

Replicated Object Execution Order

Execution order guarantees for replicated properties and remote procedure calls on receiving machines.



Unreal Engine's (UE) network replication uses a combination of reliable and unreliable communication methods to transfer information between servers and connected clients. *Reliable* communication is continually sent, halting all other network communication, until the receiving machine acknowledges it. *Unreliable* communication is sent and not resent during the current network tick if the receiving machine does not acknowledge receipt.

It is essential to understand what is guaranteed when it comes to actor property and remote procedure call (RPC) replication with respect to the relative ordering of this communication and what you can do to take this into account in your game code. This page describes what guarantees UE's replication system makes and, equally as important, does not make.

Actor Properties

Actor property updates are unreliable and sent in a single bunch. They can be thought of as a single, unreliable RPC sent after all other RPCs, but before queued RPCs. For more information about queued RPCs, see the [Force Queue](#) section of this page.

Replicated Using Order

There is no deterministic order between the OnRep (RepNotify) callbacks of different replicated variables. The call order on the client has no relation with a variable marked dirty or its declaration location in memory. If you need a reliable order between multiple variables, we recommend that you store them together in a struct.

If the order of an actor's property replication is important to your game, you might need to implement OnReps to track per-frame property updates. After the replicated values are received and their OnReps called, you can handle the changes in the

`UObject::PostRepNotifies` function. You might also need to save certain received values in their respective OnReps until they are ready to be used.

Remote Procedure Calls

The replication system in Unreal Engine executes RPCs as reliably as possible, so gameplay systems can be built without worrying about networking side effects.

Order Across Actors

There is no mechanism in which the original call order of RPCs across multiple actors is preserved and reapplied on a remote machine. Consider the following example of RPC call order on the sending machine:

```
1 AActor* MyActor;  
2 AActor* OtherActor;  
3  
4 // Valid MyActor pointer  
5 MyActor->ClientRPC1();  
6 OtherActor->ClientRPC2();  
7 MyActor->ClientRPC3();
```

 Copy full snippet

In this example, the execution order of the RPCs on the receiving machine is *not* deterministic, and the RPCs can execute in any possible combination on the receiving

machine:

```
1  RPC1 --> RPC2 --> RPC3
2  RPC1 --> RPC3 --> RPC2
3  RPC2 --> RPC1 --> RPC3
4  RPC2 --> RPC3 --> RPC1
5  RPC3 --> RPC1 --> RPC2
6  RPC3 --> RPC2 --> RPC1
```

 Copy full snippet

Order Inside an Actor

The replication system does guarantee the call order of reliable RPCs on the same actor. The order in which they are executed on the receiving machine is the same order in which they are called on the sending machine. If the call order on the sending machine is:

```
1  AActor* MyActor;
2
3  // Valid MyActor pointer
4  MyActor->ClientReliableRPC1();
5  MyActor->ClientReliableRPC2();
6  MyActor->ClientReliableRPC3();
```

 Copy full snippet

then the receiving machine always executes the RPCs in this order:

```
RPC1 --> RPC2 --> RPC3
```

 Copy full snippet

Order Between Actor and Subobjects

The order of RPC execution on the receiving machine is respected for all RPCs called on an actor and its subobjects. For example, if the sending machine sends:

```
1 AActor* MyActor;
2
3 // Valid MyActor pointer
4 MyActor->RPC1();
5 MyActor->SubObject1->RPC2();
6 MyActor->SubObject2->RPC3();
7 MyActor->RPC4();
```

 Copy full snippet

the execution order on the receiving machine is:

```
RPC1 --> RPC2 --> RPC3 --> RPC4
```

 Copy full snippet

Unreliable Versus Reliable Ordering

The order of RPC execution between unreliable and reliable RPCs can seem preserved, but this is never guaranteed. When no packet loss or packet reordering occurs, the execution order between unreliable and reliable is the same on the receiving machine as on the sending machine. Consider the following example of RPC call order on the sending machine:

```
1 AActor* MyActor;
2
3 // Valid MyActor pointers
4 MyActor->ClientReliableRPC1();
5 MyActor->ClientUnicastUnreliableRPC2();
6 MyActor->ClientReliableRPC3();
```

 Copy full snippet

It is possible that if no packet loss or reordering occurs, the receiving machine executes the RPCs in this order:

```
RPC1 --> RPC2 --> RPC3
```

 Copy full snippet

If `RPC1` is in an individual packet that is dropped or reordered, the receiving machine executes:

```
RPC2 --> RPC1 --> RPC3
```

 Copy full snippet

If `RPC2` is in an individual packet that is dropped, the receiving machine executes:

```
RPC1 --> RPC3
```

 Copy full snippet

In this last case, `RPC2` is dropped and never executed on the receiving machine since it is unreliable.



There should be no scenario in which unreliable `RPC2` is executed after `RPC3`. If the packet containing `RPC2` is reordered and arrives later than `RPC3`, it is ignored on reception.

Multicast Versus Unicast Ordering

Multicast RPC ordering is more complicated since UE's replication system does not always preserve the call order between multicast RPCs and unicast RPCs.

Multicast is Reliable

The call order between reliable multicasts and other reliable unicast RPCs is preserved. For example, if the following functions are called in this order on the sending machine:

```
1 MyActor->MulticastReliableRPC1();  
2 MyActor->UnicastReliableRPC2();
```

```
3 MyActor->UnicastReliableRPC3();  
4 MyActor->MulticastReliableRPC4();
```

 Copy full snippet

then the receiving machine or machines execute the RPCs in this order:

```
RPC1 --> RPC2 --> RPC3 --> RPC4
```

 Copy full snippet

Remember that the ordering of unreliable **RPC3** is not deterministic and it could be executed earlier or not at all.

Multicast is Unreliable

Unreliable multicasts never preserve their call order between other unicasts and reliable multicasts. For example, if the following RPCs are called in this order on the sending machine:

```
1 MyActor->MulticastUnreliableRPC1();  
2 MyActor->UnicastReliableRPC2();  
3 MyActor->MulticastUnreliableRPC3();  
4 MyActor->UnicastUnreliableRPC4();
```

 Copy full snippet

then the receiving machine or machines execute the RPCs in this order:

```
RPC2 --> RPC4 --> RPC1 --> RPC3
```

 Copy full snippet

RPC1 and **RPC3** are queued and serialized last because they are unreliable multicast RPCs. This means unicasts are executed first, and unreliable multicasts are executed last. The rules governing dropped, unreliable unicast RPCs also apply here.

If **RPC2** is in an individual packet that is dropped or reordered, the receiving machine executes the RPCs in the following order:



```
RPC1 --> RPC3 --> RPC2 --> RPC4
```

Copy full snippet

RPC Send Policy

It is possible to assign RPCs an explicit send policy that affects the ordering of RPCs. This can be done by specifying an `ERemoteFunctionSendPolicy`. For more information about RPC send policies, see the [Remote Procedure Call](#) documentation.

Force Send

An RPC with the `ERemoteFunctionSendPolicy::ForceSend` policy changes the order of unreliable multicast RPCs and prevents them from being queued. The following is an example:

```
1 MyActor->ForceSendMulticastUnreliableRPC1();  
2 MyActor->UnicastReliableRPC2();  
3 MyActor->MulticastUnreliableRPC3();  
4 MyActor->UnicastUnreliableRPC4();
```

Copy full snippet

The clients execute these RPCs in the following order:

```
RPC1 --> RPC2 --> RPC4 --> RPC3
```

Copy full snippet

Force Queue


An RPC with the `ERemoteFunctionSendPolicy::ForceQueue` policy does not respect the call order except with other `ForceQueue` RPCs and unreliable multicasts. The following is an example:

```
1 MyActor->ForceQueueRPC1();
2 MyActor->UnicastReliableRPC2();
3 MyActor->MulticastUnreliableRPC3();
4 MyActor->UnicastUnreliableRPC4();
```

 Copy full snippet

The clients execute these RPCs in the following order:

```
RPC2 --> RPC4 --> RPC1 --> RPC3
```

 Copy full snippet

Order Between RPCs and Actor Properties

It is also important to understand the order between RPC executions and when replicated property updates are applied. In this case, the following rules apply:

- RPCs are executed first.
- Properties are updated second.
- Property updates are sent as a single, unreliable data block.

The bunch payload is constructed as follows:

- Non-queued RPCs are serialized.
- Replicated property data is serialized.
- Queued RPCs are serialized.




It is possible that a replicated variable written from inside an RPC might get lost and overwritten immediately by unprocessed property updates.



An exception to this rule is unreliable multicast RPCs since they are queued at the call site and always serialized last. This means they are executed *after* property updates are applied.

The following is an example:


```
1 MyActor->ReliableRPC1();
2 MyActor->bReplicatedVar1 = true
3 MyActor->MulticastUnreliableRPC2();
4 MyActor->bReplicatedVar2 = true;
5 MyActor->ReliableRPC3();
```

 Copy full snippet

The remote machine or machines execute these in the following order:

```
RPC1 --> RPC3 --> Var1 && Var2 --> RPC2
```

 Copy full snippet

The following is another example of property updates mixed with RPCs:

```
1 MyActor->ReliableRPC1();
2 MyActor->bReplicatedVar1 = true
3 MyActor->MulticastUnreliableRPC2();
4 MyActor->bReplicatedVar2 = true;
5 MyActor->ReliableRPC3();
```

 Copy full snippet

Suppose that the property update is dropped, then the receiving machine or machines execute the RPCs and property updates in the following order:

```
1 RPC1 --> RPC3 --> RPC2
2 // After the next update
3 Var1 && Var2
```

 Copy full snippet

Another scenario, where only the reliable RPC1 is dropped, results in the following execution order on the receiving machine or machines:

```
Var1 && Var2 --> RPC2 --> RPC1 --> RPC3
```

 Copy full snippet

Test Gameplay Code with Unreliable RPCs

If you are creating or relying on code that is replicated using unreliable RPCs, it is a good idea to force them to be dropped and see how your systems react. For more information about how to do this by emulating poor network conditions, see the [Using Network Emulation](#) documentation.