Introduction to Iris

Learn about the design and components of Iris as well as how to configure your project to use Iris.

Learn to use this **Experimental** feature, but use caution when shipping with it.

Iris is an opt-in replication system that works alongside Unreal Engine's existing replication system. The system builds on Epic's experience with **Fortnite Battle Royale**, which supports up to 100 players per server instance. Iris supports robust multiplayer experiences with:

- Larger, more interactive worlds.
- Higher player counts.
- Lower server costs.

Current game code should work with minimal changes. Opting into Iris requires game code to use new engine APIs. However, existing **replicated properties** and **Remote Procedure Call (RPC)** definitions in C++ and Blueprints are compatible with minor modification. This document provides you with:

- An overview of the iris replication system.
- How to get started with Iris in your project.
- The design of iris, including several important concepts.

- The flow of Iris operation.

Overview

Networked games use a **Replication System** to communicate changes in the game state between multiple computers. Unreal Engine traditionally uses a server-client model, consisting of:

1. A **server** that hosts the game. The server's version of the game is considered to be **authoritative**, meaning that its instance of the game is the true game instance.
2. Player-controlled **clients** that connect to the game.

The replication system governs how game state changes on clients are communicated to the server, and how the accumulated changes on the server are communicated back to clients. Iris expects clients to do most of the work by notifying the replication system when changes are made. This allows Iris to make several performance improvements over the existing replication system.

Use Iris in Your Project

Unreal Engine compiles with Iris by default, but the existing replication system is still used as the default replication system. To enable Iris for use in your project, follow these steps:

1. Ensure that the Iris plugin is enabled by adding the code block below to the `Plugins` section of your `.uproject` file:

   ```
   {
   "Name": "Iris",
   "Enabled": true
   },
   ```

   Copy full snippet

2. Call `SetupIrisSupport` to quickly and easily add Iris's required dependencies to your module's `*.Build.cs` file. Add the following code to your module's `.Build.cs` file to include Iris in your module:

```
        SetupIrisSupport(Target);
```

Copy full snippet

3. Add the following to your project's `*.Target.cs` file:

```
        bUseIris = true;
```

Copy full snippet

4. Add the following your project's `DefaultEngine.ini` configuration file and uncomment settings as needed:

```
[SystemSettings]
; Required for Iris:
net.SubObjects.DefaultUseSubObjectReplicationList=1
; Required if net.Iris.PushModelMode is set to 1:
; net.IsPushModelEnabled=1
; Required if using an engine version past 5.1 where Iris compilation is enabled by default:
net.Iris.UseIrisReplication=1
```

Copy full snippet

Enable or Disable Iris at Runtime

Iris can be enabled or disabled at runtime with a command line argument:

- `-UseIrisReplication=1`: enables Iris
- `-UseIrisReplication=0`: disables Iris

Net Drivers that have enabled Iris in their `IrisNetDriverConfigs` entry will use Iris for replication.

Disable Compiling Unreal Engine with Iris

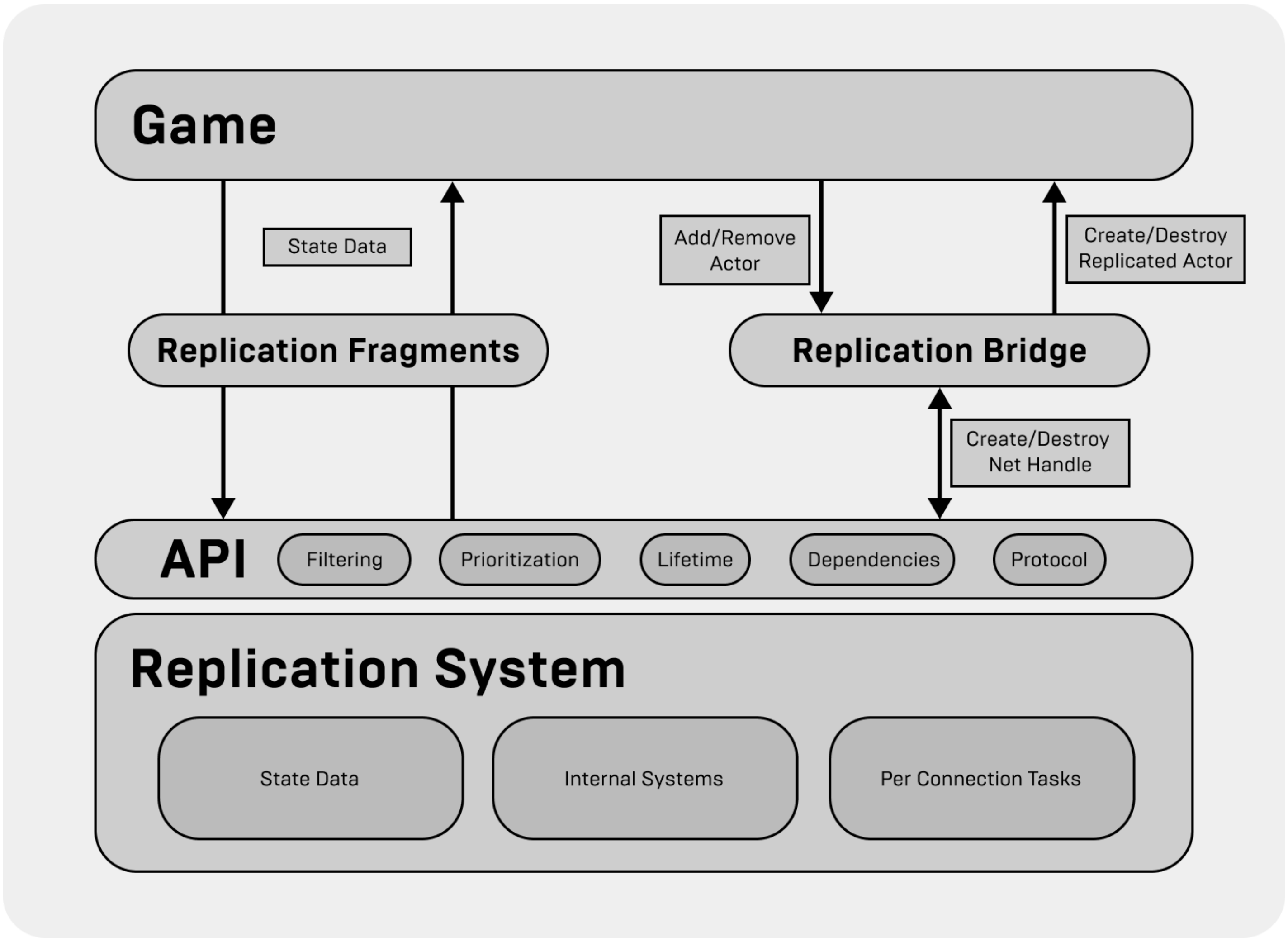
If you want to completely disable compiling Unreal Engine with Iris, set `bUseIris = false` in your `.Build.cs` file.

Performance Improvements

Iris achieves replication performance improvements by:

- Increasing scalability by removing antipatterns that constrain it.
- Enabling concurrency by separating replication and game thread data.
- Improving efficiency by sharing workloads for multiple objects and connections.

Design



Click for full-size image. Important terms in this diagram are defined in the following two sections: Primary Components of Iris and Important Concepts.

The **Iris Replication System** is the main interface for Iris in **Unreal Engine (UE)**. Iris's core goal is to minimize dependencies between the gameplay system and the replication system. Iris does this by keeping a full copy of all replicated state data in a quantized form. This minimizes the number of expensive operations and makes it possible to share work between connections, making it straightforward to do more work in parallel.

Primary Components of Iris

Replication System

The Replication System is an interface layer to Iris's internal systems exposing only necessary API functionality. The replication system performs the following functions:

- Maintains a copy of all networked state data for the game.
- Tracks state of replicated actors per-connection.
- Filters what actors are replicated to which connections.
- Prioritizes the order of replication.
- Serializes data for transport.

Replication System Components

The replication system's primary components are described in the following table:

Component	Description
Replication States	A struct containing data that needs to be replicated. The Replication System maintains a copy of all networked state data for the game.
Prioritization	Prioritize the order of replication for actors and objects.
Filtering	Filter what actors are allowed to replicate to which connections.
Net Serializers	Serialize data for transport over a network connection.
Data Streams	Interfaces that implement the replication of data over a network connection.

Replication Bridge

The **Replication Bridge** controls communication between gameplay code and the replication system. The replication bridge:

- Begins and ends replication for actors or objects.
- Builds descriptors and protocols for replicated data.
- Adds and removes actors or objects from the replication system.

Important Concepts

Net Object

Replicated actors and objects are internally represented in Iris as a Net Object. A **Net Object** consists of a:

- Replication Protocol.
- Replication Instance Protocol.
- Buffer to store quantized data.

Replication State

A **Replication State** communicates state data between the replication system and gameplay code. In its most basic form, a replication state is a struct containing data to be replicated. A replication state can be explicitly constructed or an abstract representation built from existing property based reflection data.

Replication State Descriptor

A **Replication State Descriptor** describes all aspects of a replication state required to replicate the data. This includes:

- Memory Layout
- Conditionals
- Filtering
- Prioritization
- Serialization

Every replication state type has a replication state descriptor. Replication states of the same type use the same replication state descriptor.

Replication Protocol

A **Replication Protocol** describes all aspects of a replicated object required for internal, replication system operations. This includes a list of all replication state descriptors making up the total state of a replicated object. Replication protocols are per object type and shared between all instances of the same type.

Replication Fragment

A **Replication Fragment** is the Iris component responsible for carrying replication states back-and-forth between gameplay code and the replication system.

Replication Instance Protocol

A **Replication Instance Protocol** contains the data necessary to interact with gameplay code for operations such as getting data from the source object and pushing out received state data to the target object on the receiving end. A replication instance protocol is represented as a list of replication fragments. Replication instance protocols are instance-specific.

Net Handle

All API functions operate on net handles. A **Net Handle** is a unique identifier used to associate a replicated actor or object with the internal net object representation used by the replication system. A net handle is generated by the replication system and returned when `BeginReplication` is called on an actor.

Flow of Iris Operation

This section describes the operational flow of Iris starting with the registration of objects for replication and ending with the application of newly received state data on the receiving end.

Registration

Registering objects for replication with Iris requires actions from both the game code and Iris.

Game Code

- Registers an object with the replication system.
- Declares a replication state and replication state descriptor either explicitly or implicitly through header properties.

Iris

- Constructs a replication protocol and replication instance protocol using all replication states defined by the object and its components.
- Creates a net object corresponding to the newly registered object using the previously constructed replication protocol and replication instance protocol.
- Assigns the replicated object a unique net handle so that Iris' API functions can interact with this replicated object.

Sender

The sending side consists of two stages:

1. Pre-Send Update
2. Send Update

Pre-Send Update

The pre-send update is composed of several steps that Iris takes in preparation for sending data to the receiving side. In the pre-send update, Iris:

- Polls all replicated objects for state changes if running in legacy mode.
- Quantizes all dirty state data with the appropriate net serializer.
- Updates the filtering status and prioritization of all net objects.

Send Update

The send update consists of creating and filling packets by ticking all Iris data streams.

Net Token Data Stream

The net token data stream serializes any new tokens with the appropriate net serializer.

Replication Data Stream

For data read from the replication data stream, Iris:

- Schedules objects with dirty state to be sent based on object and scheduling priority.
- Sorts objects based on priority and dependencies.
- Serializes any new data with the appropriate net serializer.

At this stage, Iris sends data to the appropriate connections.

Receiver

As soon as the receiver starts receiving packets, the receiver notifies the sender of any received packets. For each packet that is received, the appropriate data stream processes the data contained in the packet:

Net Token Data Stream

The net token data stream synchronizes object paths between server and client so they can be replicated using a smaller representation.

Replication Data Stream

For data read from the replication data stream, Iris:

- Deserializes and reads the received state data.
- Instantiates new objects immediately since they are required to build replication protocols.
- Pushes data to the game so that the new state can be applied to the appropriate replicated objects and reflected in gameplay.