Lab 7 - RPCs

COMP280-Multiplayer Game Programming

Purpose: Using RPCs

Contents

1. Introduction to Using Remote Procedure Calls (RPCs)

Let's see how **functions** can be called across the **network**. We need to use **Remote Procedure Calls (RPCs)** – one of the most powerful features of the UE networking system. Let's see:

- how to execute functions through RPCs and run them
  - on the server
  - on a client, or
  - on all clients that have an instance of a particular object.
- the common requirements for properly calling these types of functions
  - reliable (TCP), and
  - unreliable (UDP).
- how to apply this knowledge to our project

Specifically we'll deal with these topics:

- Understanding what an RPC is
- Executing RPCs over the network
- Implementing a door system

1.1. Technical requirements

- We are going to use **Unreal Engine 5.4**.
- You have to have finished the **Lab 6**.

1.2. Understanding what an RPC is

Definition - An **RPC** is a **function** that can be **called locally** but is **executed remotely** on one or more connected machines.

For example, a **server** computer may call a function to be executed on a **client** computer that will command it to **spawn** a *visual effect* or a *sound* in the level. Other application of RPCs: **sending messages** bi-directionally between a server and a client over a network connection (chatting).

There are four **types** of RPCs available in Unreal Engine:

- **Server:** Executed on the server; called from the client that owns the Actor
- **Client:** The function is called on the server by an object but executed only on the client version that owns the object calling it.
- **Remote:** The function is executed on the remote side of the connection.
- **NetMulticast:** The function is called:
    - on the server by an object and executed on the server and all connected clients with a relevant actor
    - by a client; it will only be executed locally (i.e., on the client calling it).

To recap:

RPC Type	Local call	Remote execution	Owner
Server	Client	Server	Object in the client
Client	Server	Client	Object in the client
Remote	Server/Client	Client/Sever	Object in the client
NetMulticast	Server	Client(s) + Server	Object in a relevant actor
NetMulticast	Client	No (only local)	Object in the client

Conditions for a function to be properly executed as an RPC:

- Must be called by an **Actor** and
- The Actor must be **replicated**.
- Needs to be decorated with the **UFUNCTION(x,[y])** macro where **x** is one of (**y** is optional and you'll see examples later) :
    - Client

- Server
- Remote, or
- NetMulticast

A function that will run just on the owning client will be declared in the **.h** file as below:

```
UFUNCTION(Client)
void DoSomething_Client();
```

In the corresponding **.cpp** file of the previous header, we will need to implement this function with the **_Implementation** suffix. The autogenerated code for this class – located in the **.generated.h** file – will automatically include a call to the _Implementation method when necessary.

> The suffix **_Client** above is only a UE convention, but an important one. Also valid for other RPC types (**_Server**, **_Remote** and **_NetMulticast**). On the other hand, the suffix **_Implementation** is a required one in the **.cpp** file.

Example, let's say your **.h** header file has a method declaration similar to the following piece of code (function to be executed in the Server):

```
UFUNCTION(Server)
void DoSomething_Server();
```

We will need to implement the following function in our **.cpp** file:

```
void DoSomething_Server_Implementation()
{ /* The code here */ }
```

A method is not always guaranteed to be received by the recipient due to performance reasons; however, this behavior can be adjusted via **Reliability** settings.

1.2.1 Reliability of an RPC

RPCs are **unreliable** by default (use UDP by default) – this means there is no guarantee that the function call will reach its destination. This is usually acceptable if the executed code is not so important, such as a visual effect spawned on a client or a random noise played near the character; if the message is not received, the effect will simply not spawn or the sound won't be heard, but gameplay will not be affected. There are, however, some cases where you want to **enforce reliability** and **guarantee** that a message will securely arrive at its destination – as an example, later hereby, we will execute the **sprint action** from the server side in a reliable way (using TCP). To make sure that an RPC call **is executed** on the remote machine, you can utilize the **Reliable** keyword.

Example of a function that should be executed reliably on a client; declare as follows:

```
UFUNCTION(Client, Reliable)
void DoSomethingReliably_Client();
```

This will guarantee that the method call will be received by the client and properly executed, without any risk of data loss over the network because of unreliability.

> Avoid using reliable RPCs during the Tick() event and exercise caution when binding them to player input. This is because players can repeatedly press buttons very quickly, leading to an overflow of the queue for reliable RPCs.

In addition to reliability, you may want a method to be validated in order to be executed – this is something I'm going to show you right now!

1.2.2 Validating RPCs

Unreal Engine offers an additional feature that adds the ability to check that functions will execute without bad data or input – this is what validation is all about. To declare that a method should be validated for an RPC call, you need to add the WithValidation specifier to the UFUNCTION() declaration statement and implement an additional function that will return a bool type and be named as the validated function, but with the _Validate suffix. As an example, a method marked with validation will have a declaration in the .h file, similar to the following code:

```
UFUNCTION(Server, WithValidation)
void DoSomethingWithValidation();
```

Then, in the .cpp file, you will need to implement two methods. The first one will be the regular function, and will look like the following code:

```
void DoSomethingWithValidation_Implementation()
{ /* Your code here */ }
```

The second one will be the actual validation function, and will look like the following code:

```
bool DoSomethingWithValidation_Validate()
{ /* Your code here */ }
```

The _Validate function will return true if the code is validated, or false otherwise. If validation succeeds, the corresponding method will be executed; otherwise, it won't. In this section, I have introduced RPCs and how Unreal Engine copes with them. Bear with me – if you're working in the networked games industry, mastering RPCs is key to keeping your job and growing your career! Now that you have a solid understanding of how an RPC should be implemented, it's time to write some code – and we are going to start by hunting down that pesky little bug that is stopping our thief hero from sprinting freely (and correctly!) around the dungeon.

1.3. Executing RPCs over the network

In this section, you'll do some practice with RPCs by fixing the issue we are experiencing with making the character sprint correctly. As you may remember, when the character is sprinting on the client, you will get "jumpy" behavior – the character seems to start running but it is immediately brought back to a walking speed. This happens because the sprint action is being executed on the player client, but it is not being executed on the server, which is the one that is in command; hence, the override from the server slows the character to its move speed on every update. This means that you are trying to move your character at a sprint speed but, as soon as the server replicates the movement on the client, it will bring the character back to moving speed. We don't even want the client to control this kind of important interaction – remember that it is the server who is in command – so, get back to the project and start typing some code to fix this problem!

1.3.1 Calling a function on the server

To make our character run, we simply have to execute the movement speed change on the server, instead of on the client. This will guarantee total control over the behavior and correct replication on all clients. Let's start by opening the **US_Character.h** file and do some code declarations. In the protected section, add these two methods:

```
UFUNCTION(Server, Reliable)
void SprintStart_Server();
```

```
UFUNCTION(Server, Reliable)
void SprintEnd_Server();
```

These functions have the Server attribute, which, as explained in the previous section, will execute them on the server. We have also added the Reliable attribute because we don't want to lose this RPC due to the default unreliability of the system. The _Server suffix is not mandatory and is written just for clarity (some people use a prefix, so it is up to your personal taste!). Now open the **US_Character.cpp** file and implement the two functions by adding the following code:

```
void AUS_Character::SprintStart_Server_Implementation()
{
 if (GetCharacterStats())
 {
  GetCharacterMovement()->MaxWalkSpeed = GetCharacterStats()->SprintSpeed;
 }
}
void AUS_Character::SprintEnd_Server_Implementation()
{
 if (GetCharacterStats())
 {
  GetCharacterMovement()->MaxWalkSpeed = GetCharacterStats()->WalkSpeed;
 }
}
```

The code is pretty straightforward as we are just executing the speed change inside these two new functions and, in just a moment, we are going to remove them from their previous positions (i.e., from the client calls). Here, just notice the _Implementation suffix – this is mandatory as the SprintStart_Server() and SprintEnd_Server() functions will be auto-generated by Unreal in the .generated.h class file and will be responsible for calling the actual implementations. We now need to change the SprintStart() and SprintEnd() functions, in order to call the corresponding server fuctions (i.e., SprintStart_Server() and SprintEnd_Server()). Find those two functions and remove all their content (i.e., the changes to MaxWalkSpeed) and then, in the SprintStart() function, add this simple line of code:

```
SprintStart_Server();
```

In the SprintEnd() function, add this line of code:

```
SprintEnd_Server();
```

To make the sprint action fully operational, we need to take one final step. At the moment, if the character is running and levels up, the movement will revert to walking speed. This happens because, in the UpdateCharacterStats() function, we set the MaxWalkSpeed property to the new walk speed, even if the character is sprinting. Let's fix this by finding the UpdateCharacterStats() method and adding, at its very beginning, the following code:

```
auto IsSprinting = false;
if(GetCharacterStats())
{
 IsSprinting = GetCharacterMovement()->MaxWalkSpeed == GetCharacterStats()->SprintSpeed;
}
```

This block of code just checks whether the character is sprinting and stores the result in a local variable. Then, find this line of code:

```
GetCharacterMovement()->MaxWalkSpeed = GetCharacterStats()->WalkSpeed;
```

Add the following command just after it:

```
if(IsSprinting)
{
 SprintStart_Server();
}
```

As easy as it is, if the character was sprinting, we would just call the corresponding method on the server to update everything properly. We're almost done with the movement management, but there are still a few small things we need to work on. Don't worry though, we're working hard to get everything done by the end of Chapter 10, Enhancing the Player Experience. So sit tight and stay tuned! In this section, you have started implementing a simple RPC in your Character class. In particular, you have sent a command from the client that owns the character to the server, in order to properly update the movement speed. In the next section, you'll add some more fancy RPCs for your game. Specifically, you'll develop a nifty door-opening system. Get ready to flex those programming skills!

1.4. Implementing a door system

In this section, you'll repeat some of the previously explained topics about RPCs but with a small tweak – you'll be developing some Actor-to-Actor communication over the network. What's more, it will be between a C++ class – your character – and a Blueprint Class, a door that should be opened. To accomplish this behavior, you will use a feature that you previously created in Chapter 4, Setting Up Your First Multiplayer Environment – the interact action. It may have slipped your mind with all the stuff you have developed so far, but fear not – it's time to dust it off and put it to work once again.

1.4.1 Creating the Interactable interface

To create communication between your character and the door, you'll use an interface. As you may already know, interfaces in C++ are a powerful tool for creating abstractions between different classes. They allow you to define a contract that all implementing classes must adhere to, thereby allowing you to create code that is more maintainable, extensible, and reusable. In Unreal Engine, interfaces differ from traditional programming interfaces in that it is not mandatory to implement all functions. Instead, it is optional to implement them. What's more, you can declare an interface in C++ and implement it in a Blueprint – and that's exactly what you'll be doing here. Let's start by opening your development IDE and creating a file named US_Interactable.h. Then, add the following code to the file:

```
#pragma once
#include "CoreMinimal.h"
#include "UObject/Interface.h"
#include "US_Interactable.generated.h"
UINTERFACE(MinimalAPI, Blueprintable)
class UUS_Interactable : public UInterface
{
 GENERATED_BODY()
};
class UNREALSHADOWS_LOTL_API IUS_Interactable
{
 GENERATED_BODY()
public:
 UFUNCTION(BlueprintCallable, BlueprintNativeEvent, Category = "Interaction", meta=
(DisplayName="Interact"))
 void Interact(class AUS_Character* CharacterInstigator);
 UFUNCTION(BlueprintCallable, BlueprintNativeEvent, Category = "Interaction", meta=(DisplayName="Can
Interact"))
 bool CanInteract( AUS_Character * CharacterInstigator) const;
};
```

You may notice something weird in the code you just added – there are two classes. In order to properly declare an Unreal Engine interface, you need to declare two classes:

- A class with a U prefix and extending UInterface: This is not the actual interface but an empty class whose sole aim is to make the class visible in the Unreal Engine system
- A class with an I prefix: This is the actual interface and will contain all the interface method definitions

As you can see, the U-prefixed class is decorated with the UINTERFACE() macro and the Blueprintable attribute will let you implement this interface from a Blueprint. Isn't it cool? Finally, we declare a couple of functions called Interact() and CanInteract(), respectively. The two of them can be called and implemented in a Blueprint (thanks to the BlueprintCallable and BlueprintNativeEvent attributes). Even though we will not be implementing the second function (i.e., CanInteract()) in our door Blueprint, it is nice to have such a feature – for instance, to check whether the character can open a door with a key that can be found somewhere in the dungeon. As I told you before, interfaces in Unreal Engine do not force implementation for all method declarations. So, you have created an interface to allow the character to... well, interact with something. It's time to let the thief character perform this heroic action – something you are going to implement in the next subsection.

1.4.2 Implementing the interact action

You are now ready to get back to the US_Character.h header class and add some code logic for the interact action. As we have already done for the character movement, we will need to execute this interaction from the server. To do so, open the header file and look for this declaration in the protected section:

```
void Interact(const FInputActionValue& Value);
```

Add the corresponding server call just after it:

```
UFUNCTION(Server, Reliable)
void Interact_Server();
```

As for the sprint action, this call must be Reliable as we need to make sure that it will be executed properly, and no information will be lost. As a last step, add the following line of code to the private section:

```
UPROPERTY()
AActor* InteractableActor;
```

You will be using this property as a reference to the object that should be interacted with. Now that the header has been properly updated, open US_Character.cpp and add the following includes at the start of the file:

```
#include "US_Interactable.h"
#include "Kismet/KismetSystemLibrary.h"
```

Then, look for the Interact() method that, up to now, has just been an empty shell. Inside the method, add the following:

```
Interact_Server();
```

This code performs a simple RPC to the corresponding server interaction implementation. As you obviously need to implement the server call, add it to your code, just after the Interact() function:

```
void AUS_Character::Interact_Server_Implementation()
{
 if(InteractableActor)
 {
  IUS_Interactable::Execute_Interact(InteractableActor, this);
 }
}
```

The call is executed only if a reference to the InteractableActor is found. If you come from an OOP background and are not familiar with the way interfaces work in Unreal, this call may seem pretty weird – we are performing the call to an Actor reference without any type checking! This is the way interfaces work in Unreal Engine; they are just messages that are sent to an object reference. If the object does not implement that interface, the call will simply be lost. Obviously, we want the call to be executed to

something that can be interacted with (i.e., that implements the US_Interactable interface). To achieve this, we are going to continuously check whether the character is pointing at anything that implements the interface and, if something is found, we will reference it in the InteractableActor property. Look for the Tick() method in your .cpp class and start adding the following piece of code:

```
if(GetLocalRole() != ROLE_Authority) return;
FHitResult HitResult;
FCollisionQueryParams QueryParams;
QueryParams.bTraceComplex = true;
QueryParams.AddIgnoredActor(this);
auto SphereRadius = 50.f;
auto StartLocation = GetActorLocation() + GetActorForwardVector() * 150.f;
auto EndLocation = StartLocation + GetActorForwardVector() * 500.f;
auto IsHit = UKismetSystemLibrary::SphereTraceSingle(
 GetWorld(), StartLocation, EndLocation, SphereRadius,
 UEngineTypes::ConvertToTraceType(ECC_WorldStatic),
 false, TArray<aactor*>(), EDrawDebugTrace::ForOneFrame, HitResult, true
);
```

The first thing we do here is to check whether the instance that is executing the trace has the authority to do so – this means only the server will perform the traces for all the characters, and it's obviously done to avoid cheating from the client side. Then, we perform a regular sphere trace to check whether the character is pointing at something. If you are not familiar with an Unreal Engine trace, it is a tool used to detect collisions and overlapping between objects. It is used for things such as line of sight, weapon fire, and even AI pathfinding. Traces can be configured using parameters such as collision channels, object type filtering, shapes, start/end points, and so on, which allows you to specify exactly what kind of collision should be detected and how it should interact with the environment. NOTE For more information about the inner workings of traces in Unreal Engine, you can check the official documentation, which can be found here: https://docs.unrealengine.com/5.1/en-US/traces-with-raycasts-in-unreal-engine/. After the trace, the result is stored in the HitResult variable, which we are going to use to check whether we have found an interactable Actor. To do so, add the next code just after the code you have just written:

```
if (IsHit && HitResult.GetActor()->GetClass()->ImplementsInterface(UUS_Interactable::StaticClass()))
{
 DrawDebugSphere(GetWorld(), HitResult.ImpactPoint, SphereRadius, 12, FColor::Magenta, false, 1.f);
 InteractableActor = HitResult.GetActor();
}
else
{
 InteractableActor = nullptr;
}
```

The previous check is the core of our interaction control – if the object that has been traced implements the US_Interactable interface, we store the reference and draw a magenta-colored debug sphere for testing purposes. If nothing is found, we just clean up the InteractableActor property from any previous reference. To check that everything works as expected, you can open Unreal Engine Editor and, after compiling, you can play the game. The server should now draw a red sphere trace for each character and turn green when hitting something. We still don't have anything to interact with, so you will not see the debug sphere. In the next subsection, you will implement a door Blueprint that will react to the character interaction. Creating the door Blueprint It's now time to create something that can be interacted with, and we will do this by adding some doors to the dungeon. So, let's open the Blueprints folder and complete the following steps:

1. Create a new Blueprint Class derived from Actor, name it BP_WoodenDoor, and open it.
2. In the Details panel, check the Replicates attribute to enable replication for this Actor.
3. Add a StaticMesh component and assign the door mesh to the Static Mesh property.
4. In the Components panel, select the StaticMesh component and, in the Details panel, check Component Replicates to enable replication.
5. *Take Snapshot*

Now, open the Event Graph and do the following:

5. Create a variable of type Boolean and call it DoorOpen. In its Details panel, set the Replication property to Replicated.
6. Select the Class Settings tab and, in the Interfaces category, add the US_Interactable interface. This will add the Interfaces section into the My Blueprint window.
7. In the Interfaces section of the My Blueprint tab, open the Interaction category, right-click on the Interact method, and select Implement Event. This will add an Event Interact node on the Event Graph.
8. From the outgoing pin of the event, add a Branch node and, in the Condition pin, add a getter node of the DoorOpen variable from the Variables section.
9. Connect the False pin of the Branch node to a Setter node for the Door Open variable and check this last node's incoming value pin to set it to True.

$Take\ Snapshot$ The graph at the moment is quite simple; it's just checking whether the door has already been opened and, if it is closed, marks it as open. You are going to complete the Blueprint by making the door mesh rotate in an opening animation. 10. Connect the outgoing pin of the Set Door Open node to a Timeline node. Call this node DoorOpening and double-click on it to open its corresponding graph. 11. Add a Float Track and call it RotationZ. Add two keys to the track with the values (0, 0) and (1, −90) respectively. $Take\ Snapshot$ 12. Return to the main Event Graph and drag a reference of the StaticMesh component from the Components panel into the graph itself. 13. Connect the outgoing pin of this reference to a Set Relative Rotation node. 14. Right-click on the New Rotation pin and select Split Struct Pin to expose the New Rotation Z value. 15. Connect the Update execution pin of the Timeline node to the incoming execution pin of the Set Relative Rotation node. Connect the Rotation Z pin with the New Rotation Z to complete the graph. $Take\ Snapshot$

This part of the graph will just start a rotation animation on the z axis of the mesh, making it open when interacted with. Let's now give the thief hero a moment to shine and roam the dungeon, eagerly opening doors to seek out prisoners to liberate and treasures to unearth!

1.4.3 Testing the interact action

Open your game level and drag an instance or two of the door Blueprint to start testing the game. Whenever the server-controlled sphere trace hits a door, you should be able to see a magenta-colored sphere, indicating that the object can be interacted with. Hitting the I key on the client will open the door and show the hidden treasures (or perils!) behind. $Take\ Snapshot$

So, the door system has finally been created and you are now free to put as many doors as you want inside the dungeon. As an additional exercise, you can create a Blueprint child class from BP_WoodenDoor and use the door_gate mesh to add some kind of variation to your level. In this final section, you have implemented a Blueprint that allows the character to interact with other Actors in the game. Specifically, you have created a door system that can be opened through player interaction and will be synchronized over the network. This means that every connected player will see the correct updates.

2. Summary

In this chapter, you were introduced to one of the most important and useful features of the Unreal Engine multiplayer environment, remote procedure calls, or RPCs. As you have seen, they allow you to execute functions from a server to a client and vice versa. In this chapter, you called requests from the client to the server, by improving the character sprint system and by adding interaction logic between the character and other Actors in the game (i.e., the dungeon doors). Rest assured that, by the end of

the book, you will also have seen other use cases for RPCs as they are quite ubiquitous in multiplayer games. This chapter ends the second part of this book – starting from the next chapter, you'll be working on implementing some AI logic over the network. Let's spice things up by rounding up those pesky Lichlord minions and giving our character a challenge to step up to!

2.0.1 Testing the game

To test the game, start playing it as a listen server and check that everything works fine. In particular, you should see the following behaviors:

- At the start of the game, the HUD should show 0 experience points and the character level equal to 1
- Every time a character picks a coin up, the HUD should update the total experience points
- If the target experience points are reached, the player should level up and the HUD will show the new level
- *Take Snapshot*

If everything goes according to plan, you're all set to embark on the next exciting chapter of the Lichlord multiplayer epic: client-server communication!

3. Summary

In this Lab:

4. Zip Project and Submit

- Click **File** → **Zip Project**
- Keep the default folder, select the name **US_LTOL_{YourInitials}_Lab7.zip**
- When the zipping finishes click on the provided link **Show in Explorer**
- Upload the zip in the corresponding folder (Lab7).