Object Replication

Learn how to replicate UObjects.

Actors are the primary class that replicate properties and events over a network connection in Unreal Engine (UE). However, you might want to replicate a more lightweight `UObject`-derived class in your gameplay code. UE can replicate these objects as subobjects of an owning actor or actor component.

Replicating Objects as Actor Subobjects

A *subobject* of a class is an object that is accessed through an owning object, most often an actor or actor component. You can create a subobject statically in the owning object's constructor with `CreateDefaultSubobject<T>` or dynamically at runtime with `NewObject<T>`. For more information about object creation, see the Objects documentation.

There are two different ways to replicate subobjects in UE:

- Use the registered subobjects list through the function `AddReplicatedSubObject`.
- Implement the virtual `ReplicateSubobjects` function.

The registered subobjects list is a new method that aims to reduce the number of virtual function calls and is the only method compatible with the Iris replication system.

The virtual `ReplicateSubobjects` function is a backward-compatible method that requires you to maintain subobject registration through the `ReplicateSubobjects` function.

The method you should use depends in part on the replication system you are using. The following table outlines which method each of UE's replication systems support:

Replication System	Replicate Subobjects Function	Registered Subobjects List
Generic Replication System	✓	✓
Replication Graph	✓	✓
Iris Replication System		✓

Unreal Engine will trigger an ensure statement if you are not using the registered subobjects list (`net.SubObjects.DefaultUseSubObjectReplicationList=0`) while Iris is enabled (`net.Iris.UseIrisReplication=1`) .

Default Versus Dynamic

There are some differences between replicating a default subobject and replicating a subobject created dynamically at runtime. The following two sections explain these differences.

Default Subobject Replication

You create a default subobject in the owning object's constructor with `CreateDefaultSubobject<T>`. The owning actor on the server and client both create their own instances of the default subobject when the actor is spawned.

You do not need to replicate a reference to the subobject since the object exists on both the server and the client.

The subobject starts replicating its properties when you call `ReplicateSubobject` or `AddReplicatedSubObject` on the object and it is registered. You can still reference the object over the network even if the subobject is not replicating its properties yet since it is stably named relative to its outer object.

Dynamic Subobject Replication

You create a dynamic subobject at runtime in your server gameplay code with `NewObject<T>`.

The owning actor on the server uses a replicated object reference to the subobject. This object reference is valid on the server since it creates the object, but the reference is null on the client. The replicated reference remains null until the object is created and the reference is mapped.

The subobject starts replicating its properties when you call `ReplicateSubobject` or `AddReplicatedSubObject`on the object and it is registered. When the server replicates the subobject in either of these two ways with `ReplicateSubObject` or `AddReplicatedSubObject` and the client receives the subobject's bunch, the client automatically creates the object and reads in its replicated properties. At this point, the replicated reference is now valid and properties replicate as normal between the server and client copies of the replicated subobject. Additionally, if there are any unmapped references to this object, they update to point to this new, valid object.

For more information, see the Replicate Object References documentation.

Required Setup

Regardless of the method you use to replicate actor subobjects, there are several steps to replicate objects that both of these methods have in common.

The `AActor`-derived and `UObject`-derived classes contained in the Resources section of this page serve as the assumed starting point for the following guides. This guide and the resources assume you want to replicate an object `UMyObject` defined in the file `MyObject.h` and an actor `AMyActor` defined in the file `MyActor.h`. To prepare your actor and object for replication, follow these steps:

1. In your `MyObject.h` file, add the following:

- Define the `GetLifetimeReplicatedProps` virtual function override.
    - This function is where you record the properties of your object class that you want to replicate.

```
virtual void GetLifetimeReplicatedProps(TArray<FLifetimeProperty>
```

Copy full snippet

- Define and implement the `IsSupportedForNetworking` virtual function override:
    - This function informs the replication system that references to this object class are supported for replication.

```
virtual bool IsSupportedForNetworking() const override { retu
```

Copy full snippet

2. In your `MyObject.cpp` file, add the following:
    - Include the file `Net/UnrealNetwork.h`:
        - This file includes all the `DOREPLIFETIME` macros for property replication.

```
#include "Net/UnrealNetwork.h"
```

Copy full snippet

- Implement the `GetLifetimeReplicatedProps` function and add your replicated property:
    - In this example, the `ReplicatedValue` is declared as a replicated property. Add any other properties, including `OnRep`, here that you want to replicate.

```
void UMyObject::GetLifetimeReplicatedProps(TArray<FLifetimePr
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    // Add any replicated properties here
    DOREPLIFETIME(UMyObject, ReplicatedValue)
}
```

Copy full snippet

3. In your `MyActor.h` file, add the following:
   - Forward-declare your `UMyObject` class.
   - Add a `UMyObject` pointer as a replicated property:
     - In this case, the owning actor uses a replicated object reference and the object is created dynamically at runtime.

```
public:
    /** Replicated Object Reference to subobject
    * Not necessary if MyActorSubobject is a Default Subobjec
    */
    UPROPERTY(Replicated)
    TObjectPtr<UMyObject> MyActorSubobject;
```

Copy full snippet

   - Override the `GetLifetimeReplicatedProps` function:
     - This function is where you record the properties of this class that you want to replicate.

```
virtual void GetLifetimeReplicatedProps(TArray<FLifetimeProps
```

Copy full snippet

4. In your `MyActor.cpp` file, add the following:
   - Include your `MyObject` file:

```
#include "MyObject.h"
```

Copy full snippet

   - Set your actor to replicate in its constructor with `bReplicates = true;`.
   - Implement the `GetLifetimeReplicatedProps` function:
     - This function is where you replicate a reference to your object class.

```
void AMyActor::GetLifetimeReplicatedProps(TArray<FLifetimePro
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    // Add replicated properties here
    // Not necessary if MyActorSubobject is a default subobje
    DOREPLIFETIME(AMyActor, MyActorSubobject)
}
```

Copy full snippet

- Create your object as a default subobject in your actor's constructor or at runtime in your gameplay code as described in the Default Versus Dynamic section.

For more information about replicated properties, `GetLifetimeReplicatedProps`, and `DOREPLIFETIME` replication macros, see the Replicate Actor Properties documentation.

At this point, even though the actor contains a replicated object reference, the server does not replicate the object to connected clients since the object is not registered with either `ReplicateSubobject` or `AddReplicatedSubObject`.

Registered Subobjects List

The registered subobjects list is a newer method to replicate actor subobjects. The registered subobjects list aims to reduce the number of virtual function calls in the engine. Instead of adding a virtual function call for each subobject, you must maintain a list of subobjects on your actor or actor component.

To use the registered subobjects list rather than the `ReplicateSubobjects` virtual function, set `bReplicateUsingRegisteredSubObjectList` to true in your actor's or actor component's constructor.

The following section outlines the steps to replicate actor subobjects with the registered subobjects list using the same MyActor and MyObject classes as in the Replicate Subobjects Function section.

Steps to Use Registered Subobjects List

To replicate an instance of MyObject as a subobject of MyActor with the registered subobject list, follow these steps:

1. In your `MyActor.cpp` file, add the following:
    - In your actor's constructor, set your actor to use the registered subobject list with `bReplicateUsingRegisteredSubObjectList = true;`.
    - Register your `MyActorSubobject` after it is created:
        - Subobject registration occurs with the `AddReplicatedSubObject` function, and you must remove the subobject with `RemoveReplicatedSubObject` before you delete it.

```
if(IsValid(MyActorSubobject))
{
    RemoveReplicatedSubObject(MyActorSubobject);
}

MyActorSubobject = NewObject<UMyObject>();
MyActorSubobject->ReplicatedValue = 1;

// MyActorSubobject becomes a replicated subobject here
AddReplicatedSubObject(MyActorSubobject);
```

Copy full snippet

Modifying or Deleting a Subobject

Whenever you modify or delete your subobject, make sure to call `AActor::RemoveReplicatedSubObject`. Unless you remove the reference to your subobject, the registered subobject list contains a raw pointer to the subobject even if it is modified or marked for destruction. This causes a crash after the garbage collector purges the subobject.

Push Model

The registered subobject list does not support `UActorChannel::KeyNeedsToReplicate`. Instead, we recommend you use push-model replication for replicated subobject properties when you are using the registered subobject list.

Additional Steps for Iris

Iris does not support the virtual `ReplicateSubobjects` function and requires you to enable the registered subobjects list.

If you are using Iris and you want to replicate an object not derived from `AActor` or `UActorComponent`, you must also implement the virtual `RegisterReplicationFragments` function to register their replicated properties and RPCs.

To implement `RegisterReplicationFragments` for your `UObject`-derived class, follow these steps:

1. In your `MyObject.h` file, add the following:
   - Include the Iris `ReplicationFragmentUtil` file:

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

     Copy full snippet

   - Override the virtual `RegisterReplicationFragments` function:

     ```
     #if UE_WITH_IRIS
     virtual void RegisterReplicationFragments(UE::Net::FFragmentRegistrat
     #endif // UE_WITH_IRIS
     ```

     Copy full snippet

2. In your `MyObject.cpp` file, implement the `RegisterReplicationFragments` function:

   ```
   #if UE_WITH_IRIS
   void UMyObject::RegisterReplicationFragments(UE::Net::FFragmentRegistr
   {
       // Build descriptors and allocate PropertyReplicaitonFragments for
       UE::Net::FReplicationFragmentUtil::CreateAndRegisterFragmentsForOb
   }
   ```

```
#endif // UE_WITH_IRIS
```

Copy full snippet

You must implement `RegisterReplicationFragments` for any object interfacing with the Iris replication system. Any class inherited from `AActor` implements this function automatically, but any object not inherited from `AActor` must manually implement this function.

Replicate Subobjects Function

To replicate an actor subobject with the `ReplicateSubobjects` function, you must override the virtual `ReplicateSubobjects` function in your actor class to explicitly replicate your subobject.

Steps to Use Replicate Subobjects

To replicate an instance of MyObject as a subobject of MyActor with the virtual `ReplicateSubobjects` function, follow these steps:

1. In your `MyActor.h` file, add the following, override the `ReplicateSubobjects` virtual function:

```
virtual bool ReplicateSubobjects(UActorChannel* Channel, FOutBunch* Bu
```

Copy full snippet

2. In your `MyActor.cpp` file, add the following:
   - Include `Engine/ActorChannel`:
     - This file contains the `ReplicateSubobject` function.

```
#include "Engine/ActorChannel.h"
```

Copy full snippet

- Create your object:

```
MyActorSubobject = NewObject<UMyObject>();
MyActorSubobject->ReplicatedValue = 1;
```

Copy full snippet

- Implement the `ReplicateSubobjects` function:
    - Inside this function is where your object becomes a replicated subobject and the pointer becomes valid.

```
bool AMyActor::ReplicateSubobjects(UActorChannel* Channel, FO
{
    bool bWroteSomething = Super::ReplicateSubobjects(Channel
    // MyActorSubobject becomes a replicated subobject here

    if (IsValid(MyActorSubobject))
    {
        bWroteSomething |= Channel->ReplicateSubobject(MyActo
    }

    // Add any other replicated subobjects here in a similar

    return bWroteSomething;
}
```

Copy full snippet

When you spawn an instance of `MyActor` on the server, the actor replicates to all clients, including the `MyActorSubobject` and its replicated fields. This process works similarly if you want to replicate an actor component.

If you want to construct your subobject directly in your actor's constructor, use `AActor::CreateDefaultSubobject` instead of `AActor::NewObject`. To learn more about object creation in UE, see the UObject Creation documentation.

Complex Replication Conditions

You can specify custom replication conditions for replicated subobjects with the `FNetConditionGroupManager` and the condition `COND_NetGroup`. You can include an object in multiple groups concurrently. If an object is part of multiple groups, the object replicates to a particular client if the object is part of one of the client's groups.

Create and Use a Replication Group

To create and use a replication group for replicated subobjects, follow these steps:

1. Register your subobject with the `COND_NetGroup` replication condition:

   ```
   AddReplicatedSubObject(MyActorSubobject, COND_NetGroup);
   ```

   Copy full snippet

2. Create an `FName` to represent the replication group:

   ```
   FName NetGroupAlpha(TEXT("NetGroup_Alpha"));
   ```

   Copy full snippet

3. Add your subobject to the replication group:

   ```
   FNetConditionGroupManager::RegisterSubObjectInGroup(MyActorSubobject,
   ```

   Copy full snippet

4. Add clients you want to replicate the subobject to in the same group through the client's player controller:

   ```
   PlayerController->IncludeInNetConditionGroup(NetGroupAlpha);
   ```

   Copy full snippet

Object Remote Procedure Calls

Replicated subobjects do not support calling Remote Procedure Calls (RPCs) by default. In order to support calling RPCs from an actor-owned object, you need to route the RPC through its owning actor by overriding the following functions in your `UObject`-derived class:

- `UObject::GetFunctionCallspace`
- `UObject::CallRemoteFunction`

There are example implementations of `UObject` classes that implement these functions in `..\Engine\Plugins\Runtime\ReplicationSystemTestPlugin\Source\Private\Tests\ReplicationSystem\RPC`.

After you implement both `GetFunctionCallspace` and `CallRemoteFunction`, you can add any RPCs you wish to your `UObject`-derived class and call them through your object just as you would call an RPC through an actor.

For more information about RPCs, see the Remote Procedure Calls documentation.

Actor Component Subobjects

Actor components can also have their own replicated subobjects and use the registered subobjects list. Actor components use the same API as actors for registering and unregistering replicated subobjects. Subobjects within an actor component can have custom replication conditions in the same way as subobjects within an actor.

To replicate an object as a subobject of an actor component, the owning actor component and actor must replicate before the conditions of the component's replicated subobjects are checked. For example, if a subobject has the `COND_OwnerOnly` condition, then it will not replicate if it is registered to an actor component that uses the `COND_SkipOwner` replication condition.

Server and Client Subobject Lists

If you use the registered subobjects list, the server must maintain a copy of the registered subobjects list. We also recommend that you maintain a registered subobject list locally for all actors and actor components on client machines. This is particularly important if your project records replays on clients as actors on clients temporarily switch to a local authority role when recording the actor into a replay. As a result, any actors that record for a replay should maintain their subobject list on the client regardless of their local network role.

If the subobject is a replicated property, managing the subobject list on client machines is easier if you specify the subobject as a replicated using property with an associated replication notification. Clients use the replication notification to identify when the subobject has changed so you can remove the old subobject pointer and add a new pointer to the updated subobject.

Removing a subobject from the registered subobject list on the server results in none of the object's replicated properties being sent to client machines. However, the subobject pointer is still net-referenceable until the garbage collector marks the object as garbage. Once the server detects that the object is invalid, the server notifies clients to delete the subobject locally on the next update.

Debugging

You can use the following console variables to debug your use of subobjects in your replicated actors:

Console Variable	Description	Default Value
`net.SubObjects.CompareWithLegacy`	If enabled, collect the subobjects replicated with the `ReplicateSubobjects` method and compare them with the ones replicated via the actor's registered subobjects list. If a divergence between the two is detected, an ensure is triggered.	false: disabled (default), true: enabled
`net.SubObjects.DefaultUseSubObjectReplicationList`	If enabled, actors and actor components replicate subobjects with the registered subobject list by default.	false: disabled (default), true: enabled

Console Variable	Description	Default Value
`net.SubObjects.DetectDeprecatedReplicatedSubObjects`	If enabled, trigger an ensure statement if a `ReplicateSubObjects` function is still implemented in a class that is using the registered subobject list.	false: disabled (default), true: enabled
`net.SubObjects.LogAllComparisonErrors`	If enabled, log all errors detected by `net.SubObjects.CompareWithLegacy`. If disabled, only the first ensure triggered is logged.	0: disabled (default), 1: enabled

For a list of all networking console variables and commands, see the Console Commands for Network Debugging documentation.

Resources

Classes

MyActor

```
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "MyActor.generated.h"

UCLASS()
class SUBOBJECTS_API AMyActor : public AActor
{
  GENERATED_BODY()
public:

  // Sets default values for this actor's properties
  AMyActor();

  // Called every frame
  virtual void Tick(float DeltaTime) override;

protected:

  // Called when the game starts or when spawned
  virtual void BeginPlay() override;
};
```

Copy full snippet

```
// Fill out your copyright notice in the Description page of Project Settings.

#include "MyActor.h"
#include "MyObject.h"
#include "Engine/ActorChannel.h"

// Sets default values
AMyActor::AMyActor()
{
  // Set this actor to call Tick() every frame.  You can turn this off to improv
  PrimaryActorTick.bCanEverTick = true;
}

// Called when the game starts or when spawned
void AMyActor::BeginPlay()
{
  Super::BeginPlay();
}

// Called every frame
void AMyActor::Tick(float DeltaTime)
{
  Super::Tick(DeltaTime);
}
```

Copy full snippet

MyObject

```
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "UObject/NoExportTypes.h"
#include "MyObject.generated.h"

/**
*
*/
UCLASS()
class SUBOBJECTS_API UMyObject : public UObject
{
  GENERATED_BODY()

public:

  UMyObject();

  // Add replicated properties anywhere here
  UPROPERTY(Replicated)
  int32 ReplicatedValue = 0;
};
```

Copy full snippet

```
// Fill out your copyright notice in the Description page of Project Settings.

#include "MyObject.h"

UMyObject::UMyObject()
{
  }
```

Copy full snippet