Lab 6 – Managing Actors, Replication Properties – Part 1 and Part 2

COMP280-Multiplayer Game Programming

Purpose: Managing Actors (continued), Replication Properties

Contents

1. Intro

We will:

- Manage Actors in a Multiplayer Environment
  - ~~Set up the character~~ DONE
  - ~~Add interactions (input actions, input mapping context)~~ DONE
  - Control the connection to an actor

    - Manage the Actor Relevancy and Authority
    - Add character stats in a stats data table
- Replicate Properties Over the Network
    - Enable property replication
    - Handle the level-up of the character
    - Add a HUD to the game

2. Part 1 - Managing Actors in a Multiplayer Environment

2.1. Technical requirements

- We are going to use Unreal Engine 5.4.
- You have to have finished the Lab5.

We'll tackle Actor's connection management and attributes that are relevant during a game session. The player character is now just an empty shell. Let's enrich it with adding/testing:

- ~~more **components**~~: DONE`
- ~~player **input** logic.~~ DONE
- **ownership** in a multiplayer environment
- **relevancy** in the level.

In this lab, the following is covered:

- ~~**Setting up** the character~~ DONE
- Controlling the **connection** of an Actor
- Understanding Actor **relevancy**
- Introducing **authority**

2.2. ~~Setting up the character~~ DONE

2.3. Controlling the connection of an Actor

Note that we are already playing a networked game even though you did not add any multiplayer code logic. **Why?** The **Character** class is already set to be replicated — just open the **BP_Character** Blueprint and look for the **Replication** category. You will find out that Replicate Movement has been set by default and also that the Replicates property is set to true.

Note that:

- the character on the server window moves and sprints **smoothly**
- the one on the client's window, moves a bit **jumpy** when running.

This happens because we are trying to execute the **sprint** action on the **client**, but the server is the one who is actually in command. So, the client will make the character move faster, but the server will bring it back to its move position. Basically, at the moment, we are trying to "cheat" on the client, but the server, which is **authoritative**, will forbid you from doing this. To deal with this we need to know more about **replication** and **authority** settings in the next Lab.

In Unreal Multiplayer games, each **connection** has its own **PlayerController** that has been created expressly for it; in this case, we say that the PlayerController is **"owned"** by that connection. In Unreal Engine, Actors can have an **Owner**: if the outermost Owner of an Actor is a **PlayerController**, then the PlayerController becomes the Owner of that Actor. This means that the first Actor is also **owned** by the same connection that owns the PlayerController. So ownership chain is:

Actor → PlayerController → Connection

The concept of **ownership** is used during Actor replication to determine which connections receive updates for each Actor: for instance, an Actor may be flagged so that only the connection that owns that Actor will be sent property updates for it.

As an example, let's imagine that your thief character (which is basically an Actor) is possessed by a PlayerController – this PlayerController will be the Owner of the character. During gameplay, the thief gets a pickup that grants a magical dagger: once equipped, this weapon will be owned by the Character. This means that the PlayerController will also own the dagger. In the end, both the thief Actor and the dagger will be owned by the PlayerController connection. As soon as the thief Actor is no longer possessed by the Player Controller, it will cease to be owned by the connection, and so will the weapon.

In standalone games, we retrieve the **Player Controller** or the **character** by:

- Blueprint - using nodes such as:
  - Get Player Controller, or
  - Get Player Character
- C++:
  - UGameplayStatics::GetPlayerController(), or
  - UGameplayStatics::GetPlayerCharacter().

In a networked environment we may have issues if we don't know what we are doing, as we'll get different results depending on the context.

Example: calling the **Get Player Controller** function with **Player Index** equal to **0** will results in:

- The **Listen server's PlayerController** if you are calling it from a **listen server**
- The **First client's PlayerController** if you are calling it from a **dedicated server**
- The **client's PlayerController** if you are calling it from a **client**

Furthermore, the index is not consistent across the server and different clients, so more confusion can arise.

Solution: In Multiplayer games in Unreal Engine, we need to use some of the following functions (or their corresponding nodes):

- **AActor::GetOwner()**, which returns the **Owner** of an Actor instance
- **APawn::GetController()**, which returns the **Controller** for the Pawn or Character instance
- **AController::GetPawn()**, which returns the **Pawn** possessed by the Controller
- **APlayerState::GetPlayerController()**, which will return the **Player Controller** that created the Player State instance (remote clients will return a null value)

Components, on the other hand, have their own way of determining their **owning connection**: – They start by following the component's outer chain until they find the Actor that owns them. - Then, the system determines the owning connection of that Actor via **UActorComponent::GetOwner()**.

In this section, we have just "scratched the surface" of what an **Owner** is and how to get info about it, but you should be aware that **connection ownership** is so important that it will be pervasive throughout the rest of the course: in other words, the idea of owning a connection is deemed crucial enough to be addressed throughout the multiplayer project we are developing.

In the next section, we're going to deal with a strongly connected topic: **relevancy**.

2.4. Understanding Actor relevancy

Definition: **Relevancy** is the process of determining which objects in a scene should be visible or updated based on their importance to the player.

This is an important concept in Unreal Engine, and by understanding how it works, you can make sure your game runs efficiently. In this section, we will explore this topic and show an example of how it works depending on its settings.

2.4.1 Understanding relevancy

Relevancy refers to how the Engine determines **which Actors** in the game world **should be replicated** to which clients, based on their current locations, and which Actors **are relevant** to the player's current **view** or **area**.

A game level can have a size varying from **very small** to **really huge**. This may pose a problem in updating everything on the network and for every client connected to the server. As the playing character may not need to know every single thing that's happening in the level, most of the time, it's just enough to let it know what is near.

So, the Engine uses several factors to let the player know if something has changed on an Actor:

- the distance to the Actor itself,
- its visibility,
- is the Actor currently active?

An Actor that is deemed **irrelevant** will NOT be replicated to the player's client, and this will::

- reduce network traffic and
- improve game performance.

Unreal uses a virtual function named **AActor::IsNetRelevantFor()** to test the relevancy of an Actor. It goes throw some checks/tests that can be summarized as follows:

- **First** Check: The Actor is relevant if the following applies:
  - Its **bAlwaysRelevant** flag is set to true
  - Or, it is **owned** by the Pawn or PlayerController
  - Or, it is the **Pawn** object
  - Or, the Pawn object is the **instigator** of an action such as *noise* or *damage*
- **Second** Check: If the Actor's **bNetUseOwnerRelevancy** property is **true** and the Actor itself has an **Owner**, the **owner's relevancy** will be used.
- **Third** Check: If the Actor has the **bOnlyRelevantToOwner** property set to true and does not pass the first check, then it is not relevant.
- **Fourth** Check: If the Actor is **attached** to another Actor's skeleton, then its relevancy is determined by the relevancy of its **parent**.
- **Fifth** Check: If the Actor's **bHidden** property is set to true and the root component is not colliding with the checking Actor, then the Actor is **not relevant**.
- **Sixth** Check: if **AGameNetworkManager** is set to use **distance-based relevancy**, the Actor is **relevant** if it is **closer** than the **net cull distance**.

> The **Pawn/Character** and **PlayerController** classes have slightly **different relevancy checks** as they need to consider additional information, such as the **movement** component.

This system is not perfect:

- as the distance check may give a false negative when dealing with large Actors.
- the system does not take into account sound occlusion
- the system does not take into account other complexities related to ambient sounds.

Nevertheless, the approximation is precise enough to get good results during gameplay. Let's begin implementing a concrete example to see relevancy in action by testing your character.

2.4.2 Testing relevancy

To test the effect of **relevancy** during gameplay, you'll create a simple **pickup** and play around with its settings.

2.4.2.1 Creating the Pickup Actor

- Creating a new **C++ class** inheriting from **AActor**; name it **US_BasePickup**.
- Open the generated header file and add these two component declarations in the private section:

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Components", meta=(AllowPrivateAccess = "true"))
TObjectPtr<class USphereComponent> SphereCollision;
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category="Components", meta=(AllowPrivateAccess = "true"))
TObjectPtr<class UStaticMeshComponent< Mesh;
```

We just declared the:

- **SphereCollision** component for triggering the pickup, and
- **Mesh** component for its visual aspect.

- Next, in the protected section, just after the BeginPlay() declaration, add a declaration that will handle the **character overlap** with the Actor:

```
UFUNCTION()
void OnBeginOverlap(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor
  , UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult);
```

- Immediately after that, add the declaration for the **pickup action**:

```
UFUNCTION(BlueprintCallable, BlueprintNativeEvent, Category = "Pickup", meta=(DisplayName="Pickup"))
void Pickup(class AUS_Character* OwningCharacter);
```

We need this function to be callable inside a Blueprint, so we use the **BlueprintCallable** specifier. Then, the **BlueprintNativeEvent** specifier states that the function can be overwritten by a Blueprint, but it also has a default native C++ implementation that will be called if the Blueprint does not implement anything. To natively implement the method, in the **US_BasePickup.cpp** file, we will need to implement a C++ function with the same name as the primary function but with **_Implementation** added to the end. Finally, to the public section – and after the corresponding properties, in order to avoid forward declarations – add two **getters** for the components we declared previously:

```
FORCEINLINE USphereComponent* GetSphereCollision() const { return SphereCollision; }
FORCEINLINE UStaticMeshComponent* GetMesh() const { return Mesh; }
```

- Open the **US_BasePickup.cpp** file and add the necessary includes below *US_BasePickup.h*:

```
#include "US_Character.h"
#include "Components/SphereComponent.h"
```

- Then, inside the constructor, add the following block of code, which creates the two components and attaches them to the Actor:

```
SphereCollision = CreateDefaultSubobject<USphereComponent>("Collision");
RootComponent = SphereCollision;
SphereCollision->SetGenerateOverlapEvents(true);
SphereCollision->SetSphereRadius(200.0f);
Mesh = CreateDefaultSubobject<UStaticMeshComponent>("Mesh");
Mesh->SetupAttachment(SphereCollision);
Mesh->SetCollisionEnabled(ECollisionEnabled::NoCollision);
```

- Immediately after that, set **bReplicates** to true (as Actors do not replicate by default):

```
bReplicates = true; //Actors have this false by default
```

- Inside the BeginPlay() function, add a *dynamic multi-cast delegate* for the overlap event:

```
SphereCollision->OnComponentBeginOverlap.AddDynamic(this, &AUS_BasePickup::OnBeginOverlap);
```

> We will give proper attention and focus to replication, later.

- Now add the **overlap handler** just after the closing bracket of the BeginPlay() function:

```
void AUS_BasePickup::OnBeginOverlap(UPrimitiveComponent* OverlappedComponent, AActor* OtherActor,
    UPrimitiveComponent* OtherComp, int32 OtherBodyIndex, bool bFromSweep, const FHitResult& SweepResult)
{
    if (const auto Character = Cast<AUS_Character>(OtherActor))
    {
        Pickup(Character);
    }
}
```

- Notice that the previous block of code is quite straightforward: after having checked that the overlapping Actor is AUS_Character (i.e., our multiplayer hero), we simply call the **Pickup()** method (to be implemented next). As a comparison, this is similar to Unity's **OnCollisionEnter** method.
- Add the **Pickup()** C++ implementation:

```
void AUS_BasePickup::Pickup_Implementation(AUS_Character * OwningCharacter)
{
   SetOwner(OwningCharacter);
}
```

Let's create a blueprint out of this C++ class.

2.4.2.2 Creating a pickup Blueprint class

To test **IA_Interact** input action, as well as the effects of **relevancy** in action, we'll create a Blueprint pickup:

- Open the Unreal Engine Editor.
- **Compile** your project to add the pickup to the available classes of your Blueprints.
- In your Blueprints folder, create a new **Blueprint Class** inheriting from **AUS_BasePickup** and name it **BP_SpellBook**.
- Open it and in the Blueprint **Details** panel, select a **mesh** for the **Static Mesh** property – I opted for the **spellBook** model.

To make the book float, we are going to move the mesh up and down by using a Timeline node. To do so, follow these steps:

- Open the **Blueprint Event Graph**, right-click on the canvas, and add a **Timeline node** – give it a name such as **Float**.
- Double-click on the node to **open** the corresponding editor.
- Click the **+ Track** button to add a new **Float** track and name it **Alpha**.
- Click on the **Loop** button to enable the loop mode (becomes blue).
- Right-click on the curve panel and select the **Add key to...** option. Then, set **Time** to **0** and **Value** to **0**.
- Create another **key**, but this time set **Time to 2.5** and **Value to 0.5**.
- Create one **last key**, this time with **Time equal to 5** and **Value equal to 0**.
- Right-click on **each** of the keys and set the **Key Interpolation value** to **Auto**.

You have just created a **sinusoidal value** that will indefinitely **loop** between 0 and 1 values; you'll use this floating value to move the book up and down. To implement this floating movement, return to the Event Graph and do the following:

- Connect the **Event Begin Play** node to the **Timeline** node.
- Drag the **Mesh** component from the Components panel onto the **Event Graph canvas**. Click and drag from its **outgoing pin** to add a Set Relative Location node.
- Connect **Timeline** → **Update** outgoing pin to the **Set Relative Location** incoming execution pin.
- Drag the **Timeline** → **Alpha** pin to a **Multiply** node — α goes to first parameter; set the second parameter to **100** (so the **Multiply** node calculates $\alpha \cdot 100$).
- Right-click on the **New Location** pin of the **Set Relative Location** node and select **Split Struct Pin** to expose the X, Y, and Z values.
- Connect the **Result** pin of the **Multiply** node to **New Location Z** of the **Set Relative Location** node (so, make Z of the **BP_SpellBook** move up and down).

> This floating animation is purely a **visual effect**, so we just won't worry about whether it is synchronized over the network.

Now that the Blueprint item has been created, it's time to add it to the level and test its pickup functionality – something we are going to do in the next subsection.

2.4.3 Testing the relevancy settings

Let's test how the spell book behaves in a multiplayer environment when relevancy settings are changed:

- Drag an instance of the **BP_SpellBook** Blueprint into the level, near the **PlayerStart** Actor, so that the player will be in the line of sight once it has been spawned.
- Open the **PB_SpeelBook Blueprint**
  - Select **Class Defaults**
  - Search for the **Replication** category.
- Try playing the game as a **listen server** with **three players**, and every player should see the book as expected. (Remember how to do it?)
- $Take\ Snapshot$
- Stop the application from playing and get back to the **BP_SpellBook** Blueprint.
- Look for the **Net Load** on Client property and **uncheck** it.

> This property will load the Actor during **map loading**.

- Disable it, so the Actor will be loaded only when it becomes relevant for the client.

You are now ready to test different situations, depending on the properties you change in the next steps.

2.4.3.1 Setting the net cull distance

- Test **distance culling** – Notice that at the moment, your object is set to be relevant at a very far distance.
    - Lower **Net Cull Distance Squared** to a very low number, for instance, **500**.
    - Playtest. Notice that the server window will **show** the book, while the two clients will **not**!
    - With one of the client windows active, try **walking** near the zone where the book should be
    - Notice that it will immediately pop up!
- *Take Snapshot*

2.4.3.2 Setting the Actor as always relevant

- Return to the spell book Blueprint and set the **Always Relevant** property to **True**
- Play the game
- Notice that every client will be able to see the book from the start.
- *Take Snapshot*

This happens because the book has now been marked as something that should be relevant no matter where the character is in the level; as a consequence, it will be immediately loaded by the client and visible to the players. This is obviously not a desirable situation – getting a continuous update for every Actor in the level is something we don't want to happen, considering that there could be a multitude of moving elements in our game. But you probably already imagined that, didn't you?

Let's avoid this issue by setting relevancy based on the **Owner** of the Actor.

2.4.3.3 Setting the relevancy for the Owner

- Recall that the C++ code for the Pickup() function assigns the Owner of the pickup to the character overlapping it.
- Let's see what happens if the Actor is relevant only to the Owner:
    1. Set the **Only Relevant to Owner** property to **True**.
    2. Set the **Always Relevant** property to **False**.
    3. Set **Net Cull Distance Squared** to a really low number, say **10**.

With the last step, we are setting the spell book so that it won't be relevant to any client unless it is directly on the object; this will let us test who is the **Owner** of the Actor. The clients won't be able to see the book unless they enter its collision zone, which is when the character becomes the Owner of the pickup. Once another character enters the pickup zone, it will become the new Owner and the book will become relevant. After a few moments, the first client will see the book **disappear** as the character is no longer the Owner of the pickup, and so it is no longer relevant to it!

There is one last property you should be aware of: **Net Use Owner Relevancy** will return the relevancy of an Actor depending on its owner relevancy. This will come in handy once you assign a **weapon** to a character or to an enemy!

To recap, we dealt with **relevancy** and tested it in action. This will be important to optimize the game, Next, let's see another significant concept, **authority**.

2.4.4 Introducing authority

Recall that, the term **authority** refers to which instance of the game has the final say over certain aspects of the game state.

In an Unreal Engine multiplayer environment, the **server** is authoritative over the game state: this means that the server makes the final decisions about things such as:

- player movement,
- damage calculation, and
- other game mechanics.

When a client **requests** to perform an action that affects the game state, it **sends a message** to the server requesting permission to perform that action. The **server** then determines whether the action is valid and, if so, **updates** the game state accordingly. Once the server has updated the game state, it **sends** a message to all clients to inform them of the updated state.

In Unreal Engine, Actors can be either locally or remotely controlled, and the concept of authority is important in determining which controls are valid. Actors that are locally controlled have authority over their own actions, while those that are remotely controlled receive commands from the server and follow those commands. Overall, the concept of authority ensures that all players see a consistent game state and that no one player has an unfair advantage.

2.4.4.1 Controlling authority with the Role and Remote Role properties of an Actor

In Unreal Engine, there are two properties that return important information about Actor replication:

- Role, (specifies the Actor's role on the local machine)
- Remote Role (specifies the Actor's role on the remote machine)

These two properties provide information about:

- who has authority over the Actor,
- whether the Actor is replicated or not, and
- the method of replication.

In Unreal Engine, an Actor can have one of four possible **roles** during network play:

- **ROLE_Authority**: The running instance has authoritative control over the Actor
- **ROLE_AutonomousProxy**: The running instance is an autonomous proxy of the Actor
- **ROLE_SimulatedProxy**: The running instance is a locally simulated proxy of the Actor
- **ROLE_None**: In this case, the role is not relevant

Example: If **Role** is set to **ROLE_Authority** and **RemoteRole** is set to either **ROLE_SimulatedProxy** or **ROLE_AutonomousProxy**, then the current instance of the game is responsible for **replicating** this Actor to remote connections.

Only the server **replicates** Actors to connected clients as clients will **never replicate** Actors to the server. This means that only the server will have **Role** set to **ROLE_Authority** and **RemoteRole** set to **ROLE_SimulatedProxy** or **ROLE_AutonomousProxy**.

2.4.4.2 Autonomous and simulated proxy

- Test the spell book pickup again
- Notice that once the Actor's Owner changed, the book did seem to stay relevant to both the old and the new Owner for a moment.
- *Take Snapshot*

To avoid using **excessive amounts of CPU** resources and bandwidth, the server does not replicate Actors during every update but at a frequency that is determined by the **AActor::NetUpdateFrequency** property.

The same thing will happen when updating any Actor during movement, and the client will receive data at predefined intervals; as a consequence, the player may get seemingly erratic updates on an Actor.

To avoid these kinds of issues, the Engine will try to **extrapolate** movement based on the latest data available.

The default behavior relies on predicting the movement and is governed by a Simulated Proxy by setting the role to the value of **ROLE_SimulatedProxy**. In this mode, the client continuously updates the location of the Actor based on the latest velocity received from the server.

When an Actor is controlled by a **PlayerController** object, you may use an **Autonomous Proxy** by setting the role to a value of **ROLE_AutonomousProxy**. In this case, the system will receive additional information directly from the human player, making the process of predicting future actions smoother.

Here we dealt with **authority** and Actor **roles**. These notions shall be helfull in the future.

3. Part 2 - Replicating Properties Over the Network

Let's use **property replication** to:

- replicate character skills.
  - create a **coin pickup** (based on **BP_BasePickup**) to grant the character **experience points** that will give the character a **level-up** during gameplay.
- updating a simple user interface that will show:
  - the character experience points and
  - the character level.

3.1. Technical requirements

- We are going to use Unreal Engine 5.4.
- You have to have finished the Lab6 - Part 1.

3.2. Adding character stats

Let's create a set of **statistics** that will be plugged into the **Character** class. The first thing to do is to define your character stats. In particular, you will need the following data:

- A **walk** and a **sprint speed**, to handle the different paces of your character during gameplay
- A **damage multiplier** to manage more powerful hits whenever the character **levels up**
- A **level-up value** to check whenever the character has reached the next level
- A **stealth multiplier** that will handle how much **noise** the character makes when **walking** or **sprinting**

Notice that the character has no health – that is because this is a stealth game and players will have to move carefully through the dungeon. Once they are discovered, they won't have the option of facing a swarm of undead lackeys in this particular game! As a consequence, gameplay will be more focused on defeating enemies from a distance or slipping silently away from them. With the previous information, we'll create a **data structure** containing all the data points for initializing the character, and then we'll create a **data table** that will be used to manage the experience our thief will gain during gameplay.

3.2.1 Creating the stats structure

> Non-class entities cannot be made directly from within Unreal Editor.

- Open Visual Studio
- Create a file in your **US_LOTL_{YourInitials}** → **Source** → **US_LOTL_{YourInitials}** folder called **US_CharacterStats.h** (as this is a data structure, we won't need a **.cpp** file).
- Open the file and insert the following code:

```
#pragma once
#include "CoreMinimal.h"
#include "Engine/DataTable.h"
#include "US_CharacterStats.generated.h"
USTRUCT(BlueprintType)
struct UNREALSHADOWS_LOTL_API FUS_CharacterStats : public FTableRowBase
{
 GENERATED_BODY()
 UPROPERTY(BlueprintReadWrite, EditAnywhere)
```

```
 float WalkSpeed = 200.0f;
 UPROPERTY(BlueprintReadWrite, EditAnywhere)
 float SprintSpeed = 400.0f;
 UPROPERTY(BlueprintReadWrite, EditAnywhere)
 float DamageMultiplier = 1.0f;
 UPROPERTY(BlueprintReadWrite, EditAnywhere)
 int32 NextLevelXp = 10.0f;
 UPROPERTY(BlueprintReadWrite, EditAnywhere)
 float StealthMultiplier = 1.0f;
};
```

> The include section is self-explanatory. Notice the **USTRUCT()** declaration instead of **UCLASS()** and an **F** prefix on the structure name (i.e., **FUS_CharacterStats**). This is the standard method to declare a structure in Unreal Engine. Alos, notice that the structure extends **FTableRowBase**; this allows UE to create data tables from this structure. Inside the structure declaration, we are just adding a list of properties that are marked:
>
> - **BlueprintReadWrite** to let Blueprints access and modify the data, and
> - **EditAnywhere** to let you edit the values inside the data table you are going to create in the next steps.

3.2.1.1 Creating a stats data table

To create the actual data from the above structure, we will use a **DataTable** object, a tabular structure that organizes interconnected data in a coherent and practical manner. The data fields can include any valid **UObject** property, including **asset references** from the projects, such as **materials** or **textures**. To create the **character data table**, cary out the following steps:

1. Open your **Blueprints** folder in the Content Browser.
2. **Compile** your project in order to make the **C++ structure** available in the Editor.
3. Right-click in the Content Browser and select **Miscellaneous** → **Data Table**.
4. In the **Pick Row Structure** pop-up window, select **US_CharacterStats** from the drop-down menu
5. Click the **OK** button to generate the data table and name it **US_CharacterStats**.
6. Double-click on the newly created asset to **open** it. You will get an **empty** dataset.

> A data table can also be generated by importing a **.csv** or **.json** file into your project. Additionally, Unreal Engine will let you easily export your project tables in .csv and .json formats. For more information about the importing and exporting processes, check the official documentation linked here: https://docs.unrealengine.com/5.1/en-US/data-driven-gameplay-elements-in-unreal-engine/.

Let's add some **data rows** organized by character levels (we'd like to level-up the character in the future). Let's start by adding a **single row** for your character **base level**:

- Click on the **Add** button in the Table panel.
- Notice that the **Row Name** field will be named **NewRow**;
- Right-click on it and select **Rename**. Change the name of this field to **level_01**.
- You are now ready to set some stats for the first experience level of your character. Look for the Row Editor section in the data table and insert the following values:
  - **Walk Speed** = **250.0**
  - **Sprint Speed** = **800.0**
  - **Damage Multiplier** = **1.0**
  - **Next Level Xp** = **10**
  - **Stealth Multiplier** = **1.0**
- Add two more levels, **level_02** and **level_03**.
- Populate them as follows:

Level	Walk Speed	Sprint Speed	Damage Multiplier	Next Level Xp	Stealth Multiplier
level_01	250.0	800.0	1.0	10	1.0
level_02	275.0	850.0	1.1	25	1.5
level_03	300.0	900.0	1.0	50	2.0

Let's read the info from our code. Open **US_Character.h** header file to add the data table declaration.

3.2.1.2 Reading the data table from the character

In this section, you are going to add the data table to the character in order to read its values depending on the experience level. The first thing to do is to add a reference to the US_Character.h header file. So, in the private section of the header file, after all the existing declarations, add this code:

```
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "Character Data", meta = (AllowPrivateAccess = "true"))
class UDataTable* CharacterDataTable;
struct FUS_CharacterStats* CharacterStats;
```

The first declaration will let you reference the data table directly from the child Blueprint Classes, while the structure declaration will let you reference a single row from the data table and use it as the character statistics. After that, you will need to create a function that will allow the system to update the current level of the character. In the public section, add the following method declaration:

```
void UpdateCharacterStats(int32 CharacterLevel);
```

The last thing you need to add to the class header is a getter function for the stats structure. Still in the public section, just before the last closing bracket, add the following line of code:

```
FORCEINLINE FUS_CharacterStats* GetCharacterStats() const { return CharacterStats; }
```

You can now save this file and open US_Character.cpp to handle the data retrieval. At the top of the file, add the include declarations for the classes you'll be using in a moment:

```
#include "US_CharacterStats.h"
#include "Engine/DataTable.h"
```

Next, implement the UpdateCharacterStats() method by adding the following code at the end of the file:

```
void AUS_Character::UpdateCharacterStats(int32 CharacterLevel)
{
 if(CharacterDataTable)
 {
  TArray<FUS_CharacterStats*> CharacterStatsRows;
  CharacterDataTable->GetAllRows<FUS_CharacterStats>(TEXT("US_Character"), CharacterStatsRows);
  if(CharacterStatsRows.Num() > 0)
  {
   const auto NewCharacterLevel = FMath::Clamp(CharacterLevel, 1, CharacterStatsRows.Num());
   CharacterStats = CharacterStatsRows[NewCharacterLevel - 1];
   GetCharacterMovement()->MaxWalkSpeed = GetCharacterStats()->WalkSpeed;
  }
 }
}
```

As you can see, first we check that the data table is referenced (you'll add it later, from the character Blueprint) and then use the GetAllRows() method to fetch all the table rows into a local array (i.e., the CharacterStatsRows variable). If there is at least one row in the data table, we get the one corresponding to the level of the character minus 1 (i.e., for a level 1 character, we will get the row number 0). Notice, as well, the FMath::Clamp() method, which guarantees we are not trying to get a level value that's higher than the available rows in the dataset. After that, we retrieve the WalkSpeed column from the row and assign its value to the MaxWalkSpeed property of the character movement component – this means that, if there is a data table assigned, your character will start the game with a value from the dataset and not from the constructor. You are

now ready to update your character stats to level 1 – something you are about to do in the BeginPlay() function. To do so, inside the BeginPlay() function, and just before the closing bracket, add this code:

```
UpdateCharacterStats(1);
```

The last thing you need to do is to update the two sprint methods that, at the moment, are using hardcoded values but need to use the data table stats. To do so, search for the SprintStart() method and find the following line:

```
GetCharacterMovement()->MaxWalkSpeed = 3000.f;
```

Then, change it to the following code:

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

Let's do the same with the SprintEnd() method, which should be positioned just after the previous one. Find the following line: GetCharacterMovement()->MaxWalkSpeed = 500.f; Then change it using the following code block:

```
if(GetCharacterStats())
{
 GetCharacterMovement()->MaxWalkSpeed = GetCharacterStats()->WalkSpeed;
}
```

In both cases, the code is self-explanatory – we just check that there is valid data referenced in the character stats and assign the sprint or walk speed to the character movement component. Now save your file and compile the project, just to be sure that everything is fine and ready for the next step. Your character is now ready to accept the data table we created at the beginning of this chapter.

3.2.1.3 Adding the data table to the character

To add the data table asset to the character, switch back to Unreal Editor and follow these steps:

1. Open the BP_Character Blueprint.
2. Select the Class Defaults tab and, in the Details panel, look for the Character Data category.
3. In the Character Data Table attribute, click on the drop-down menu and select DT_CharacterStats.

Your character is now ready to use the statistics from the dataset – even though the poor thief is locked into a level 1 experience level, later on, you will set them free in the dungeon and see how they fare! Test the game to check that everything works properly. Just remember what I said in the previous chapter: movement is still buggy as the client and the server are trying to force the character to conform to different speed values, but you are nearing the solution. So, in this section, you have improved the character by adding some statistics retrieved from a data table and using them to initialize some properties. At the moment, you have just used the movement ones, but don't be afraid! Once the character is completed, everything will fall into place. In the upcoming section, we'll dive into the topic of property replication in Unreal – something that will come in handy when it's time to level up your character and something you'll be doing by the end of this chapter.

3.3. Understanding property replication

As stated before, property replication allows for the synchronization of objects in an Unreal multiplayer environment. It should be noted that, as the server is authoritative, updates will never be sent by the client. Obviously, the client may (politely) ask the server to change a property value, and the server will behave accordingly. Additionally, property replication acts as a reliable service: consequently, the Actor on the client will have the same value as the server sooner or later. This means that if you're trying to modify, from the client, a property that is replicated, any changes you make to that property will only be temporary. You should be already familiar with this topic as the character's movement logic, at the moment, is a bit buggy – we are trying to make the character run from the client, but the server is blocking our commands as soon as the network is updated. This is happening because, as soon as the server sends an update to the client with a new value for that property, any changes that you made locally on the client will be overwritten and replaced with the new, correct value from the server. Consequently, if the server does not update frequently, it may take a while for the client to be notified about the new, correct value. Fixing that nasty bug is something we will be doing in Chapter 7, Using Remote Procedure Calls (RPCs), where you'll

need to learn how to call a function from the client to the server. The main focus of this chapter, however, is to understand how to replicate properties. So, without further ado, let's check how things work under the hood!

3.3.1 Enabling property replication

In order for a property to be replicated, you need to set up a few things. First of all, in the Actor constructor that will contain the property, you will need to set the bReplicates flag to true.

> A class or Blueprint extending from APawn or ACharacter will have the bReplicates property set to true by default, while a regular Actor won't.

Then, the property that should be replicated will need to have the Replicated specifier added to the UPROPERTY() macro. As an example, you can replicate the score for your character with the following code:

```
UPROPERTY(Replicated)
int32 Score;
```

If you are in need of a callback function to be executed when a property is updated, you can use ReplicatedUsing= [FunctionName] instead – this attribute will let you specify a function that will be executed when an update is sent to the client. For example, if you want to execute a method called OnRep_Score() whenever your character score is replicated, you will write something similar to the following code:

```
UPROPERTY(ReplicatedUsing="OnRep_Score")
int32 Score;
```

You will then need to implement the OnRep_Score() method in the same class; this function must declare the UFUNCTION() macro. Once all the replication properties are properly decorated by the previous attributes, they need to be declared inside the AActor::GetLifetimeReplicatedProps() function by using the DOREPLIFETIME() macro. Using the previous score example, you will need to declare the Score property by using the following code:

```
DOREPLIFETIME(AMyActor, Score);
```

After a property is registered for replication, it cannot be unregistered, as Unreal Engine will optimize data storage to reduce the computation time: this means that, by default, you will not have much control over how a property replicates. Luckily, you can use the DOREPLIFETIME_CONDITION() macro instead, which will let you add an additional condition for more precise control over replication. Values for these conditions are predefined – one example is COND_OwnerOnly, which will only send data to the Actor's owner (we will use this value later in the chapter). As another example, if you need even more fine-grained control in property replication, you can use the DOREPLIFETIME_ACTIVE_OVERRIDE() macro, which will let you use your own conditions defined inside the Actor. The major downside of using additional conditions for replication is performance, as the engine will need to do additional checks before replicating a property – this means that it is advisable to use the DOREPLIFETIME() macro in situations where no pressing requirements dictate the use of an alternative option. Now that you understand how an object can be replicated, it's time for me to introduce how objects are referenced across the network.

3.3.2 Referencing Actors and components over the network

Sooner or later, you will need to reference an Actor or a component from your code – this means that, in a multiplayer game, you will need to know whether the reference can be replicated or not. Simply put, an Actor or a component can be referenced over the network only if it is supported for networking. There are some simple rules that will help you determine whether your object can be referenced over the network:

- If an Actor is replicated, it can also be replicated as a reference
- If a component is replicated, it can also be replicated as a reference
- Non-replicated Actors and components need to be stably named in order to be replicated as references

> An object that is stably named means that an entity that will be present in both the server and the client that has the same name. For instance, an Actor is stably named if it was not spawned during gameplay but was loaded directly in the level from a package.

This section has provided you with an introduction to the fundamental concepts of network replication in Unreal Engine, explaining how it interacts with Actors and components. If you feel a bit lost about too much theory, don't be afraid! You'll be taking all that theory and transforming it into a tangible, working example by creating a level-up system for your character.

3.4. Handling character level-ups

As I previously mentioned, in this section, you are going to level up your hero's experience and skills. As usual, you'll be dabbling in code magic to make it happen! After all, you are programming a fantasy game. I know it might seem like a good idea to write your code inside the Character class but trust me when I say that there's actually a much better spot for it. That is the PlayerState class, which we incidentally have already set for this occasion – a while ago, I asked you to create the US_PlayerState class and now is the time to add some valuable code in it. As introduced in Chapter 4, Setting Up Your First Multiplayer Environment, PlayerState is a class that holds information about a player's game state and exists on both the server and clients. As we need to synchronize experience points and levels for the character, this is the ideal location to place everything. What we need to do here is to keep track of experience points and, as soon as the character reaches a new level, broadcast the information across the network and update the character statistics. But first, the most important thing is to have a clear idea of what we are going to do.

3.4.1 Planning ahead

As the PlayerState class will keep important information about the character, it's mandatory to think ahead about what you want to achieve and how to get to that point – this means we have to plan exactly what we will be adding to this class. Here are some of the main features this gameplay framework class will implement:

- Keeping track of the character's current level and experience points
- Synchronizing the aforementioned properties over the network
- Updating the Character class whenever the player levels up
- Broadcasting events whenever the character gets some experience points or levels up

As a starting point, in the next subsection, we'll start by declaring the required properties and functions.

3.4.2 Declaring PlayerState properties and functions

In the following steps, we are going to define the main properties that will let the character level up whenever they have enough experience – this means we will need to track the thief's experience points and level. Additionally, whenever values change, we will replicate these properties over the network and notify this event to each registered Actor in the game. So, let's start by opening the US_PlayerState.h file and adding the following code in the protected section:

```
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, ReplicatedUsing="OnRep_Xp", Category = "Experience")
int Xp = 0;
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, ReplicatedUsing="OnRep_CharacterLevelUp", Category = "Experience")
int CharacterLevel = 1;
UFUNCTION()
void OnRep_Xp(int32 OldValue) const;
UFUNCTION()
void OnRep_CharacterLevelUp(int32 OldValue) const;
```

As you can see, the first thing we have done is declare the two properties Xp (short for experience points) and CharacterLevel; both can be modified in Unreal's Details panel for playtesting purposes thanks to the EditDefaultsOnly attribute, but BlueprintsReadOnly makes them non-modifiable in a Blueprint, to keep all the level-up logic inside the C++ source code. As an additional attribute, we use the ReplicatedUsing attribute, which I introduced in the previous section. This will let us execute a function whenever a property is updated – in this case, we have set OnRep_Xp for the Xp property and OnRep_CharacterLevelUp for CharacterLevel. Next, create a public section in your header file and add this code:

```
UFUNCTION(BlueprintCallable, Category="Experience")
void AddXp(int32 Value);
```

This function will let us assign new experience points to the PlayerState. We need to make it BlueprintCallable in order to use this function from our Blueprints – for instance, from a pickup. Just after that, add this declaration:

```
virtual void GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const override;
```

As explained in the previous section, we need to override this method in order to declare the properties that will be replicated (more on this in a moment). All the necessary setup for implementing replication in our two properties has been completed, but a few additional elements still need to be incorporated to ensure everything works properly. We need to broadcast some information whenever these properties change – this will come in handy when you implement a user interface later in this chapter. To implement such functionality, you'll be using delegates. You may be already familiar with this topic in C++, but you should be aware that, in Unreal Engine, a delegate provides a way to call member functions on C++ objects in a generic, type-safe manner through dedicated macros.

> If you want more information about the types of delegates supported by Unreal Engine and how they can be used in your project, check out the official documentation, which can be found here: https://docs.unrealengine.com/5.1/en-US/delegates-and-lamba-functions-in-unreal-engine/.

As we want broadcast events for the two properties, we will be declaring two delegates – one for each property. At the beginning of the header file, just before the UCLASS() declaration, add the following code:

```
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FOnXpChanged, int32, NewXp);
DECLARE_DYNAMIC_MULTICAST_DELEGATE_OneParam(FOnCharacterLevelUp, int32, NewLevelXp);
```

These two lines are pretty similar – they both declare a dynamic multi-cast delegate with a single parameter. Being dynamic, the delegate can be serialized, and so, used in a Blueprint, while the multi-cast declaration will allow you to attach multiple function delegates and use the Broadcast() method to notify every listener of changes in your system. We will use these features in our Blueprint classes to bind events and react accordingly. Let's declare our delegate function. Create a protected section, and add the following two lines of code, which will be used to broadcast the events:

```
UPROPERTY(BlueprintAssignable, Category = "Events")
FOnXpChanged OnXpChanged;
UPROPERTY(BlueprintAssignable, Category = "Events")
FOnCharacterLevelUp OnCharacterLevelUp;
```

As their purpose is self-explanatory, I guess it's time stop talking and start writing down the implementation!

3.4.2.1 Implementing the PlayerState logic

Now that all the properties and methods have been declared, you are going to implement the PlayerState logic – whenever the character gains some experience, you should check whether it has reached enough points to level up. Experience points gained and level-ups should be broadcast to the system, in order to keep everything synchronized. Start by opening the US_PlayerState.cpp file and adding the required include declarations:

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

Next, add the implementation for the GetLifetimeReplicatedProps() method:

```
void AUS_PlayerState::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const
{
 Super::GetLifetimeReplicatedProps(OutLifetimeProps);
 DOREPLIFETIME_CONDITION(AUS_PlayerState, Xp, COND_OwnerOnly);
 DOREPLIFETIME_CONDITION(AUS_PlayerState, CharacterLevel, COND_OwnerOnly);
}
```

As you can see, we are using the DOREPLIFETIME_CONDITION() macro, introduced in the previous section, to declare that the Xp and CharacterLevel properties should be replicated – in this case, we just want the property to be replicated on the owning player of the character (i.e., in the player's client), and we do so by using the COND_OwnerOnly flag. Next, add the implementation for the AddXp() method using the following code:

```
void AUS_PlayerState::AddXp(const int32 Value)
{
 Xp += Value;
 OnXpChanged.Broadcast(Xp);
 GEngine->AddOnScreenDebugMessage(0, 5.f, FColor::Yellow, FString::Printf(TEXT("Total Xp: %d"), Value));
 if (const auto Character = Cast<AUS_Character>(GetPawn()))
 {
  if(Character->GetCharacterStats()->NextLevelXp < Xp)
  {
```

```
    GEngine->AddOnScreenDebugMessage(3, 5.f, FColor::Red, TEXT("Level Up!"));
    CharacterLevel++;
    Character->UpdateCharacterStats(CharacterLevel);
    OnCharacterLevelUp.Broadcast(CharacterLevel);
   }
  }
}
```

Here, whenever we receive an experience point update, we simply add the value to the character pool (i.e., the Xp property). Next, we ascertain that the character is an AUS_Character type through a cast and, if the cast is successful, we retrieve its statistics to check whether it should level up. If the check is successful, we simply increase the character level and call the UpdateCharacterStats() method to make the thief update the skill row. As we change the values for the properties, we then broadcast the new value to all listeners. A couple of (temporary) debug messages complete the code. The PlayerState is now almost finished – we just need to broadcast the values to the clients whenever values are updated from the server side. To do so, add this last block of code to the file:

```
void AUS_PlayerState::OnRep_Xp(int32 OldValue) const
{
 OnXpChanged.Broadcast(Xp);
}
void AUS_PlayerState::OnRep_CharacterLevelUp(int32 OldValue) const
{
 OnCharacterLevelUp.Broadcast(CharacterLevel);
}
```

The broadcast call is self-explanatory – every registered Actor will receive the notification, along with the new value for the Xp and CharacterLevel properties. So, in the preceding steps, you have successfully developed a fully operational replication system that effectively manages the character experience gained and skill development. I'm aware that the task at hand may feel daunting and even sometimes counter-intuitive, but with time and practice, you can rest assured that everything will become easier and more manageable! There's still something missing in our game: actual experience points. Let's not waste any time and work on adding an item that our character can use to gain experience points. In the next steps, you'll be creating some coins, starting with the previously created US_BasePickup class, to grant your thief the much-desired experience.

3.4.2.2 Adding coin pickups to the level

So, we are ready to create some coins that will be used in the game to add experience points to the character – this will be a simple Blueprint that will be spawned whenever enemies are killed or that will be available in the level. To do this, go back to Unreal Editor and compile the project, in order to update it with all your improvements. Then, navigate to the Blueprints folder and complete the following steps:

1. Right-click in the Content Browser and select Blueprint Class | US_BasePickup to create a new Blueprint from it.
2. Name the Blueprint BP_GoldCoinPickup and double-click on it to open it.
3. In the Components panel, select the Mesh component and assign to the Static Mesh attribute the coin static mesh. Change its scale to 2, in order to make it more visible in the game.

Now that the pickup has a base shape, it's time to add some code logic to make things fully functional. Open the Event Graph tab and take the following steps:

4. Create a variable of type Integer and call it EarnedXp, giving it a default value of 5.
5. Right-click in the canvas and look for Event Pickup, adding it to the graph.
6. Add a Cast To US_PlayerState node and connect its incoming execution pin to the outgoing execution pin of the event.
7. Click and drag from the Owning Character pin of the Event node and, after releasing the button, add a Get PlayerState node from the options that appear.
8. Connect the PlayerState outgoing pin to the Object pin of the Cast node.
9. Click and drag from the outgoing As Us PlayerState pin to create an Add Xp node.
10. Connect the success execution pin of the cast node to the incoming execution pin of the Add Xp node.
11. Drag a Get Earned Xp node from the Variables section into the canvas and connect its pin to the Value pin of the Add Xp node.
12. Finally, add a Destroy Actor node and connect it to the outgoing execution pin of the Add Xp node.

As you can see, the visual scripting code is quite straightforward – whenever a character picks up a coin, its PlayerState will be updated with the experience points granted by it. To test the game, just drag a bunch of coins inside your level and play the game. Every time a character picks up a coin, you should see a display message, and when the character has enough experience, you should get another message, the level-up one. It should be noted that, in the previous code, the pickup event will be called both on the client and on the server – this is something that should not be done as it may provoke issues in your game. Luckily, in this case, the PlayerState will correctly handle the data, so we don't have to worry about it. As an extra exercise, you can add a floating animation to the coin, just like you did for the spell book earlier.

3.4.2.2.1 Adding coin subclasses

As an optional step, you can create different coin pickups with different values for experience points. Here's how to do so:

1. Right-click on PB_GoldCoinPickup and select Create Child Blueprint Class, naming the asset BP_SilverCoinPickup.
2. Assign a value of 3 to Earned Xp and MI_Metal as the mesh material.

To provide your character with various items to search for, repeat this step as many times as you desire. This will grant your character a diverse set of treasures to seek out. In this section, you have created a level-up system for your thief hero. With the help of replication, a character will get the correct level-up notification upon reaching enough experience points. At the moment, this can be achieved by collecting coin pickups around the level – later on, you'll spawn treasure upon defeating those nasty Lichlord minions! In the next section, I will guide you through the creation of a simple UI that will show the character level and the experience points that have been gained; you'll perform this task by listening to PlayerState notifications and reacting accordingly.

3.5. Adding a HUD to the game

In this section, you will create a Heads Up Display (HUD) for the game that will assist in monitoring the player character's progress during the game. As you may already know, the best way to create such information is through the Unreal Motion Graphics (UMG) system – a GUI-based editor that allows developers to create user interface elements for their game, such as menus, HUDs, and other display screens. You'll be using this system to create the HUD widget with the relative info. What we need to show at the moment is quite simple – one set of text showing the character's experience points and another set showing the level. Let's start by creating the Blueprint and the visual elements.

3.5.1 Creating the Widget Blueprint

To create the Widget Blueprint, within Unreal Editor, take the following steps:

1. Open your Blueprints folder, right-click on the Content Browser, and select User Interface | Widget Blueprint to create a User Widget Blueprint. Name the newly created asset WB_HUD and double-click on the asset to open it.
2. Drag a Canvas element from the Palette tab into the Designer view. This canvas will act as the main container for your visual elements.
3. Drag a Text element into the previously added Canvas and call it XpLabel. Make sure that the Is Variable field in the Details panel is checked to expose this element in the graph you'll be using later.
4. Position the label somewhere on the canvas that suits your needs; in my case, I opted for the top-left corner of the screen.
5. Drag another Text element into the Canvas instance and call it CharacterLevelLabel. Again, make sure that the Is Variable field in the Details panel is checked to expose this element in the graph you'll be using later.
6. Position the label somewhere on the canvas that suits your needs; in my case, I opted for the top-right corner of the screen.

Now that you have created the widget, it's time to add some Visual Scripting code to make it fully functional.

3.5.2 Adding code logic to the Widget Blueprint

In the following steps, you'll add some code logic to the Blueprint, in order to listen to events from the PlayerState and react accordingly.

3.5.2.1 Creating a custom event for the experience points label

Let's start by creating a custom event that will update the experience points label. To do this, open the Graph panel of your widget and take the following steps:

1. Create a custom event and call it OnXpChanged_Event.
2. Select it and, in the Details panel, add an input of type Integer. Name it NewXp.
3. From the MyBlueprint panel, drag a getter node for XpLabel.
4. From the XpLabel outgoing pin, click and drag, adding a SetText (Text) node.
5. Connect the OnXpChanged_Event execution pin to the incoming SetText (Text) execution pin.
6. Connect the New Xp pin of the Event node to the In Text pin of the SetText (Text) node. This operation will automatically add a To Text (Integer) node converter.

As an additional, optional step, you may want to add an Append node – this is usually used to prefix some descriptive text to be shown in the text label, such as Experience Points (for example, Experience Points: 150). Now that you have a custom event to handle the experience points label, it's time to do the same for the character level.

3.5.2.2 Creating a custom event for the character level label

Let's now create a custom event that will update the character level label:

1. Create a custom event and call it OnCharacterLevelUp_Event.
2. Select it and, in the Details panel, add an input of type Integer. Name it NewLevel.
3. From the MyBlueprint panel, drag a getter node for CharacterLevelLabel.
4. From the CharacterLevelLabel outgoing pin, click and drag and, after releasing the mouse button, select a SetText (Text) node from the options that appear.
5. Connect the OnLevelLabelChanged_Event execution pin to the incoming SetText (Text) execution pin.
6. Connect the New Level pin of the Event node to the In Text pin of the SetText (Text) node. This operation will automatically add a To Text (Integer) node converter.

Just like the previous label, you may want to use the Append node to prefix the character level label with descriptive text such as Level: (for example, Level: 1). Now that you have a custom event to handle the character level label, it's time to bind these events to the notifications broadcast by the PlayerState.

3.5.2.3 Binding to PlayerState events

In this final step of the Widget Blueprint, you will bind the previously created events to the PlayerState, in order to update the HUD every time an update notification is dispatched:

1. Add an Event on Initialized node to the graph. This node is executed only once during the game (i.e., when the object has been initialized) and is the best place to add bindings.
2. Connect the event to a Delay node with Duration equal to 0,2. As the PlayerState won't be available at initialization time, waiting until it is available is a quick solution to solve the issue.
3. Add a Branch node and connect its incoming execution pin to the Completed execution pin of the Delay node. Connect the False execution pin of the Branch node to the incoming execution pin of the Delay node; this will create a loop that will go on until the PlayerState has been properly initialized.

Now we are going to recover the PlayerState from the player owning this widget:

4. Add a Get Owning Player node to the graph. This node returns the player that is controlling (i.e., owns) the HUD.
5. From the Return Value pin of this node, click and drag to create a Get PlayerState node.
6. From the PlayerState outgoing pin of the newly created node, click and drag to create a Cast To US_PlayerState node. Right-click on this node and, from the options, select Convert to Pure Cast. As the game is based on the US_PlayerState class, we are pretty sure that we are going to recover that type of PlayerState, so we don't need to worry about validation.
7. Connect the Success pin of the Cast To US_PlayerState node to the Condition pin of the Branch node.
8. From the outgoing As US PlayerState pin, click and drag to create a new variable by selecting Promote to variable. You will automatically get a Set PlayerState node in the graph – name the variable PlayerState.

9. Connect the True execution pin of the Branch node to the incoming Set PlayerState execution pin.
10. *Take Snapshot*

Now that you have a reference to the PlayerState, it's time to bind the custom events to the delegate you created in the previous sections:

10. From the outgoing pin of the Set PlayerState node, click and drag to create a Bind Event to On Xp Changed event; this event is available thanks to the delegate declaration included in the US_PlayerState class.
11. Connect the outgoing execution pin of Set PlayerState to the incoming execution pin of the Bind Event to On Xp Changed node.
12. From the Event pin of the bind node, click and drag to add a Create Event node. This node has a drop-down menu – here, select OnXpChanged_Event (NewXp), which will execute the OnXpChanged_Event custom event whenever the system receives the corresponding notification from the PlayerState.
13. Connect the outgoing execution pin of the Bind Event to On Xp Changed node to an On Xp Changed Event node; this will call the event upon initialization, to update the HUD.
14. From the Variables section, drag a Get PlayerState node and from it, create a Get Xp node. Connect the outgoing pin of the Get Xp node to the New Xp pin of the On Xp Changed Event node.
15. *Take Snapshot*

The last part of the binding phase is almost identical to the steps you have just taken, with the exception that we are creating a binding for the player level:

15. From the outgoing pin of the On Xp Changed Event node, click and drag to create a Bind Event to On Character Level Up node.
16. Drag a Get PlayerState node from the Variables section and connect it to the Target pin of the Bind Event to On Character Level Up node.
17. From the Event pin of the Bind Event to On Character Level Up node, click and drag to add a Create Event node. From the drop-down menu, select OnCharacterLevelUp_Event (NewLevel). This selection will execute the OnCharacterLevelUp_Event custom event whenever the system receives the corresponding notification from the PlayerState.
18. Connect the outgoing execution pin of the Bind node to an On Character Level Up Event node; this will call the event upon initialization, to update the HUD.
19. From the Variables section, drag a Get PlayerState node to create a Get Character Level node. Connect the outgoing pin of the Get Character Level node to the New Level pin of the On Character Level Up Event node.
20. *Take Snapshot*

You have finally created all the bindings to listen to any PlayerState notifications and update the HUD accordingly. It's now time to add the final step – showing the HUD in-game.

3.5.3 Adding the HUD to the character

Now you will add the HUD to the player viewport. If you are already familiar with Unreal Engine user interfaces in standalone games, you may already know how things work. However, you should be aware that, in a multiplayer environment, a user interface widget should be attached to the game viewport only if the character is controlled locally (i.e., is the owning client). If you don't check whether the character creating the widget is controlled locally, you will create a widget for each character spawned in the level – including those controlled by other players and replicated in the client. Having a cluttered mess of superimposed HUDs is obviously something you don't want to have in your game! To add the HUD to the character, follow these steps:

1. Start by finding the BP_Character Blueprint and opening it.
2. In the Event Graph, find the Begin Play event. Then, add a Branch node to the execution pin of the event.
3. Connect the Condition pin of the Branch node to an Is Locally Controlled node – this will guarantee we are attaching the HUD only to the character controlled by the client.
4. From the True execution pin of the Branch node, create a Create Widget node. From the Class drop-down menu, choose WB_HUD to select our HUD.
5. Connect the outgoing execution pin of the Create Widget node to an Add to Viewport node. Connect the Return Value pin to the Target pin.
6. *Take Snapshot*

The previous Visual Scripting code is pretty easy to understand, but it is important to mention that the viewport is only added to the character controlled by the client, as having multiple HUDs overlaying each other would not be desirable! Now that everything has been properly set, you are going to test your game to see how it works!

3.5.4 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!

4. Summary

In this Lab: In Part 1, we:

- Managed Actors in a Multiplayer Environment
  - Set up the character
  - Added interactions (input actions, input mapping context)
  - Controlled the connection to an actor

In Part 2, we: Continued with:

- Managed the Actor Relevancy and Authority

Also:

- Added character stats in a stats data table
- Created coin pickups to allow player to get Xp-s and progress.
- Created a HUD to show the progress to the player.

5. Zip Project and Submit

- Click **File → Zip Project**
- Keep the default folder, select the name **US_LTOL_{YourInitials}_Lab5.zip**
- When the zipping finishes click on the provided link **Show in Explorer**
- Upload the zip in the corresponding folder (Lab5).