Replication Graph

Overview of the Replication Graph feature and Replication Graph Nodes.

The **Replication Graph** Plugin is a system for network replication in multiplayer games that is designed to scale well with large numbers of players and replicated Actors. As an example, Epic's own Fortnite Battle Royale starts each game with 100 connected players and about 50,000 replicated Actors. The standard network replication strategy, which is to require each replicated Actor to determine whether or not it should send an update to each connected client, performs poorly in cases like this and will bottleneck the server's CPU. Solutions like dividing Actors into staggered groups, or simply updating less frequently, might mitigate the issue, but will also degrade the client experience by decreasing update frequency. The Replication Graph eliminates the need for Actors to evaluate each connected client individually, solving the CPU performance issue without sacrificing the client experience.

Structure

The Replication Graph contains a series of **Replication Graph Nodes**, which are responsible for building lists of Actors to replicate to each client on demand. Because this system is built from persistent objects, as opposed to merely being function calls processed by the replicated Actors themselves, data can be stored across multiple frames and shared between client connections. This persistent, shared data is what enables the Replication Graph system to decrease the time it takes to produce replication lists for each client.

Replication Graph Nodes (we'll call them "nodes" for short) do the actual work of establishing which Actors potentially require updates, sorting them into groups, storing precomputed lists to send to clients, and so on. Their ultimate task is to provide "replication lists" of Actors on demand to each client connection as quickly as possible, so that the server spends as few CPU cycles as possible per Actor, per client. Each node can function in a unique way, and developers are encouraged to write custom nodes for their own games as needed. Nodes may be game-agnostic, or may leverage game-specific information. Putting Actors into different nodes based on their roles in your game can give you better control over how and when they replicate. Building new nodes and using the Replication Graph to assign Actors to the best nodes possible based on how they behave in your game will provide the greatest decreases in server CPU time spent on preparing network replication lists.

Enabling The System

You can configure your project to use a custom **Replication Driver** (the parent class of Replication Graph) in one of two ways:

- Specifying a Replication Driver class in the "DefaultEngine.ini" file.
- Binding a function that returns an instance of your Replication Driver class to the default Replication Driver's creation Delegate.

The ShooterGame project is a good example of how to set up and implement a Replication Graph. However, note that Replication Graph is disabled on console builds because it does not currently work in splitscreen games.

Configuration (.ini) Files

To configure the Engine's default Replication Driver, open the "DefaultEngine.ini" file for your project. Find (or add) the `[/Script/OnlineSubsystemUtils.IpNetDriver]` section and set (or add) the "ReplicationDriverClassName" entry so that it indicates the name of the Replication Driver (or Replication Graph) class you wish to use. This should look roughly as follows, substituting "ProjectName" with your actual project's name, and "ClassName" with your custom class name:

```
[/Script/OnlineSubsystemUtils.IpNetDriver]
ReplicationDriverClassName="/Script/ProjectName.ClassName"
```

Copy full snippet

Binding In Code

If your project has multiple game modes or maps that have vastly different networking requirements, binding to a Delegate will enable you to create the appropriate Replication Driver for the current game mode or map in code. To use this method, bind a function to the `UReplicationDriver` function called `CreateReplicationDriverDelegate`. Your bound function must return a valid instance of your desired Replication Driver class, as this sample lambda function does:

```
UReplicationDriver::CreateReplicationDriverDelegate().BindLambda([](UNetDriver* ForNetDriver, const FURL& URL, UWorld* World) -> UReplicationDriver*
{
return NewObject<UMyReplicationDriverClass>(GetTransientPackage());
});
```

Copy full snippet

High-Level Example

For a game with a large number of connected clients and an even larger number of synced Actors, a Replication Graph that assigns Actors based on type and status to different nodes can save a tremendous amount of CPU time. This makes it possible to build games that would not be viable with traditional replication methods. At a conceptual level, a game at this scale might build a Replication Graph and Replication Graph Nodes with the following features to handle its huge volume of replicated Actors and connected clients:

- **Separate Actors into groups based on location.** The world can be divided up into grid spaces for games in the battle royale, MOBA, or MMORPG genres, or predefined rooms or zones for dungeon crawlers or corridor-style first- or third-person shooters, or any method that fits your game's play spaces. Adding Actors into each grid cell or room from which that Actor can potentially be seen or heard will make client updates fast, as the node can simply provide the client with the persistent Actor list for whatever grid cell or room the client's camera is in.

- **Identify "dormant" placed Actors and keep them in a separate list.** While some Actors, like those representing players or AI-controlled characters, are likely to need frequent updates, there may be many Actors that are pre-placed in the level and that don't move or change state on their own until a player interacts with them. These Actors may go for a long time (possibly the entire game session) without needing to send a network update. In Fortnite Battle Royale, for example, players and projectiles would be expected to update constantly until being removed from the game. A tree, on the other hand, would be expected to lie dormant for a long time, requiring no updates to any client. When the tree is damaged, any client who can see the tree would need to receive an update about it. Finally, when the tree is destroyed, any client who receives the update describing the tree's destruction never needs to receive any further updates about the tree.
- **If characters in your game can pick up and carry items, update those items with their carriers.** When a player pulls out an item or weapon and carries it around, or wears a piece of clothing or armor, add the Actor representing the item (assuming it is a separate Actor and not merely a Component) to a special group that always gets updated when the owning player updates, and never updates otherwise.
- **Make a list of special Actors that always known to all clients.** Special Actors that are always network-relevant to every player and can be put into a simple node that tracks these Actors, keeping them out of other lists where they might eat up CPU cycles doing unnecessary computation.
- **Make a list of special Actors that are always (or never) relevant to certain clients.** A similar always-relevant list node could be made for individual players, or for teams of players. This is especially useful for things like ensuring that a player's teammates are always updated, or that opponents who have been "revealed" by a special in-game detection power are visible to the entire team of the player who revealed them. If the "reveal" expires, these Actors can be added back into their default nodes.

Building a Replication Graph that intelligently assigns Actors to different nodes based on knowledge of the Actor's role within the game can make the best use of your server's CPU time. The end result is steady server performance for games that otherwise could not run on current hardware. The Replication Graph Plugin includes several Replication Graph Node classes that you can use in large-scale online games. Developers are also encouraged to build custom node classes based on knowledge of the inner workings of their specific game.