlessTravel` directly from another path is not a supported route and can leave Fusion in an inconsistent state. Enable `bUseSeamlessTravel` only after confirming the transition map is non-networked, its actors must not carry a Fusion Actor Component. Non-seamless is the safer default and the path the code exercises most heavily.

Across any travel, store travel-critical per-player data on `APlayerState`, not on the controller or pawn. The new map's GameMode re-spawns controllers and pawns, and they lose their state; the PlayerState survives in the `InactivePlayerArray` within PlayerTTL, so score, team, and persistent flags belong there.

CHAPTER 11

11

LEVEL STREAMING AND WORLD PARTITION

Fusion observes streaming rather than driving it. You decide which sub-levels load; Fusion binds the networked identities of the actors inside them.

IN THIS CHAPTER

- How Fusion Observes Streaming
- World Partition
- Persistent vs Streamed Replicated Actors
- All Clients Need the Same Persistent Level
- Coordinating Sub-Levels Across Clients
- The LoadMapBehaviourOverride CVar

Streaming is one place where Fusion is deliberately hands-off. It watches Unreal's level events and binds replicated identities accordingly, but it never decides which sub-levels to load; that stays your call, and coordinating it across clients stays your job.

HOW FUSION OBSERVES STREAMING

The plugin subscribes to Unreal's level-added and level-removed delegates and forwards them to the client. It does not decide which sub-levels to stream; it only reacts to whatever level set your game has chosen. On every sub-level add, Fusion walks the level's actors and registers every actor that is `bNetStartup` and whose name is stable for networking. Each one is keyed by a hash built from the level's name and the actor's index in that level. That hash is what lets two clients agree on a map actor's networked identity without any shared table.

Your game remains responsible for calling `LoadStreamLevel` and `UnloadStreamLevel` (or the World Partition equivalent). Fusion observes the result and binds identities; it does not stream levels on its own, and it does not implicitly coordinate streaming across peers.

WORLD PARTITION

Partition streaming runs on Unreal's standard pipeline, the same as in a non-Fusion project; Fusion does not drive cell loading. Because Fusion is the transport and there is no replication server to acknowledge the standard make-visible and make-invisible transactions, the plugin sets the skip-transaction flags on every streaming level, so Unreal does not wait on an acknowledgement that will never arrive.

> NOTE **Partition assumes a shared persistent level**
>
> Fusion's model attaches the current map on the Master Client and assumes a shared persistent level. Partition workflows where different clients see different cells are project-specific and must be designed around that. When consistency matters (every peer must have the boss arena loaded before the fight) coordinate activation through replicated state on a shared, typically MasterClient-owned, actor.

PERSISTENT VS STREAMED REPLICATED ACTORS

Map actors, those with `bNetStartup` and a stable networking name, are eligible whether they live in the persistent level or a streamed sub-level. For duplicate instances of the same sub-level, set an `OptionalLevelNameOverride` on each so their actors hash to distinct identities; this is the same name-override requirement stock Unreal networking has. Dynamically spawned actors with a Fusion Actor Component follow the spawn/destroy path from Chapter 3 instead, and are unaffected by level add/remove events; their lifetime is governed by ownership, not by the sub-level they spawned into.

ALL CLIENTS NEED THE SAME PERSISTENT LEVEL

The map-change handshake assumes a shared persistent level, and the room-wide `ChangeWorld` flow brings every client onto the same active world. Differences between clients must live in sub-levels loaded on top of a shared persistent base, never in the persistent level itself. If clients have different persistent levels, the Master-Client-driven map walk will not match remote map-actor entries, and `bNetStartup` actors become invisible across peers.

> PITFALL **Diverging persistent levels**
>
> It is tempting to give each client a slightly different base map for a local variation. Don't. Keep one shared persistent level for everyone and push every difference into streamed sub-levels; otherwise startup actors silently fail to replicate and the bug is maddening to track down.

COORDINATING SUB-LEVELS ACROSS CLIENTS

Fusion does not replicate `LoadStreamLevel` and `UnloadStreamLevel`; each client decides independently which sub-levels to stream. If every client must load the same sub-level, you coordinate it. Two patterns are common: an RPC (Chapter 8) telling every client to load the sub-level, or, usually cleaner, a replicated property on a MasterClient-owned actor whose OnRep triggers a local load.

```
UPROPERTY(ReplicatedUsing = OnRep_ArenaActive)
bool bArenaActive = false;

UFUNCTION()
void OnRep_ArenaActive()
{
    FLatentActionInfo Latent;
    if (bArenaActive)
        UGameplayStatics::LoadStreamLevel(
            this, FName("Arena"), true, false, Latent);
    else
        UGameplayStatics::UnloadStreamLevel(
            this, FName("Arena"), Latent, false);
}
```

Partial overlap is a supported state: a networked actor that lives inside a sub-level only has local registrations on the clients that have streamed that sub-level in. A client without the sub-level loaded simply does not receive updates for those actors; they exist in the room but not on that

client. Streaming the level in later retroactively creates the registrations and resumes updates.

THE LOADMAPBEHAVIOUROVERRIDE CVAR

`Fusion.LoadMapBehaviourOverride` overrides the `LoadMapAutomatically` project setting at runtime: `0` uses the project setting, `1` forces auto-load, and `2` disables auto-load so your game drives travel via the `OnMapLoadPerform` delegate from Chapter 10. Set it in your project module's startup, before Fusion initialises, when you need a non-default value.

```
// In project module startup, before Fusion initialises:
IConsoleManager::Get()
    .FindConsoleVariable(
        TEXT("Fusion.LoadMapBehaviourOverride"))
    ->Set(2); // disable auto-load; drive travel yourself
```

The CVar affects only the persistent-level switch. Sub-level streaming is unaffected: `LoadStreamLevel` and `UnloadStreamLevel` are driven directly by your game and never consult it.

PART FIVE

SHIPPING IT

Seeing inside replication when it misbehaves, and the judgement call the whole model turns on: where shared authority fits your game, and where it does not.

CHAPTER 12

12

LOGGING AND DEBUGGING

When replication misbehaves, you need to see inside it. Fusion ships a log channel, severity controls, an on-screen sink, and editor inspectors for bandwidth and memory.

IN THIS CHAPTER

Distributed systems fail in distributed ways: the bug is rarely on the machine you are looking at. Fusion's debugging surface is built around that reality: it lets you watch what is happening across clients without a debugger attached to each one.

THE LOGFUSION CHANNEL

`LogFusion` is the log category every plugin-side message lands in. Filtering the Output Log by `LogFusion` is the fastest way to see what Fusion is doing: connection, join, ownership, and map changes all narrate themselves here. Make it the first thing you filter to when something looks wrong.

LOGGING MACROS

Five macros wrap the shared Fusion-core logging, so call sites stay symmetric with the cross-platform core code rather than calling `UE_LOG` directly. The important practical difference: messages flow through every configured output sink, including the on-screen overlay, not only the standard log channel.

Macro	Level	Use for
`FUSION_LOG_TRACE`	Trace	Per-frame internal diagnostics
`FUSION_LOG_DEBUG`	Debug	Verbose flow tracing during development
`FUSION_LOG`	Info	Notable lifecycle events (connect, join, map change)
`FUSION_LOG_WARN`	Warning	Unexpected state that did not break gameplay
`FUSION_LOG_ERROR`	Error	Failures that need attention

```
FUSION_LOG_WARN("Object %s lost ownership during transfer",
    *GetName());
```

> **TIP** **Prefer the macros over UE_LOG(LogFusion, ...)**
>
> A raw `UE_LOG` only reaches the standard log channel. The `FUSION_LOG_*` macros route through every configured sink, including the on-screen overlay, so the same line you write once shows up everywhere you have asked Fusion to surface logs.

CONFIGURING LEVELS AND OUTPUTS

Two parallel bitmasks in the Fusion settings control logging. `EnabledLogLevels` gates which severities are forwarded; bits that are not set are dropped before they reach any sink. `EnabledLogOutput` selects the active sinks: `StandardLogOutput` writes into the `LogFusion` channel, while `OnScreenDebugMessageLogOutput` routes to the viewport as on-screen debug messages.

ON-SCREEN DEBUG MESSAGES

Enabling the on-screen sink surfaces Fusion log calls as viewport messages alongside their Output Log entries, useful for spotting events during play without watching the log window. The catch is volume.

> **PITFALL** **Drowning the viewport in Trace**
>
> Combine the on-screen sink with a restrictive `EnabledLogLevels`: Warning and Error only is a good default. Trace and Debug multiply quickly during a play session and will overflow the viewport, hiding the one message you actually needed to see.

EDITOR DEBUG PANELS

Fusion ships editor inspectors that scan for active sessions and host a per-world tab for each, so with multiple PIE clients running, each gets its own tab and the panels do not collide. Find them under Window > Photon > Fusion Debug Tools. Two are worth knowing by name.

The bandwidth inspector shows per-object send and receive bandwidth as sortable trees plus a time-series graph. It is the right starting point for any "why is bandwidth high" investigation: sort by send size and the heaviest replicator is usually sitting at the top, telling you exactly which actor to put on a diet, with interest management (Chapter 7) or manual state-copy control (Chapter 4).

The string-heap inspector shows per-object string-heap occupancy and visualises fragmentation. Entries that never get freed and slowly bloat the heap show up as a growing slab over time, the visual signature of a leak in replicated string state.

RUNTIME CVARS

Fusion's runtime CVars are set the standard Unreal way: from the in-game console, a `.ini` file, or `IConsoleManager` from C++.

CVar	Effect
`Fusion.LoadMapBehaviourOverride`	0 = use project setting; 1 = force auto-load; 2 = disable auto-load (you drive travel via OnMapLoadPerform).
`Fusion.DisableGameStateNetworking`	Skip auto-attaching AGameStateBase as a networked source, for projects shipping their own GameState replication.

```
Fusion.LoadMapBehaviourOverride 2
```

As in Chapter 11, set `LoadMapBehaviourOverride` in your project module's startup, before Fusion initialises, when you need a non-default value.

CHAPTER 13

FROM PROTOTYPE TO PRODUCTION

You have the pieces. The last question is the one this whole model turns on: where does shared authority fit your game, and where does it not?

IN THIS CHAPTER

- The Authority Question, Revisited
- Living With Cheat Tolerance
- How the Pieces Fit
- A Shipping Checklist

Every chapter so far has been about how Fusion works. This one is about judgement: when its trade-offs are the right ones, how to live with the cheat tolerance they imply, and how the pieces fit together into a shippable whole.

THE AUTHORITY QUESTION, REVISITED

We opened the book with the central trade: shared authority gives you cheap, scalable, operationally simple multiplayer in exchange for cheat tolerance, because clients write their own state. By now that sentence should read very differently than it did in Chapter 1: you know exactly which state clients write, who the owner is, and how the Master Client coordinates the rest.

The honest guidance is unchanged and worth stating plainly. Co-op, casual, party, sandbox, social, persistent worlds, mobile, web, and XR are where shared authority shines, the games where a player cheating mostly hurts themselves, or where the social context already polices behaviour. Ranked competitive PvP, where a single cheater poisons the experience for honest players and there is money or status on the line, still wants a dedicated server. Most games are not ranked competitive PvP.

LIVING WITH CHEAT TOLERANCE

Cheat tolerance is not the same as defencelessness. Shared authority changes where you spend your integrity budget, not whether you have one.

- **Own the state that matters on the Master Client.** Match flow, scores that decide a winner, and shared world state can live on MasterClient-owned objects, where a single coordinated peer writes them.
- **Keep authority close to consequence.** A player owning their own avatar can lie about their position, but they cannot lie about yours. Design so that the worst a client can corrupt is its own experience.

- **Validate where you can.** Even without a server, the Master Client can sanity-check the shared state it owns and react to absurd values it observes from other clients, correcting them rather than letting them stand.
- **Add a real authoritative server for the rare value that truly must be tamper-proof.** When a value genuinely cannot tolerate any client writing it, that needs a custom authoritative server, which sits outside the standard Fusion Cloud model.

> TIP **Match the integrity to the stakes**
>
> Spend your anti-cheat effort in proportion to what a cheat actually costs your players. For a co-op base-builder, that may be almost nothing. For a leaderboard with prizes, it is the Master Client and a real authoritative server. Knowing which game you are making is most of the answer.

HOW THE PIECES FIT

Step back and the architecture is coherent. You connect to a room (Chapter 2). You spawn actors that opt into networking through a component (Chapter 3), replicate their properties with familiar `UPROPERTY` markup (Chapter 4), and decide per actor who owns them (Chapter 5). Physics stays smooth through Forecast (Chapter 6), bandwidth stays bounded through interest management (Chapter 7), and events travel as explicitly targeted RPCs (Chapter 8). The game framework adapts to having no server by gating room-wide logic on the Master Client (Chapter 9), changing maps through `ChangeWorld` (Chapter 10), and coordinating streaming through replicated state (Chapter 11). When it misbehaves, the log channel and editor inspectors show you where (Chapter 12).

None of it requires you to run infrastructure. That, in the end, is the point: it lets a small team ship multiplayer that scales, and spend its energy on the game instead of on the servers underneath it.

A SHIPPING CHECKLIST

Before you call a Fusion build production-ready, walk this list. Each item maps to a chapter, and most correspond to a bug I have personally shipped at least once.

- `AppVersion` is wired to your build pipeline, so incompatible builds never share a room.
- Region selection uses Best on first launch and Select thereafter.
- Network-dependent initialisation runs in `OnObjectReady`, never `BeginPlay`.
- Every write to replicated state is gated on `IsOwner` or `CanModify`.
- Match-flow logic in the GameMode is gated on `IsMasterClient`.
- State that must survive a join is a replicated property, not an RPC.
- State that must survive a disconnect lives on `APlayerState` or a MasterClient-owned actor.
- Heavy replicators have been checked in the bandwidth inspector and given interest keys.
- Returning to the menu uses `StopFusionSession` with a `ReturnWorld`.
- All clients share one persistent level; differences live in streamed sub-levels.

Ship it. Then watch the bandwidth graph, read the warnings, and fix what real players find. They always find something, and now you know exactly where to look.

No server to provision, no host to babysit. Spend the energy on the game.

GLOSSARY

`Shared authority`	Fusion's model in which every networked object has an owning client that writes its state, with no single authoritative server for the world. Contrast with server authority.
`Owner`	The int32 player id of the client that is the local authority for a given networked actor. Only the owner writes that actor's replicated state.
`Master Client`	The single peer elected to make room-wide decisions (map changes, GameState, singletons). Not a server; auto-migrates when it leaves.
`Room`	The unit of a multiplayer session. A stateful service on the Photon Cloud that holds shared state and relays updates and RPCs.
`UFusionActorComponent`	The component that makes an actor networked under Fusion. Fusion adopts the actor only when it has both this component and bReplicates enabled.
`UFusionOnlineSubsystem`	The UGameInstanceSubsystem that owns the connection and is the entry point for connect, join, and room calls.
`Forecast`	Fusion's distributed physics-prediction module, integrated with Chaos. Predicts physics bodies locally and corrects toward authoritative snapshots.
`Interest management`	Filtering which client receives updates for which object, using 64-bit interest keys and per-client subscriptions. Saves bandwidth and hides state.
`Interest key`	A 64-bit value stamped on an object and matched against a client's subscribed keys. Key 0 means global (everyone receives it).

`Transaction ownership`	The default take-and-release ownership mode: a current owner releases, then another client may claim.
`PlayerAttached`	Ownership mode where the object is destroyed when its owning player leaves the room. Used for avatars and PlayerState.
`EFusionStatus`	The session lifecycle enum: None, Connecting, Connected, JoiningRoom, InRoom, LeavingRoom, plus the terminal Disconnected and Error.
`SEND_FUSIONRPC`	The C++ macro that marks a function as a Fusion RPC with an explicit target (All, Owner, Master Client, Everyone Else).
`ChangeWorld`	The Master-Client-only call that schedules the next map for the whole room, replacing ServerTravel under Fusion.
`AppVersion`	A Project Settings field that segments players by client build so incompatible versions cannot share a room.

ABOUT THE AUTHOR

Victor Chanet is the founder of Victor Game Studio, an independent studio that has shipped Polyplaza Ultimate, Polyplaza 2, and Astral on Steam and the Epic Games Store.

He works primarily in Unreal Engine and C++, and has shipped shared-authority multiplayer in production using Photon Fusion. This book distills what that experience taught him: the parts that ship, not just the parts that compile.

When he is not writing gameplay code, he is usually writing about it, in the hope that the next developer spends less time stuck than he did.

USOURCECONTROL

SOURCE CONTROL, BUILT FOR GAME TEAMS

USourceControl is the version-control system Victor Game Studio uses every day. It is built for the realities of game development rather than retrofitted from a text-only workflow.

Large binary assets, fast iteration, locking for un-mergeable files, and teams that span code and art: these are the cases it is designed around. Multiplayer work in particular is merge-heavy, and a source-control layer that understands game projects removes a surprising amount of daily friction.

If you ship games, the problems it solves are probably already on your desk. Learn more at the Victor Game Studio site.