Components of Iris

Learn more about the primary components of the Iris replication system and how to use them.

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

The **Iris Replication System** is the main interface for Iris in **Unreal Engine (UE)**. Iris minimizes dependencies between the gameplay system and the replication system by keeping a full copy of all replicated state data in a quantized form. Iris only performs expensive operations once and shares work between connections, which allows the system to do more work in parallel. This parallel work saves time and resources.

This page provides:

- Instructions for how you access the replication bridge and replication system
- An overview of the internal workings of how Iris replicates objects.

Interface

The Iris replication system interface lets you manage the replication system itself as well as providing functions to control:

- Connections
- Groups
- Prioritization
- Filtering
- Remote Procedure Calls

All Iris interface functions operate on net ref handles (`FNetRefHandle`) created through the replication bridge. A *net ref handle* is an identifier created by the Iris replication bridge. The net ref handle is a key that uniquely maps a

network object to its associated gameplay instance (`UObject`, `UActorComponent`, or `AActor`) provided during creation.

> For more information about the terminology used to describe Iris, see the Glossary of Iris Terms documentation page.

Replication Bridge

The Iris replication bridge is responsible for all functionality that operates on gameplay instance objects, such as actors. The replication bridge sits between your gameplay code and the internal Iris replication system. The bridge is the central channel for all operations in Iris that involve communicating state data between the Iris replication system and gameplay code.

On the sending machine, the replication bridge manages the lifetime of network objects and the protocols required by the replication system. You can use the replication bridge to obtain a net ref handle for your replicated object. Then use the net ref handle with the replication system to customize replication settings for your object.

On the receiving machine, the replication bridge is responsible for instantiating actors and managing their lifetime.

More explicitly, the replication bridge:

1. Begins and ends replication for replicated objects with `BeginReplication` and `EndReplication`.
2. Drives the creation of replication protocols and replication instance protocols.
3. Serializes and deserializes the data required to instantiate objects.
4. Instantiates objects on the game side for the replication system.
5. Handles the mapping between gameplay instances and net ref handles.

> Relevant header files for the replication bridge include:
>
> - `..\Engine\Source\Runtime\Engine\Public\Net\Iris\ReplicationSystem\ActorReplicationBridge.h`
> - `..\Engine\Source\Runtime\Experimental\Iris\Core\Public\Iris\ReplicationSystem\ObjectReplicationBridge.h`

Control Replication

To replicate an object, Iris first creates a network representation of the object. The replication bridge manages the creation of net objects and net ref handles in `BeginReplication`.

Begin Replication

During `BeginReplication`, Iris sets up a replication protocol that describes all aspects of the replicated data associated with the object. This replication protocol is shared for all instances of the same type. For each instance of an object, the replication system creates a replication instance protocol that contains instance specific data.

Instance specific data includes where to locate the data that should replicate and where to push received data for this particular instance.

Access Replication Bridge

Once Iris creates a net ref handle for a replicated object, you can use the replication system API to control how the object replicates. To obtain the net ref handle for your replicated object, follow these steps:

1. Include the necessary Iris files:

```
#if UE_WITH_IRIS
#include "Net/Iris/ReplicationSystem/ReplicationSystemUtil.h"
#include "Net/Iris/ReplicationSystem/ActorReplicationBridge.h"
#endif UE_WITH_IRIS
```

Copy full snippet

2. Retrieve a reference to the replication bridge for your actor:

```
// The actor for which you want to obtain its net ref handle
AActor* RepActorPtr;

UActorReplicationBridge* ReplicationBridge =
UE::Net::FReplicationSystemUtil::GetActorReplicationBridge(RepActorPtr);
```

Copy full snippet

3. Obtain the net ref handle for your replicated actor:

```
UE::Net::FNetRefHandle RepActorNetRefHandle = ReplicationBridge-
>GetReplicatedRefHandle(RepActorPr);

// Now you can retrieve the replication system and perform API operations
```

Copy full snippet

End Replication

During `EndReplication`, the replication system cleans up any networking resources created for your gameplay object and stops replication of the object.

State Tracking

Iris performs optimally when the game notifies the replication system when there is work to do. Iris does not expect the replication system to poll for changes. The replication system expects to be notified whenever a replicated property is modified. To implement this modification system efficiently, all replicated variables in Iris are part of

replication states. A *replication state* is a struct containing data that should be replicated or transmitted over the network.

Replication states have allocated space for bitfield tracking the dirtiness of members. A member is *dirty* if the state of the member has changed, but the change has not yet been communicated to downstream systems. The replication state also contains an embedded identifier used to notify the replication system that an object has dirty state data that needs to be updated.

Replication System

The replication system is the primary API for interacting with replicated objects in Iris. When you have obtained the net ref handle of the object you want to interact with through the replication bridge, you can customize the object's replication behavior. Customization includes filtering which connections the object replicates to or updating the object's replication priority.

Access Replication System

To access the replication system used for a replicated object:

1. Include the necessary Iris files:

```
#if UE_WITH_IRIS
#include "Net/Iris/ReplicationSystem/ReplicationSystemUtil.h"
#endif UE_WITH_IRIS
```

Copy full snippet

2. In your gameplay code, retrieve the replication system for your replicated object:

```
// The actor for which you want to obtain its net ref handle
AActor* RepActorPtr;

UReplicationSystem* ReplicationSystem =
UE::Net::FReplicationSystemUtil::GetReplicationSystem(RepActorPtr);
```

Copy full snippet

Relevant header files for the replication system include:

- `..\Engine\Source\Runtime\Engine\Public\Net\Iris\ReplicationSystem\ReplicationSystemUtil.h`
- `..\Engine\Source\Runtime\Experimental\Iris\Core\Public\Iris\ReplicationSystem\ReplicationSystem.h`

For examples on how to customize replication for replicated objects, see the Filtering and Prioritization documentation.

Process Flow of Replication System

This section describes the process flow for the internal workings of Iris and how the different components of Iris work together.

The process flow consists of a sending machine and one or more receiving machines. The sending machine has new information about replicated objects that the sender must communicate to other connected machines in a multiplayer session. The replication process for the sending machine consists of two steps: a pre-send update and a send update. The connected machines that have outdated information are the receiving machines that must update their information about replicated objects to match the sending machine's state.

Net serializers and data streams are used during the process of communicating data between sending and receiving machines. Net serializers transform data for efficient transport over a network. Data streams define how that data is sent, including any special delivery guarantees.

Sending Machine

Pre-Send Update

The pre-send update on the outgoing side of the replication system is where the bulk of the shared work is done. This is also the only time that the system accesses data from gameplay code. This update consists of the following steps:

1. If running in legacy mode, poll replicated objects for state changes.
2. Copy and quantize all dirty state data.
3. Reset the dirtiness status of replicated objects.
4. Update the filtering status of net objects.
5. Update prioritization of net objects.

Send Update

When filling out packet data, the goal is to not touch any data that is not contained within the replication system. This makes it easier to accomplish tasks concurrently, in a cache-friendly manner. The send update process proceeds as follows for all connections:

- Create and fill packets by ticking all data streams.
    - Replication Data Stream:
        1. Schedule objects with dirty state to be sent based on object priority and scheduling priority.
        2. Sort based on priority and dependencies.
        3. Serialize state data.

Receiving Machine

The end receiving the packets processes incoming packets as follows for all connections:

- First update delivery notifications.

- Data streams process the data contained in each packet:
  - Replication Data Stream:
    - **Read state data:** State data is dispatched after the entire packet is read. New objects are instantiated immediately since the replication system relies on them to build replication protocols.
    - **Dispatch state data:** Replication system pushes new data to the game to update gameplay objects.

More Information

For more information about the components of the Iris Replication System, including examples of how to change the replication behavior of actors, see the other documentation pages in this section:

Filtering

Filter object replication to certain network connections with Iris.

Prioritization

Prioritize objects for replication with Iris.