CS378 Lab 7: Networking

This lab will explore the basics of working with network functionality in Unreal and understanding the principles of using replicated variables and RPCs. You will create a thrilling multiplayer game built around random counting. While not exactly “game of the year” material, this will give you an idea of how to get started on networking!

Getting Started (Complete Before Class)

Create a new project named Lab_7 from a blank template building on C++. You can choose whether or not to include starter content. Go ahead and include the Mannequin assets in your Content folder, so you can see the other player characters more clearly. Set up a map with at least three Player Starts positioned to face each other. You may want to put some assets in the scene to make it easier to discern where the specific players are spawning.

Create a C++ subclass that inherits from ACharacter called Lab7Character. We only need to add WASD controls to this character, so you can attach the Axis bindings in Lab7Character, or you can create a APlayerController called Lab7PlayerController if you want to be more “correct.” Either way, create a BP of your Lab7Character called Lab7CharacterBP, and add a Mannequin to the skeletal mesh. Set this to be the default pawn to spawn into the scene via your Lab7GameModeBase.

Make sure everything compiles and that’s it! We’ll be adding any additional functionality during the lab.

Testing Networking in the Editor

For the sake of simplicity, we are only going to test the networking locally in Editor. In practice, this is no substitute for testing across a network, but it will be sufficient for the minimum networking specification of the current assignment. For testing purposes, we’ll switch Play to “Standalone Game.” This will ideally have slightly fewer quirks than playing in “Selected Viewport.” You can set the number of players you want to simulate under Multiplayer Options → Number of Players. You will also want to set the Net Mode to “Play as Listen Server” since we’re assuming the game’s host will also be a player. You can always play with these settings to see what happens, so feel free to experiment!
Creating a Replicated Variable

We will now add a replicated variable to the Lab7Character class. For the sake of demonstrating functionality, we will need to add both a Count integer and a UTextRenderComponent to display our current “score.” Count will be replicated using RepNotify, which means when the variable’s value changes, this change will be replicated to all other clients. We will give Count a UPROPERTY macro that looks like this:

\[
\text{UPROPERTY(ReplicatedUsing = OnRep_Count)}
\]
\[
\text{int32 Count;}
\]

To ensure this variable is replicated, we need to add it to our override of GetLifetimeReplicatedProps. This function will look something like this:

```cpp
void ALab7Character::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const
{
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);
    DOREPLIFETIME(ALab7Character, Count);
}
```

We now need to make this OnRep_Count function, and we’ll want to create some function that will be executed when OnRep_Count is called on the client. We’ll call this function DisplayCount() since that’s what we’ll do when Count is incremented. DisplayCount will update a Text Render Component to display a local (unreplicated) GUI displaying the other players’ scores. We will create a UTextRenderComponent called CountText and then attach this Component to our Lab7Character in the Lab7Character’s constructor. Once you’ve successfully attached this component, go to the Lab7CharacterBP and make sure the text is displaying in a nice, visible location.
Calling RepNotifies

We now need a way to increment `Count` so we can test out its replication. For the sake of simplicity, we’ll do this by simply having each Playable Character call a function `IncrementScore` on a timer set to a random timer length. Don’t make the timer length too short, or it will be hard to monitor all your network activity, but also don’t make it too long, or you’ll have to wait.

This function, `IncrementScore` will increment `Count`, call `DisplayCount` locally to update this Character’s score, and then again call `IncrementScore0` on a timer. When `Count` is incremented here, it will call `OnRep_Count` on the other clients, so we should have `OnRep_Count` call the function `DisplayCount` as well, so that all clients can see the other players’ scores. At this point, when you test your code, you should see each Playable Character with their score displayed above their head (or wherever you put your Text Render Component).

Note that you will need to add:

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

In order to access these networking features.

Calling RPCs

The next step is to try making some “win condition” that is verified on the server and updates the state for the clients. To do this, we’re going to create a couple functions. `ServerDeclareWinner()` will have the macro: `UFUNCTION(Server, Reliable, WithValidation)`, and it will require `Implementation` and `Validate` thunks. This function will be called when a Playable Character’s `Count` reaches 10 or higher. For the sake of simplicity, winning the game makes you leave it...i.e. we’ll `Destroy` the `Lab7Character` that won the game then let the others keep counting.

This is not very interesting behavior, but to do more exciting things to our `Lab7Characters`, we’ll probably need to access the `PlayerArray` in the `GameState` class. You are welcome to try out something fancier (and certainly this will only help with the networking assignment), but we’ll be coming back to `GameState` soon if you are unsure how to use it.

Since we’re Destroying the `Lab7Character` on the server, this destruction will be replicated to the clients, so we’re basically done with our “game.” If you’d like to play around with some of the other features of networking, you can try adding some of these calls:

- A `HasAuthority()` boolean to select functionality based on whether the code is currently executing on the server or a client
- A `NetMulticast` to execute code both on the server and all the clients

While our demo game is perhaps to trivial to need these, you can add On Screen Debug Messages to help you get an idea of what’s happening where.
Submission

After you’re satisfied, collect some video footage showing the system in action. Also screenshot exciting parts of your code, and submit these files plus the project code via your GitLab account and include a link to your video via Youtube. Link your repository as your Canvas submission.