Detailed Actor Replication Flow

A detailed description of low-level Actor replication.

Actor Replication is a detailed, multi-step process where the **Network Driver** (Net Driver) determines what actors need to replicate to which connections in what order. This page provides an overview of the actor replication process.

The majority of actor replication happens inside the `UNetDriver::ServerReplicateActors` function. This is where the server first gathers all actors it has determined to be relevant for each client, then sends any properties that have changed since the last time each connected client was updated. The `UActorChannel::ReplicateActor` function then handles the details of actor replication to a specific channel.

Important Properties

There is a defined flow for how actors are updated, certain framework callbacks that are invoked, and properties used to determine whether an actor is replicated during the current server tick. Some important ones include:

Property	Description
`AActor::NetUpdateFrequency`	Determines how often an actor should replicate.
`AActor::PreReplication`	Called before any replication occurs.
`AActor::bOnlyRelevantToOwner`	True if this actor only replicates to its owner.
`AActor::IsRelevancyOwnerFor`	Determines relevancy when bOnlyRelevantToOwner is true.
`AActor::IsNetRelevantFor`	Determines relevancy when bOnlyRelevantToOwner is false.
`AActor::NetDormancy`	Determines if an actor is dormant or awake.

Actor Replication Flow Overview

The following steps constitute a high-level overview of the actor replication process:

1. Determine which actors are replicating and perform checks to determine dormancy, update frequency, and owning connection.
    - Add actors that pass these checks to a list to be considered for replication.
2. Loop through each connection and perform checks based on the current actor and connection. At the end of this step, there is a list of actors that are considered for replication for each connection.
    - Sort the actors by priority for each connection.
3. Determine if the actor is relevant for this connection.
4. Replicate the actor to the current connection.

The following sections provide a more detailed description of each step in the above Actor Replication Flow Overview.

Add Actors to the Considered for Replication List

This step performs an initial pass over all actors to determine which actors are actively replicating by checking whether `AActor::SetReplicates(true)` has been called for the actor. For each actor that is actively replicating, the NetDriver performs the following checks:

1. Determine if the current actor is initially dormant (`ENetDormancy::DORM_Initial`).
    - If initially dormant, skip this actor.
2. Determine if the current actor needs to update by checking the `AActor::NetUpdateFrequency` value.
    - If not, skip this actor.
3. If `AActor::bOnlyRelevantToOwner` is true, check the owning connection of this actor for relevancy by calling `AActor::IsRelevancyOwnerFor` on the viewer of the owning connection.
    - If relevant, add to the owned relevant list on the connection.
    - In this case, this actor will only send to a single connection.

For any actor that passes these initial checks, `AActor::PreReplication` is called. In `AActor::PreReplication`, you can decide if you want properties to replicate for certain connections. Use the `DOREPLIFETIME_ACTIVE_OVERRIDE` macro for specific control over which connections an actor replicates to. If the actor passes all the above checks, add the actor to the considered for replication list.

Loop Through Each Connection

Next, the system loops through each connection and performs the following checks and actions for each actor that is on the considered for replication list from the previous step for the current connection:

1. Determine if the current actor is dormant by calling `AActor::NetDormancy`.
    - If this actor is dormant for this connection, skip this actor.
2. If there is no channel yet:
    - Determine if the client has loaded the level the current actor is in.
        - If the level is not loaded, skip this actor.
    - Determine if the current actor is relevant by calling `AActor::IsNetRelevantFor` for the connection.
        - If the actor is not relevant, skip this actor.

Add any actors on the connections owned relevant list from above. At this point, there is a list of relevant actors for this connection that are not dormant. Sort the actors in this list by

priority (`AActor::GetNetPriority`) in decreasing order. Sorting the actors by priority is especially important as we want to ensure that actors with the highest priority are considered for replication before lower priority actors, especially if a large number of actors are being considered and a possibility that the connection becomes saturated.

Loop Through Sorted Actors List

For each actor in this connection's considered replication list:

1. If the connection hasn't loaded the level this actor is in, close the channel (if any) and continue.
2. Every 1 second, determine if the actor is relevant to the connection by calling `AActor::IsNetRelevantFor`.
    - If not relevant for 5 seconds, close the channel.
    - If relevant and no channel is open, open a channel.
    - If at any point this connection is saturated:
        - For remaining actors:
            - If relevant for less than 1 second, force an update next tick.
            - If relevant for more than 1 second, call `AActor::IsNetRelevantFor` to determine if we should update the next tick.

For any actor that passes all of the above, the actor is replicated to the connection with a call to `UActorChannel::ReplicateActor`.

You can control how many clients `UNetDriver::ServerReplicateActors` replicates per call in a few different ways:

1. Engine configuration and command-line arguments:
    a. Launch your project with the `-limitclientticks` command-line argument.
    b. Change the value of `NetClientTicksPerSecond` in the `[/Script/Engine.Engine]` engine configuration category.
2. Command-line arguments:
    a. Launch your project with the command-line arguments: `-limitclientticks -ini:Engine:[/Script/Engine.Engine]:NetClientTicksPerSecond=<VALUE>`, where `<VALUE>` is the number of client ticks per second you want to use.
3. Console variables:
    a. Set the `net.MaxConnectionsToTickPerServerFrame` console variable

See `UNetDriver::ServerReplicateActors_PrepConnections` for more information.

Replicate an Actor to a Connection

`UActorChannel::ReplicateActor` is the primary method that replicates an actor and all its components to a connection. The flow looks something like this:

1. Determine if this is the first update since this actor channel was opened.
    - If so, serialize specific information that is needed (initial location, rotation, etc).
2. Determine if this connection owns this actor.
    - If not owned, and this actor's role is `ENetRole::ROLE_AutonomousProxy`, then downgrade to `ENetRole::ROLE_SimulatedProxy`.
3. Replicate this actor's changed properties.
4. Replicate each component's changed properties.
5. For any deleted components, send a special delete command.

After the list of actors has been exhausted, or the channel has become saturated, the next connection is considered and the process is repeated until all connections have been updated.

More Information

For more information about Actor Replication, see the following header files in the Unreal Engine Source Code:

- `/Engine/Source/Runtime/Engine/Classes/Engine/NetDriver.h`
  - Information about `UNetDriver::ServerReplicateActors`.
- `/Engine/Source/Runtime/Engine/Classes/GameFramework/Actor.h`
  - Information about `AActor` and its functions and properties.
- `/Engine/Source/Runtime/Engine/Classes/Engine/ActorChannel.h`
  - Information about `UActorChannel` and `UActorChannel::ReplicateActor`.
- `/Engine/Source/Runtime/Engine/Classes/Engine/EngineTypes.h`
  - Information about types such as `ENetRole` and `ENetDormancy`.