

'Unreal Engine 4' Network Compendium

Created by Cedric 'eXi' Neukirchen

Blog: cedric-neukirchen.net

Co-Founder of Salty Panda Studios

You can hire us: saltypandastudios.com

Document-Version 1.5.4

Table of Contents

ATTENTION	4	Dedicated Server.....	50
Introduction	5	Listen-Server.....	51
Network in Unreal	6	Replication	52
A small example:.....	6	What is 'Replication'?.....	52
Another example:.....	6	How to use 'Replication':.....	53
IMPORTANT!	7	Replicating Properties.....	54
Framework & Network	8	Remote Procedure Calls	59
Common Classes.....	11	Requirements and Caveats.....	60
Game Mode.....	12	RPC invoked from the Server.....	61
Examples and Usage.....	13	RPC invoked from a Client.....	61
Blueprint.....	13	RPCs in Blueprints.....	62
UE4++.....	17	RPCs in C++.....	63
Game State.....	20	Validation (C++).....	65
Examples and Usage.....	21	Ownership	67
Blueprint.....	21	Actors and their Owning Connections.....	69
UE4++.....	24	Actor Relevancy and Priority	71
Player State.....	26	Relevancy.....	71
Examples and Usage.....	27	Prioritization.....	73
Blueprint.....	27	Actor Role and RemoteRole	75
UE4++.....	29	Role/RemoteRole Reversal.....	76
Pawn.....	31	Mode of Replication.....	77
Examples and Usage.....	32	ROLE_SimulatedProxy.....	78
Blueprint.....	32	ROLE_AutonomousProxy.....	79
UE4++.....	36	Traveling in Multiplayer	80
Player Controller.....	40	Non-/Seamless travel.....	80
Examples and Usage.....	42	Main Traveling Functions.....	81
Blueprint.....	43	UEngine::Browse.....	81
UE4++.....	45	UWorld::ServerTravel.....	82
HUD.....	48	APlayerController::ClientTravel.....	82
Widgets (UMG).....	49		
Dedicated vs Listen Server	50		

Enabling Seamless Travel.....	83	Destroying Sessions.....	96
Persisting Actors / Seamless Travel.....	84	Destroy Session via Blueprint.....	96
Online Subsystem Overview.....	85	Destroy Session via C++.....	96
Online Subsystem Module.....	86	Searching Sessions.....	97
Basic Design.....	86	Searching Session via Blueprint.....	97
Use of Delegates.....	87	Searching Session via C++.....	97
Interfaces.....	88	Joining Sessions.....	98
Profile.....	88	Joining Session via Blueprint.....	98
Friends.....	88	Join Session via C++.....	98
Sessions.....	88	Cloud-Based Matchmaking.....	99
Shared Cloud.....	89	Following and Inviting Friends.....	100
User Cloud.....	89	How to Start a Multiplayer Game.....	101
Leaderboards.....	89	Advanced Settings.....	102
Voice.....	89	Use Single Process.....	104
Achievements.....	89	Run as Dedicated Server.....	105
External UI.....	90	Start and Connect to a Server.....	106
Sessions and Matchmaking.....	90	Blueprint.....	106
Basic Life-Time of a Session.....	91	Start a Server.....	106
Session Interface.....	92	Connect to a Server.....	106
Session Settings.....	93	UE4++.....	107
Session Management.....	94	Start a Server.....	107
Creating Sessions.....	94	Connect to a Server.....	107
Create a Session via Blueprint	94	Starting via Command Line.....	108
Create a Session via C++.....	94	Connection Process.....	109
Updating Sessions.....	95	The major steps are.....	109
		Sources:.....	111

ATTENTION

This Compendium should **only** be used with a base understanding of the Singleplayer Game Framework of Unreal Engine 4.

It will **NOT** teach beginners how to use Unreal Engine 4 in general.

Please refer to the “**Blueprint Compendium**”, created by **Marcos Romero**, for getting started with Unreal Engine 4 as a beginner.

This Network Compendium expects you to know how to use Blueprints and also C++ (for the C++ examples).

Examples were created in Unreal Engine Version **4.14.x**, so they might differ a bit from the newest UE4 version you are using.

It's solely meant to get you started with Networking in UE4!

Introduction

Welcome to this, hopefully, useful **Unreal Engine 4 Network Compendium**.

The official Documentation is already quite good, but I wanted to sum up all the stuff, which I learned in the past two years while working with UE4 and Multiplayer Games, in one big **Compendium with Blueprint and Code** examples.

The following pages will introduce you into the **Network Framework**, explain **Classes**, **Replication**, **Ownership**, and more. When ever I can, I will provide a Blueprint and a C++ example, which should shed some light on how the things work.

The pages will, **of course**, also contain lots of content taken from the official Documentation (Source 1*), since this is how the system works after all.

This Document is meant as a summary and not as a written redefinition of something that already has been explained!

Network in Unreal

Unreal Engine 4 uses a standard **Server-Client** architecture.

This means, that the Server is **authoritative** and all data must be send from Client to Server first.

Then the Server validates the data and reacts depending on your code.

A small example:

When you move your Character, as a Client, in a Multiplayer Match, you **don't** actually move your Character by yourself, but tell the Server that you want to move it. The Server then updates the location of the Character for everyone else, including you.

Note: To prevent a feeling of “**lag**” for the **local** Client, Coders often, in addition, let this Player directly control their Character locally, although the Server still might override the Character's Location when the Client starts cheating! This means, that the Client will (almost) never '**talk**' to other Clients directly.

Another example:

When sending a Chat-Message to another Client, you are actually sending it to the Server, which then passes it to the Client you want to reach. This could also be a **Team, Guild, Group**, etc.

IMPORTANT!

Never trust the Client! Trusting the Client means, you don't test the Clients actions before executing them. This allows **cheating**!

A simple example would be Shooting:

Make sure to test, on the Server, if the Client actually has Ammo and is allowed to shoot again instead of directly processing the shot!

Framework & Network

With the previous information about UE4's Server-Client architecture, we can split the framework into 4 sections:

- **Server Only** – These Objects only exist on the Server
- **Server & Clients** – These Objects exist on the Server and on all Clients
- **Server & Owning Client** – These Objects only exist on the Server and the owning Client
- **Owning Client Only** – These Objects only exist on the Client

“**Owning Client**” is the Player/Client, who owns the Actor.

You can see this the same way as you own your computer.

Ownership becomes important for “**RPCs**” later.

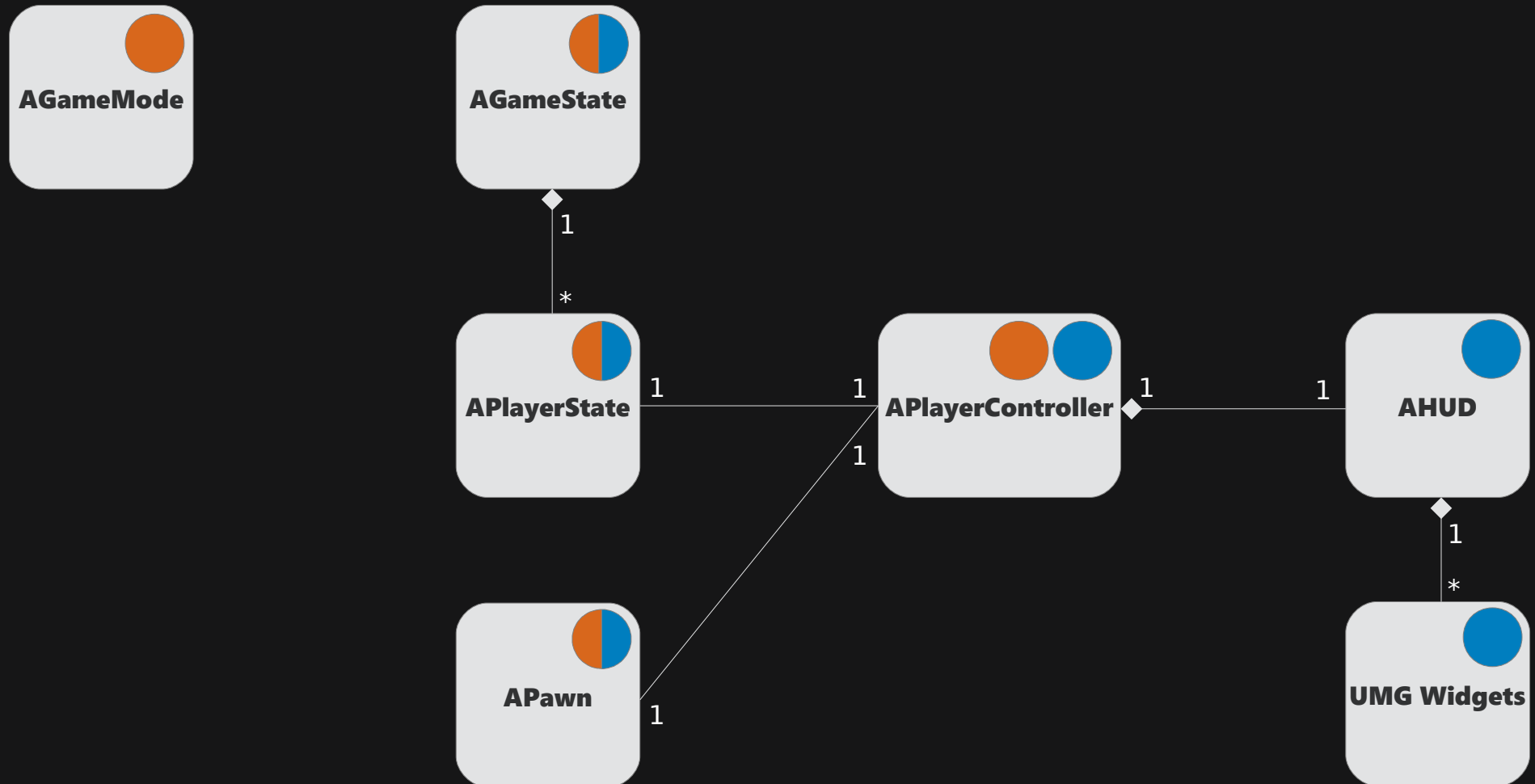
The following table, on the next page, shows you some of the common classes and in which section they exist.

Server Only

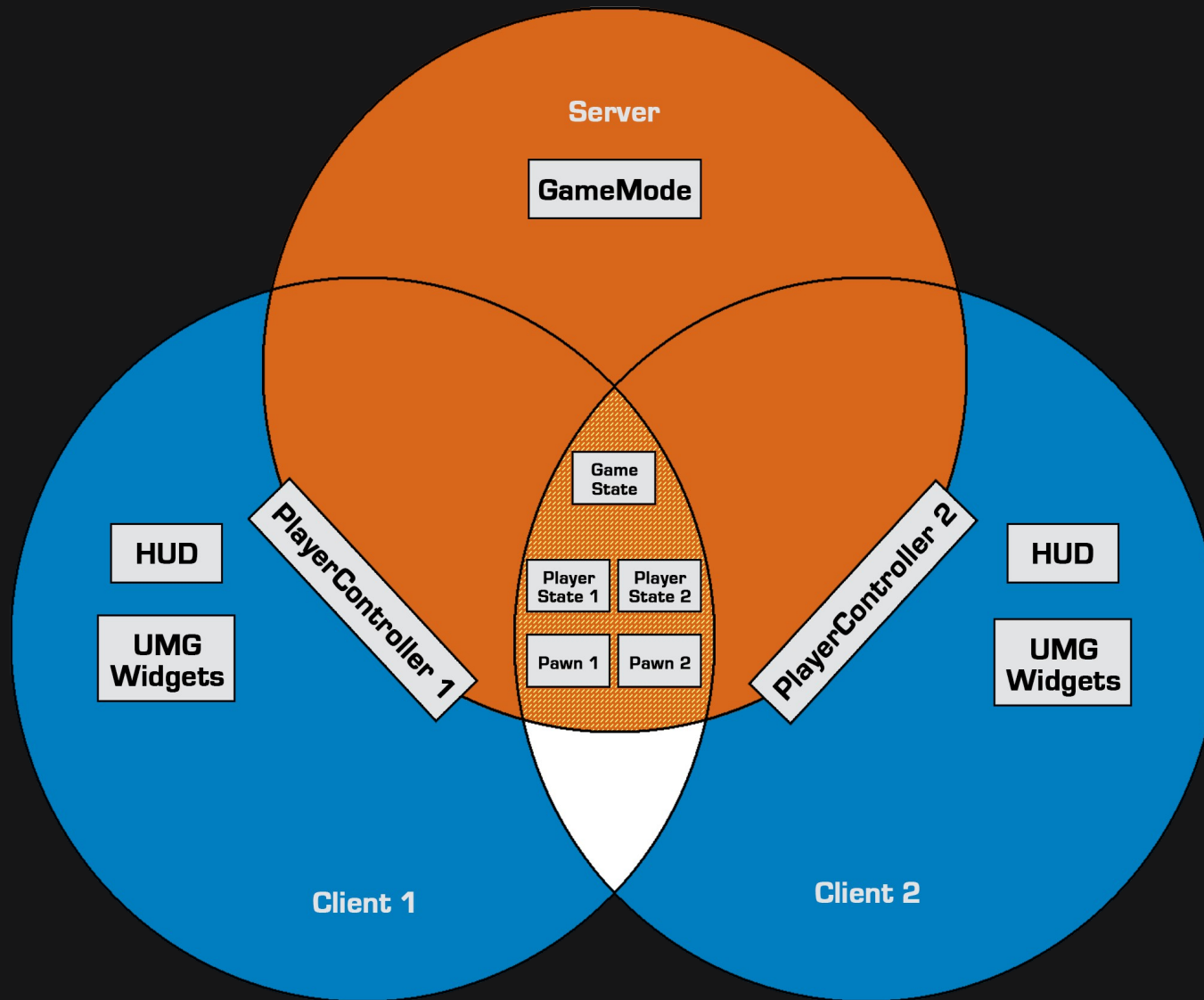
Server & Clients

Server & Owing Client

Owning Client Only



This is how some of the most important classes are laid out in the network framework. (Source 2*)



The picture on the left shows another example (Dedicated Server with 2 clients) of how objects of these classes would be distributed through the network framework:

The intersection between Client 1 and Client 2 has no objects, because they don't actually share objects that only they know about. (Source 2*)

Common Classes

The following pages will explain some of the most **Common Classes**.
They will also provide small examples of how these Classes would be used.

This list will be extended through out the whole document and is only meant to get you started, so I can freely talk about the Classes without explaining them!

The listed examples will all require knowledge about Replication.
So if you can't follow them, just ignore them for now until you read the Chapters about **Replication** etc.

Note: Some game genres might use these Classes differently.
The following examples and explanations aren't the only way to go.

Game Mode

(click name to get to the detailed API Page for this Class)

NOTE: With 4.14, the GameMode Class got split into **GameModeBase** and **GameMode**. GameModeBase has less features, because some games just might not need the full feature list of the old GameMode Class.

The class **AGameMode** is used to define the **RULES** of your game. This includes the used classes, like **Apawn**, **PlayerController**, **APlayerState** and more.

It is only available on the Server. Clients don't have an object of the GameMode and will only get a **nullptr** when trying to retrieve it.

Example:

GameModes might be known through common modes as **Deathmatch**, **Team Deathmatch** or **Capture the Flag**.

This means, that a GameMode can define things like:

- Do we have Teams or does everyone play for their own score?
- What are the winning conditions? How many kills does someone/a team need?
- How are the points achieved? Killing someone? Stealing a flag?
- What Characters will be used? What weapons are allowed? Only pistols?

Examples and Usage

In Multiplayer, the GameMode also has some interesting functions that help us managing Players and/or the general Match flow.

Blueprint

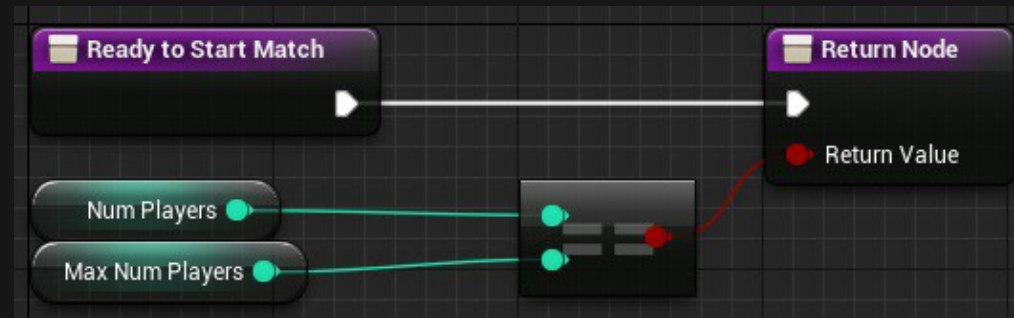
The first stop we gonna take is the “**Override Function**” section of the Blueprint Version:

Override Function

- Spawn Default Pawn For
- Should Reset
- Ready to Start Match
- Ready to End Match
- Player Can Restart
- Must Spectate
- OnSwapPlayerControllers
- OnSetMatchState
- OnRestartPlayer
- OnLogout
- OnChangeName
- Init Start Spot
- Get Default Pawn Class for Controller
- Find Player Start
- Choose Player Start
- Can Spectate

You can implement your own logic for these functions to fit specific Rules to your Game. This includes changing the way the GameMode spawns the **DefaultPawn** or on what you want to decide if the Game is **Ready to Start**:

For example checking if all Players have joined yet and are ready:



But there are also **Events** that you can use to react to certain things, which happen through out the Match.

A good example for this, which I often use, is the '**Event OnPostLogin**'.

This is called **every** time a new Player **joins** the Game.

Later you will learn more about the **Connecting Process**, but for now we will go along with it.

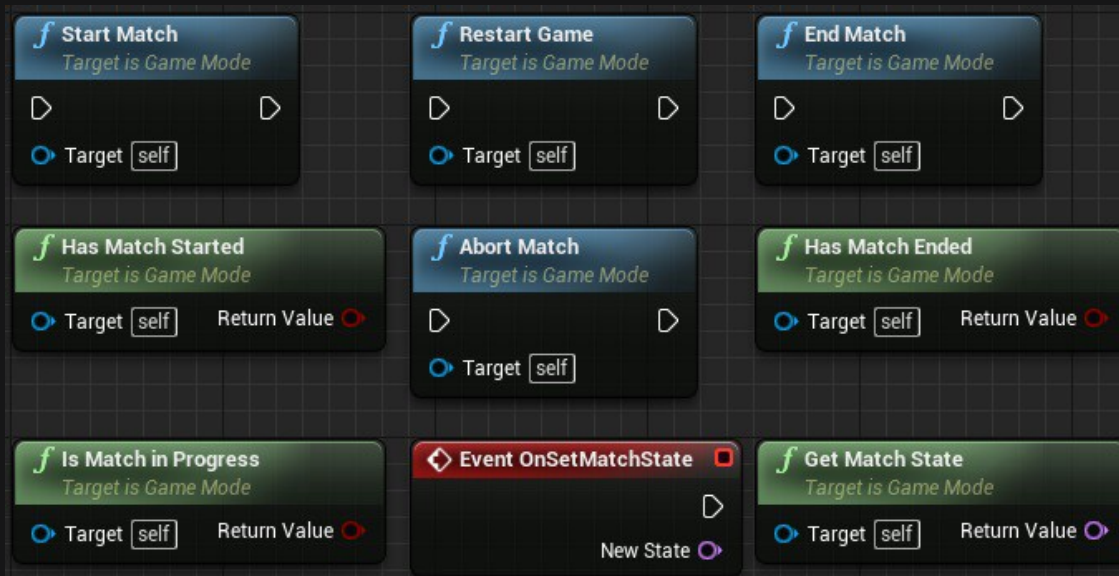
The Event passes you a valid PlayerController, that is **owned** by the connecting Player's Connection (later more about that too).

This can be used to already interact with this Player, spawn a new Pawn for him or just save his PlayerController in an Array for later.

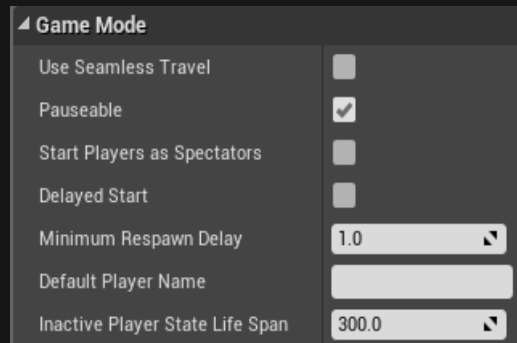
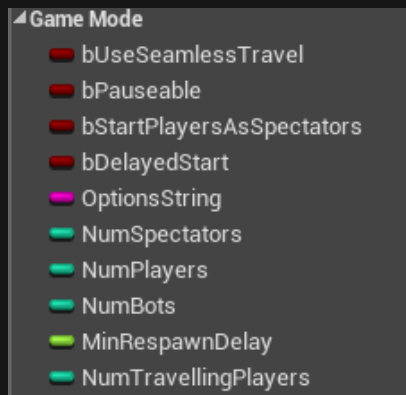


As already mentioned, you can use the GameMode to manage the general **Match Flow** of your Game. This is also linked to the functions, which you can override (like '**Ready to start Match**').

These Functions and Events can be used to control your current Match State. Most of them will be called **automatically** when the '**Ready to..**' functions return **TRUE**, but you can also use them **manually**.



'**New State**' is a simple '**FName**' Type. You could now ask, “why is this not handled in the GameState Class?” Well it is. These GameMode functions are actually working hand in hand with the GameState. This is just to give you a point to manage the State far away from the Client, since the GameMode only exists on the Server!



Of course the GameMode also has important variables you can and want to use.

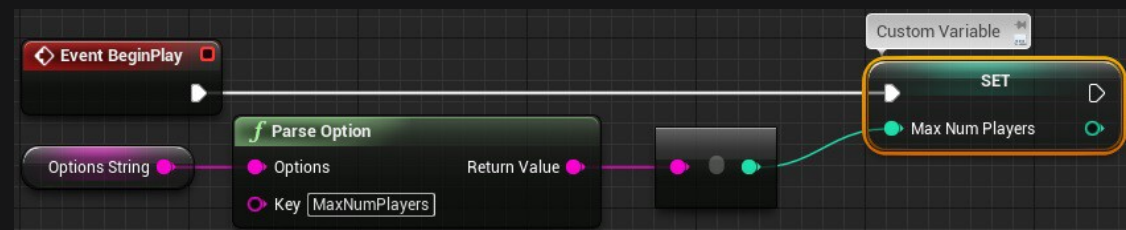
This is the list of already **inherited** Variables. Some of them can be set via the **ClassDefaults** of the GameMode Blueprint:

Most of them are pretty self-explaining, like '**Default Player Name**', which gives you the ability to give every connecting Player a default **PlayerName** that can be accessed via the PlayerState Class.

Or '**bDelayedStart**', which will keep the Game from starting, even if the '**Ready to start Match**' meets all other criteria.

One of the more important Variable is the so called '**Options String**'. These are options, separated by a '?', which you can pass via the '**OpenLevel**' function or when you call '**ServerTravel**' as a **ConsoleCommand**.

You can use '**Parse Option**' to extract passed Options, such as '**MaxNumPlayers**':



UE4++

All of the Blueprint stuff, on the previous page, can also be done in C++.

Without writing up the same information again, I will provide some code examples on how to recreate the previous Blueprint Images.

Since '**ReadyToStartMatch**' is a '**BlueprintNativeEvent**', the actual C++ Implementation of the function is called '**ReadyToStartMatch_Implementation**'. This is the one we want to override:

```
/* Header file of our GameMode Child Class inside of the Class declaration */  
// Maximum Number of Players needed/allowed during this Match  
int32 MaxNumPlayers;  
  
// Override Implementation of ReadyToStartMatch  
virtual bool ReadyToStartMatch_Implementation() override;
```

```
/* CPP file of our GameMode Child Class */  
bool ATestGameMode::ReadyToStartMatch_Implementation() {  
    Super::ReadyToStartMatch();  
  
    return MaxNumPlayers == NumPlayers;  
}
```

The '**OnPostLogin**' function is virtual and simply called '**PostLogin**' in C++.

Let's override this too:

```
/* Header file of our GameMode Child Class inside of the Class declaration */  
// List of PlayerControllers  
TArray<class APlayerController*> PlayerControllerList;  
  
// Overriding the PostLogin function  
virtual void PostLogin(APlayerController* NewPlayer) override;
```

```
/* CPP file of our GameMode Child Class */  
void ATestGameMode::PostLogin(APlayerController* NewPlayer) {  
    Super::PostLogin(NewPlayer);  
  
    PlayerControllerList.Add(NewPlayer);  
}
```

Of course, all the **Match-handling** functions can be overridden and changed too, so I won't list them here. You can check out the **API** that I linked in the **GameMode Title**.

The last C++ example for the GameMode will be the '**Options String**':

```
/* Header file of our GameMode Child Class inside of the Class declaration */  
// Maximum Number of Players needed/allowed during this Match  
int32 MaxNumPlayers;  
  
// Override BeginPlay, since we need that to recreate the BP version  
virtual void BeginPlay() override;
```

```
/* CPP file of our GameMode Child Class */  
void ATestGameMode::BeginPlay() {  
    Super::BeginPlay();  
  
    // 'FCString::Atoi' converts 'FString' to 'int32' and we use the static 'ParseOption' function of the  
    // 'UGameplayStatics' Class to get the correct Key from the 'OptionsString'  
    MaxNumPlayers = FCString::Atoi( *(UGameplayStatics::ParseOption(OptionsString, "MaxNumPlayers")) );  
}
```

Game State

(click name to get to the detailed API Page for this Class)

NOTE: With 4.14, the GameState Class got split into **AGameStateBase** and **AGameState**.

GameStateBase has less features, because some games just might not need the full feature list of the old GameState Class.

The class **AGameState** is probably the most important class for shared information between Server and Clients.

The GameState is used to keep track of the current State of the Game. This includes, for Multiplayer important, a **List of connected Players (APlayerState)**.

The GameState is replicated to all Clients. So everyone can access it. This makes the GameState to one of the most centered classes for Multiplayer Games.

While the GameMode would tell how much kills are needed to win, the GameState would keep track of the current amount of kills of each Player and/or Team!

What information you store here is completely up to you. It could be an array of scores or an array of a custom struct that you use to keep track of groups and guilds.

Examples and Usage

In Multiplayer, the GameState Class is used to keep track of the **current State** of the **Game**, which also includes the **Players** and their **PlayerStates**. The GameMode makes sure, that the **Match State** functions of the GameState are called and the GameState itself gives you the opportunity to use them on Clients too.

Compared to the GameMode, the GameState doesn't give us much to work with. But this still allows us to create our own logic, which should mostly try to **spread information** to Clients.

Blueprint



We get a few **Variables** from the base GameState Class that we can use. **PlayerArray***, as well as the **MatchState** and the **ElapsedTime** are **replicated**, so Clients can also access them.

This does not count for the **AuthorityGameMode**. Only the Server can access it since the GameMode only exists on the Server.

*The PlayerArray is not directly replicated, but every PlayerState is and they add themselves to the PlayerArray on Construction.

Also the GameState collects them once when it is created.

An additional page for the PlayerArray, which describes in C++, where it gets “replicated”.

Inside of the PlayerState Class itself:

```
void APlayerState::PostInitializeComponents() {  
    [...]  
    UWorld* World = GetWorld();  
    // Register this PlayerState with the Game's ReplicationInfo  
    if(World->GameState != NULL) {  
        World->GameState->AddPlayerState(this);  
    }  
    [...]  
}
```

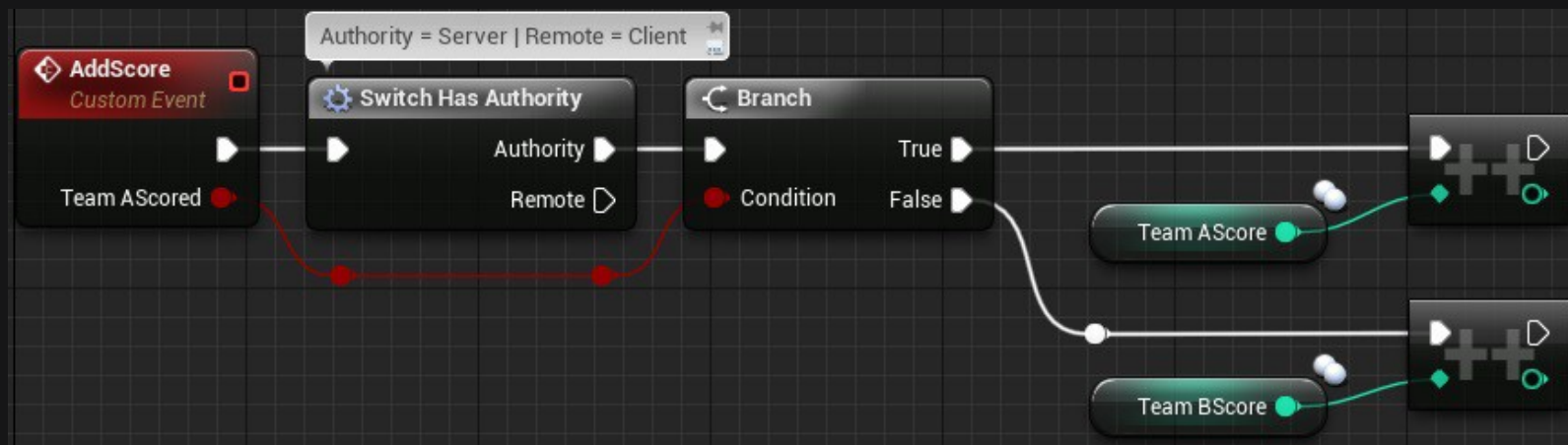
And in the GameState itself:

```
void AGameState::PostInitializeComponents() {  
    [...]  
    for(TActorIterator<APlayerState> It(World); It; ++It) {  
        AddPlayerState(*It);  
    }  
}  
  
void AGameState::AddPlayerState(APlayerState* PlayerState) {  
    if(!PlayerState->IsInactive) {  
        PlayerArray.AddUnique(PlayerState);  
    }  
}
```

All of this happens on the Server and on the Client instances of Player- and GameState!

A small example, which I could provide you with, would be **keeping track** of the **Score** of two **Teams 'A' and 'B'**. Let's say we have a **Custom Event**, which is called when a Team scores.

It passes a boolean, so we know which Team scored. Later in the “**Replication**” Part you will read about the Rule that **only** the Server can replicate Variables, so we make sure only he can call this Event. It is called from another Class (for example a Weapon that killed someone) and this should happen on the Server (always!), so we don't need an **RPC** here.



Since these variables and the GameState are replicated, you can use these two Variables and get them in any other Class you want to, for example, to display them in a **Scoreboard Widget**.

UE4++

To recreate this small example, we need a bit more Code, but despite the function itself, the Code needed to setup the Replication is only needed once per Class.

```
/* Header file of our GameState Class inside of the Class declaration */  
// You need this include to get the Replication working. Good place for it would be your Projects Header!  
#include "UnrealNetwork.h"  
  
// Replicated Specifier used to mark this variable to replicate  
UPROPERTY(Replicated)  
    int32 TeamAScore;  
UPROPERTY(Replicated)  
    int32 TeamBScore;  
  
// Function to increase the Score of a Team  
void AddScore(bool TeamAScored);
```


You will read more about this function in the Replication part!

```
/* CPP file of our GameState Class */  
// This function is required through the Replicated specifier in the UPROPERTY Macro and is declared by it  
void ATestGameState::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const {  
  
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);  
  
    DOREPLIFETIME(ATestGameState, TeamAScore);  
    DOREPLIFETIME(ATestGameState, TeamBScore);  
}
```

```
/* CPP file of our GameState Class */  
void ATestGameState::AddScore(bool TeamAScored) {  
  
    if(TeamAScored)  
        TeamAScore++;  
    else  
        TeamBScore++;  
}
```

Player State

(click name to get to the detailed API Page for this Class)

The class **APlayerState** is the most important class for a specific Player. It is meant to hold current information about the Player. Each Player has their own PlayerState. The PlayerState is also replicated to everyone and can be used to retrieve and display data on other Clients. An easy way to access all PlayerStates, that are currently in the Game, is the **PlayerArray** inside of the GameState Class.

Example information that you might want to store here:

- **PlayerName** – Current Name of the connected Player
- **Score** – Current Score of the connected Player
- **Ping** – Current Ping of the connected Player
- **GuildID** – A Guild's ID, in which the Player might be
- Or other replicated information that other Players might need to know about

Examples and Usage

In Multiplayer, the PlayerState is meant to hold information about the **State** of the **connected Player**.

This includes **Name**, **Score**, **Ping** and your custom variables.

Since this PlayerState Class is replicated, it can easily used to **retrieve data** of a Client on other Clients.

Blueprint

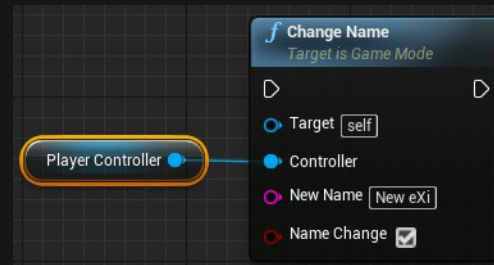
Sadly the main important functions aren't exposed to Blueprint at the time I'm writing this.

So I will only cover them in the C++ part of the PlayerState examples. But we can have a look at some of the Variables:



These variables are all replicated, so they are kept in **sync** on **all Clients**.

Sadly they are **not** easily settable in Blueprints, but nothing keeps you from creating your own versions of them.



An example on how the **PlayerName** variable can be set by calling "**ChangeName**", a **GameMode function**, and passing it the PlayerController of the Player.

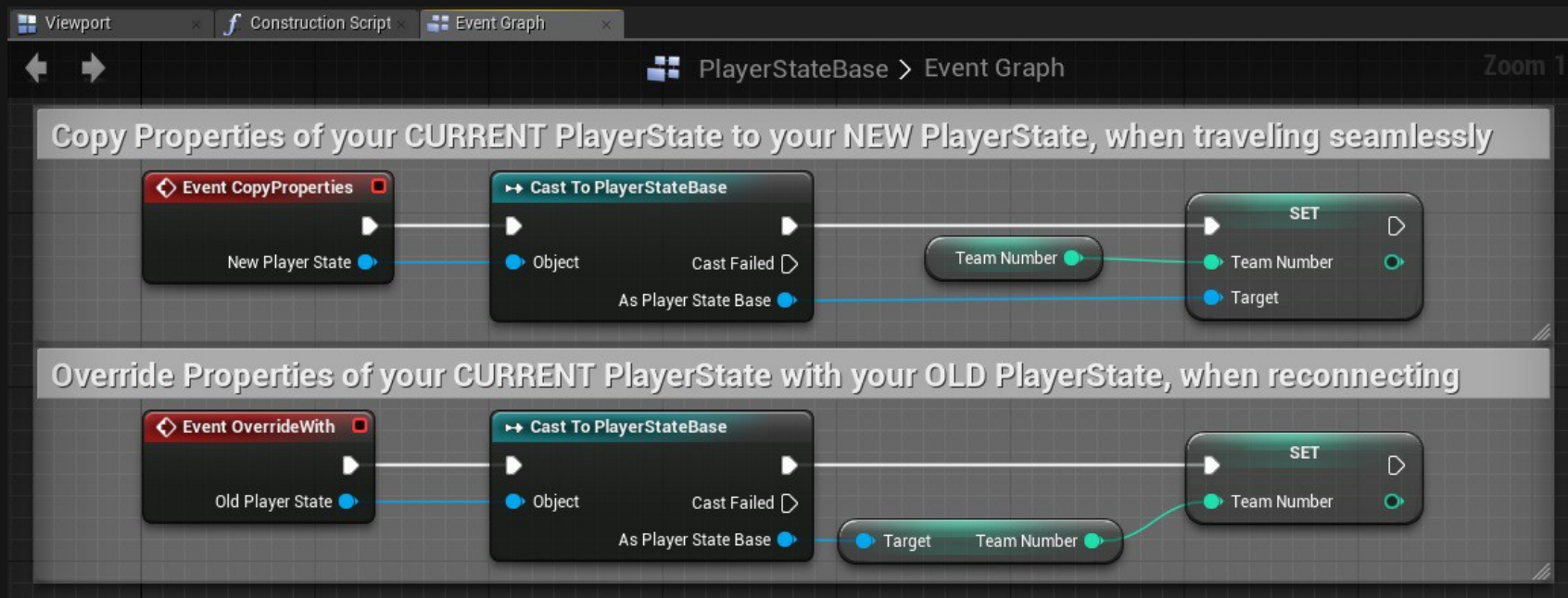
The PlayerState is also used to make sure that data stays persistent during Level Changes or unexpected connection issues.

The PlayerState has two functions dedicated to deal with **reconnecting Players** and also **Players that traveled** with the Server to a new Map.

With 4.13 released and my Pull Request accepted, these functions are now available in Blueprint Children of the PlayerState.

The PlayerState takes care of copying over the information, which it already holds, into a new PlayerState.

This either gets created through the **Level Change** or because a Player is **reconnecting**.



UE4++

The same also exists in C++, of course. Here is the

```
/* Header file of our PlayerState Child Class inside of the Class declaration */  
// Used to copy properties from the current PlayerState to the passed one  
virtual void CopyProperties(class APlayerState* PlayerState);  
  
// Used to override the current PlayerState with the properties of the passed one  
virtual void OverrideWith(class APlayerState* PlayerState);
```

These functions can be implemented into your own C++ PlayerState Child-Class to manage data that you added to your custom PlayerState. Make sure to add the “override” specifier at the end and calling “Super::” so that the original implementation stays active (if you want it to).

The implementation could look similar to this:

```
/* CPP file of our PlayerState Child Class */

void ATestPlayerState::CopyProperties(class APlayerState* PlayerState) {
    Super::CopyProperties(PlayerState);

    if(PlayerState) {
        ATestPlayerState* TestPlayerState = Cast<ATestPlayerState>(PlayerState);

        if(TestPlayerState)
            TestPlayerState->SomeVariable = SomeVariable;
    }
}

void ATestPlayerState::OverrideWith(class APlayerState* PlayerState) {
    Super::OverrideWith(PlayerState);

    if(PlayerState) {
        ATestPlayerState* TestPlayerState = Cast<ATestPlayerState>(PlayerState);

        if(TestPlayerState)
            SomeVariable = TestPlayerState->SomeVariable;
    }
}
```

Pawn

(click name to get to the detailed API Page for this Class)

The class **APawn** is the '**Actor**' that the Player actually controls. Most of the time it's a human character, but it could also be a cat, plane, ship, block, etc. The Player can only possess one Pawn at a time, but can easily switch between Pawns by un-possessing and re-possessing them.

The Pawn is mostly replicated to all Clients.

The Pawn's child class **ACharacter** is often used, because it comes with an already networked **MovementComponent**, which handles replicating the Position, Rotation, etc. of a Players Character.

Note: Not the Client is moving the Character. The Server is getting the Move-inputs from the Client and is then moving and replicating the Character!

Examples and Usage

In Multiplayer, we mostly use the **Replication** Part of the Pawn to display the Character and share some information with others. A simple example is the '**Health**' of a Character.

But we don't only replicate '**Health**' to make it visible for others, we also replicate it, so that the Server has authority over it and the Client can't cheat.

Blueprint



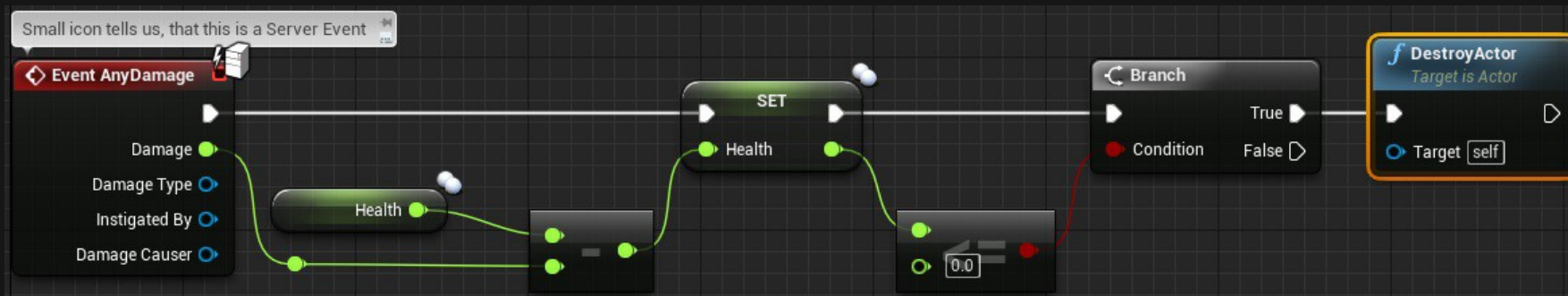
Despite the '**standard**' overridable functions, the Pawn also has two that react to possessing and un-possessing. They could be used to hide a Pawn, when he gets un-possessed:



Note: Since possessing happens on the Server, these Events are only called on the **Server**

Version if the Pawn and need a **MulticastRPCFunction** to change visibility, like in the Screenshot above!

The second example will take the '**Health**' example into account. The following picture will show how you would use the '**EventAnyDamage**' and a replicated '**Health**' variable to lower the Players health. This happens on the Server and not on the Client!



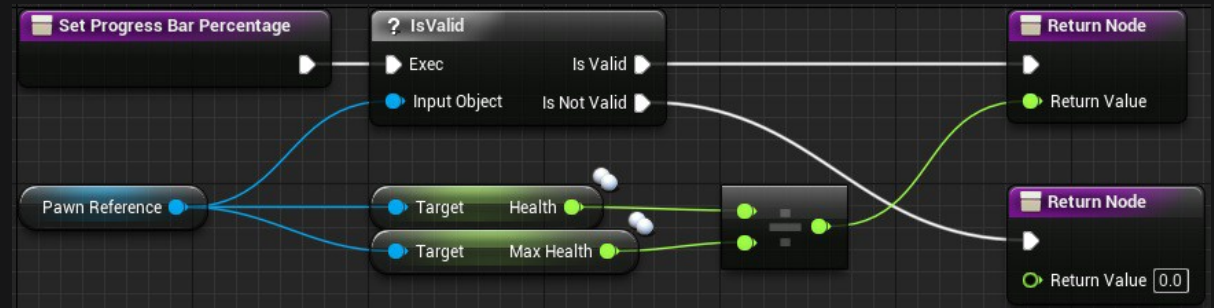
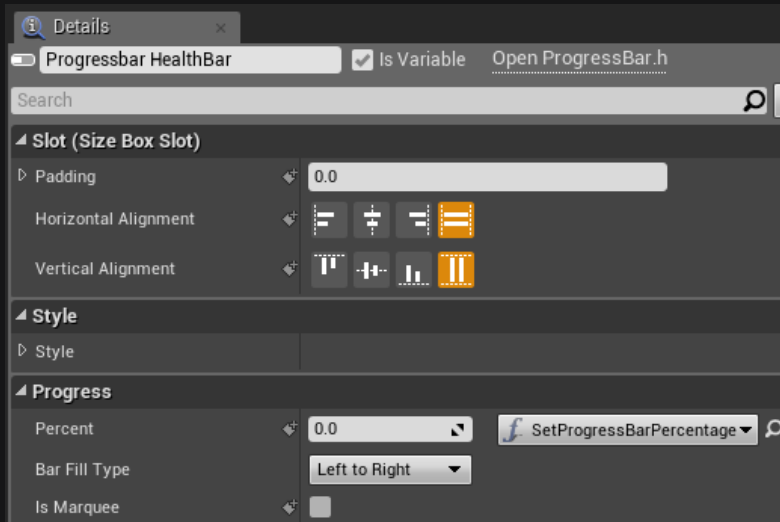
Since the Pawn should be replicated, the '**DestroyActor**' Node will also destroy the Client versions of the Actor, if called by the Server. On the Client site, we can use the replicated '**Health**' Variable for our HUD or for a **HealthBar** on top of everyone's head.

You can easily do this by creating a **Widget** with a **Progressbar** and a **Reference** of a **Pawn**.

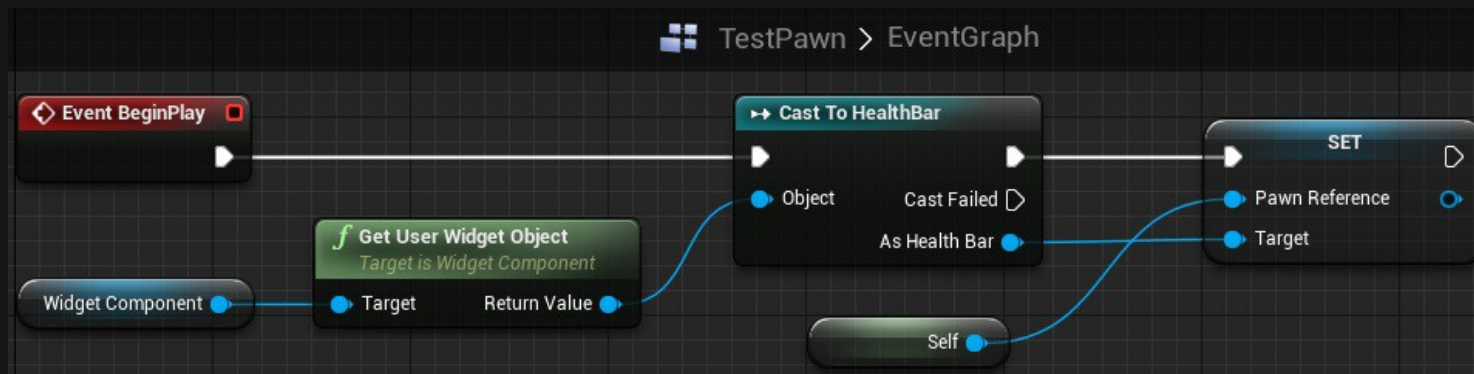
Since this is a Network Compendium, I expect you to know about Widgets, or at least to read about them somewhere else, otherwise this would be too much for this Document.

Let's assume we have a '**Health**' and '**MaxHealth**' Variable on our '**TestPawn**' Class, all set to replicate.

Now after creating the '**TestPawn**' reference Variable inside of the **Widget**, as well as the **ProgressBar**, we can bind the percentage of that bar to the following function:



Now, after setting up the **WidgetComponent**, we can set the '**Widget Class to use**', to your **HealthBar** Widget and on **BeginPlay**, do the following:



'BeginPlay' is called on **all** Instances of the Pawn, means on the Server and all Client Versions.

So **every** Instance now sets itself to the Pawn Reference of the Widget it has.

And since the Pawn and the Health Variable is replicated, we have the correct percentage above every Pawn's head.



If the **Replication** process is **not clear** at this point, it's all cool, just **keep reading** and at the end of this Document, you will understand why this works so easily!

UE4++

For the C++ examples, I won't recreate the Widget Blueprint Example. It's just too much background stuff that needs to be done to get Widgets working in C++ and I don't want to cover this here. So we will focus on the **Possess Events** and the **Damage Event**.

In C++, the two Possess Events are called:

```
virtual void PossessedBy(AController* NewController);  
  
virtual void UnPossessed();
```

The '**UnPossessed**' Event does not pass the old PlayerController though.

```
/* Header file of our Pawn Child Class, inside of the Class declaration */  
// SkeletalMesh Component, so we have something to hide  
class USkeletalMeshComponent* SkeletalMesh;  
  
// Overriding the UnPossessed Event  
virtual void UnPossessed() override;
```

We also need a **MulticastRPCFunction**. You will read about it later in the **RPC** Chapter:

```
/* Header file of our Pawn Child Class, inside of the Class declaration */  
UFUNCTION(NetMulticast, unreliable)  
    void Multicast_HideMesh();
```

```
/* CPP file of our Pawn Child Class */  
void ATestPawn::UnPossessed() {  
    Super::UnPossessed();  
  
    Multicast_HideMesh();  
}  
  
// You will read later about RPC's and why that '_Implementation' is a thing  
void ATestPawn::Multicast_HideMesh_Implementation() {  
    SkeletalMesh->SetVisibility(false);  
}
```

And we also want to recreate the **Health** example in C++. As always, if you don't understand the steps of Replication at this moment, don't worry, the **upcoming** Chapters will explain it to you.

Just skip the examples for now if they seem to complicated in terms of Replication.

The '**TakeDamage**' Function is the equivalent of the '**EventAnyDamage**' Node. To cause Damage, you normally call '**TakeDamage**' on the Actor you want to deal damage to and if that Actor implements the function, it will react to it, similar to how this example does it.

```
/* Header file of our Pawn Child Class, inside of the Class declaration */  
// Replicated Health Variable  
UPROPERTY(Replicated)  
    int32 Health;  
  
// Overriding the Damage Event  
virtual float TakeDamage(float Damage, struct FDamageEvent const& DamageEvent,  
                          AController* EventInstigator, AActor* DamageCauser) override;
```

```

/* CPP file of our Pawn Child Class */

// This function is required through the Replicated specifier in the UPROPERTY Macro and is declared by it
void ATestPawn::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const {

    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    // This actually takes care of replicating the Variable
    DOREPLIFETIME(ATestPawn, Health);
}

float ATestPawn::TakeDamage(float Damage, struct FDamageEvent const& DamageEvent,
                             AController* EventInstigator, AActor* DamageCauser) {
    float ActualDamage = Super::TakeDamage(Damage, DamageEvent, EventInstigator, DamageCauser);

    Health -= ActualDamage;        // Lower the Health of the Player

    if(Health <= 0.0)              // And destroy it if the Health is less or equal 0
        Destroy();

    return ActualDamage;
}

```

Player Controller

(click name to get to the detailed API Page for this Class)

The class **APlayerController** might be the most interesting and complicated Class that we come across. It's also the center for a lot of Client stuff, since this is the first Class that the Client actually **'owns'**.

The PlayerController can be seen as the 'Input' of the Player. It is the link of the Player with the Server. This means, every Client has one PlayerController.

A Client's PlayerController only exists on himself AND the Server, but other Clients don't know about other PlayerController.

Every Client only knows about his own!

The result of that is, that the Server has a reference of all Client PlayerControllers!

The term 'Input' does not directly mean, that all actual Input (Button Presses, Mouse Movement, Controller Axis, etc.) needs to be placed in the PlayerController.

It is a good practice to place Character specific Input (Cars work differently than Humans) into your Character/Pawn Classes and to put Input that should work with all Characters, or even when the Character Object is not valid, into your PlayerController!

An important note here is:

Input is always first passed through the PlayerController. If the PlayerController does NOT use it, it will be processed in the other Classes that might use the same Input. Consuming Input can, of course, be deactivated when needed.

Furthermore, an important thing to know is:

How do I get the correct PlayerController?

The famous node '**GetPlayerController(0)**' or code line '**UGameplayStatics::GetPlayerController(GetWorld(), 0);**' works differently on Server and Clients, but it's not really difficult.

- Calling it on the Listen-Server will return the Listen-Server's PlayerController
- Calling it on a Client will return the Client's PlayerController
- Calling it on a Dedicated Server will return the first Client's PlayerController

Other numbers than '0' will not return other Clients for a Client. This index is meant to be used for local Players (Splitscreen), which we won't cover.

Examples and Usage

Although that the PlayerController is one of the **most important** Classes for Networking, there isn't much to it by default. So we will create a small example, just to make clear why it's needed. In the Chapter about **Ownership**, you will read about why the PlayerController is important for RPCs. The following example will show you how to utilize the PlayerController to increase a replicated variable in the GameState by pressing a Widget Button.

Why do we need the **PlayerController** for this?

Well, I don't want to write down the **RPC** and **Ownership** Chapter here again, so just a short explanation:

Widgets **only** exist on the Client/Listen-Server and even if they are owned by the Client, a Server RPC has no instance on the Server to run on.

It's simply **not** replicated!

This means we need a way to get the Button Press over to the Server, so he can then increase the variable.

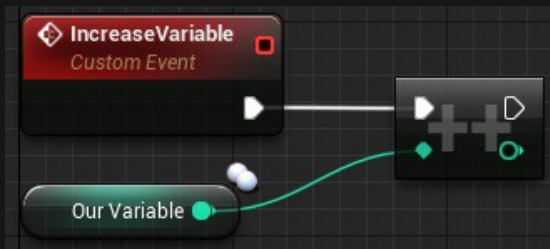
Why not calling the RPC on the GameState directly? Because it's owned by the Server.

A Server RPC needs the Client as the owner!

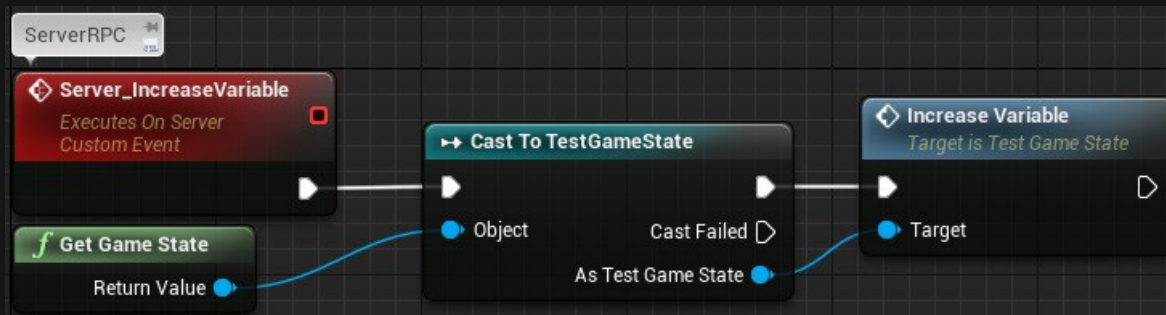
Blueprint

So first of all, we need a simple **Widget** with a **Button** that we can press. This is a **Network Compendium**, so please learn the Widget Stuff on your own. I know you can do that!

I will post the Images in the **opposite order**, so you can see where it ends and what Events actually call the Events of the previous Images. So starting with our goal, the GameState. It gets a normal Event that increments a replicated Integer variable:



This Event will get called on the **Server Side**, inside of our ServerRPC in our PlayerController:



And at last, we have our Button, which gets pressed and calls the **ServerRPC**:



So when we Click on the Button (**Client Side**), we use the ServerRPC in our PlayerController to get to the Server Side (possible, because the PlayerController is **owned** by the Client!) and then call the '**IncreaseVariable**' Event of the GameState to increment the replicated Integer.

This Integer, since it is replicated and set by the Server, will now update on all instances of the GameState and Clients can also see the update!

UE4++

For the C++ version of this example, I will replace the Widget with the **BeginPlay** of the PlayerController. This doesn't make much sense, but implementing Widgets in C++ needs some more code which I don't want to post here, since it would be a bit too much!

```
/* Header file of our PlayerController Child Class, inside of the Class declaration */  
// Server RPC. You will read more about this in the RPC Chapter  
UFUNCTION(Server,unreliable, WithValidation)  
    void Server_IncreaseVariable();  
  
// Also overriding the BeginPlay function for this example  
virtual void BeginPlay() override;
```

```
/* Header file of our GameState Child Class, inside of the Class declaration */  
// Replicated Integer Variable  
UPROPERTY(Replicated)  
    int32 OurVariable;  
  
public:  
    // Function to Increment the Variable  
    void IncreaseVariable();
```

```

/* CPP file of our PlayerController Child Class */
// Otherwise we can't access the GameState functions
#include "TestGameState.h"

// You will read later about RPC's and why that '_Validate' is a thing
bool ATestPlayerController::Server_IncreaseVariable_Validate() {
    return true;
}

// You will read later about RPC's and why that '_Implementation' is a thing
void ATestPlayerController::Server_IncreaseVariable_Implementation() {
    ATestGameState* GameState = Cast<ATestGameState>(UGameplayStatics::GetGameState(GetWorld()));

    GameState->IncreaseVariable();
}

void ATestPlayerController::BeginPlay() {
    Super::BeginPlay();

    // Make sure only the Client Version of this PlayerController calls the ServerRPC
    if(Role < ROLE_Authority)
        Server_IncreaseVariable();
}

```

```

/* CPP file of our GameState Child Class */
// This function is required through the Replicated specifier in the UPROPERTY Macro and is declared by it
void ATestGameState::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const {
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);

    // This actually takes care of replicating the Variable
    DOREPLIFETIME(ATestGameState, OurVariable);
}

void ATestGameState::IncreaseVariable() {
    OurVariable++;
}

```

That's quite some code. If you don't understand the use of some of the functions and their naming yet, **don't worry**. The **upcoming** Chapters will help you understand why it's done like this.

HUD

(click name to get to the detailed API Page for this Class)

The class **AHUD** is a Class that is only available on each Client and can be accessed through the PlayerController. It will be automatically spawned.

Before UMG (Unreal Motion Graphics) has been released, the HUD Class has been used to draw text, textures and more in the viewport of the Client.

By now, Widgets replaced the HUD Class most of the time.

You can still use the HUD Class to debug or maybe have an isolated area to manage spawning, showing, hiding and destroying Widgets.

Since the HUD isn't directly linked to Networking, examples would only show Singleplayer stuff. That's why I will skip them.

Widgets (UMG)

(click name to get to the detailed API Page for this Class)

Widgets are used in Epic Games' new UI System, called **Unreal Motion Graphics**.

They inherit from **Slate**, which is a language used to create UI within C++ and also used for the Unreal Engine 4 Editor itself.

Widgets are only available locally on Client (Listen-Server).

They are **NOT** replicated and you always need a separated, replicated Class to perform replicated actions through, for example, Button-Presses.

To learn more about UMG and Widgets, please use the provided API Link above.

We already had a small example for the Widgets usage inside of the APawn examples. So I will skip them here.

Dedicated vs Listen Server

Dedicated Server

A **Dedicated Server** is a standalone Server that does **NOT** require a Client.

It runs separated from the Game Client and is mostly used to have a Server running that Players can always join/leave.

Dedicated Servers can be compiled for **Windows** and **Linux** and can be run on **Virtual Servers** that Players can connect to via a fixed IP-Address.

Dedicated Servers have no visual part, so they don't need a UI, nor do they have a PlayerController. They also have no Character or similar representing them in the Game.

Listen-Server

A **Listen-Server** is a Server that is also a Client.

This means, that the Server always has a at least **ONE** Client connected.

This Client is called the Listen-Server and if he disconnects, the Server will shutdown.

Due to being also a Client, the Listen-Server needs UI and has a PlayerController, which represents the Client part. Getting '**PlayerController(0)**' on a Listen-Server, will return the PlayerController of that very Client.

Since the Listen-Server runs on the Client itself, the IP that others need to connect to is the one of the Client. Compared to the Dedicated Server, this often comes with the problem of Internet Users **NOT** having a static IP.

But using an **OnlineSubsystem** (explained later), the changing IP problem can be solved.

Replication

What is 'Replication'?

Replication is the act of the Server passing information/data to the Clients.

This can be limited to specific entities and groups. Blueprints mostly perform Replication according to the Settings of the affected **AActor**.

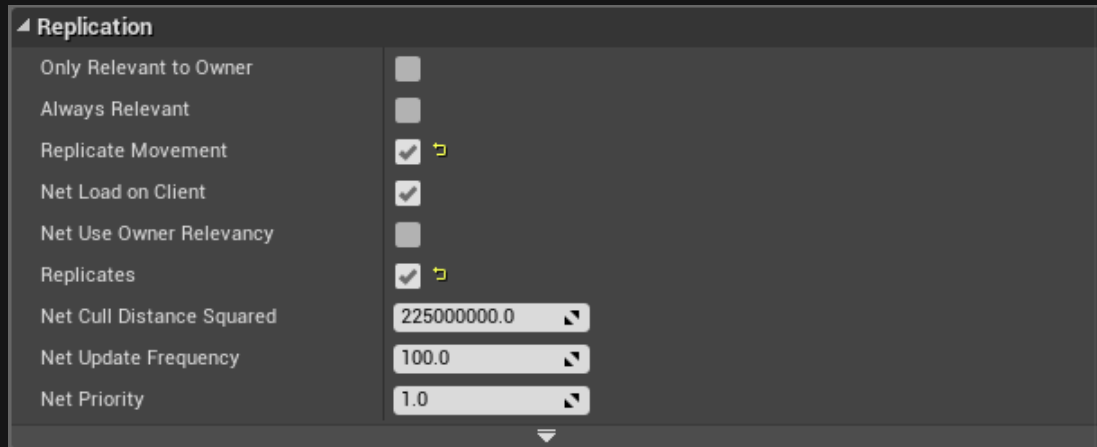
The first Class, which is capable of replicating properties, is the Actor.

All of the before mentioned Classes inherit from Actor at some point, giving them the ability to replicate properties if needed.

Though not all of them do this the same way.

The GameMode, for example, **doesn't** replicate at all and **only** exists on the Server.

How to use 'Replication':

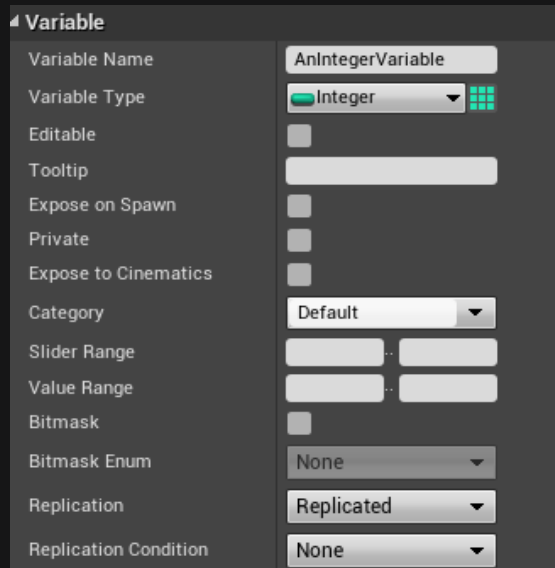


Replication can be activated in the **Class Defaults/Constructor** of an Actor child:

```
ATestCharacter::ATestCharacter() {  
    // Network game, so let's setup Replication  
    bReplicates = true;  
    bReplicateMovement = true;  
}
```

An Actor with '**bReplicates**' set to TRUE will be spawned and replicated on all Clients when spawned by the Server. And **ONLY** when spawned by the Server. IF a Client spawns this Actor, the Actor will **ONLY** exist on this very Client.

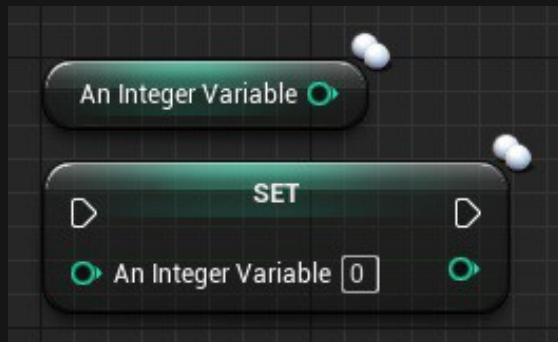
Replicating Properties



So, when Replication is enabled, we can replicate variables. There are multiple ways to do this. We will start with the most basic one:

Setting the “**Replication**” Drop-Down menu to “**Replicated**” will make sure this variable gets replicated to all replicated Instances of this Actor. Of course this only works for Actors that are set to replicate.

With 4.14, variables can now be replicated under certain condition, even in Blueprints. To read more about the conditions, scroll further.



Replicated Variables are marked with **2** white circles.

Replicating a variable in C++ is slightly more work at the beginning.
But it also allows us to specify who we want to replicate this variable to.

```
/* Header file inside of the Classes declaration */  
// Create replicated health variable  
UPROPERTY(Replicated)  
float Health;
```

The .cpp file will get this '**GetLifetimeReplicatedProps**' function. The header declaration, of that function, is already provided through some UE4 macro, so we don't need to care about that one. Here you actually define the rules of replicating your variables.

```
void ATestPlayerCharacter::GetLifetimeReplicatedProps(TArray<FLifetimeProperty>& OutLifetimeProps) const {  
    Super::GetLifetimeReplicatedProps(OutLifetimeProps);  
  
    // Here we list the variables we want to replicate + a condition if wanted  
    DOREPLIFETIME(ATestPlayerCharacter, Health);  
}
```

You can also do a conditional replication here:

```
// Replicates the Variable only to the Owner of this Object/Class  
DOREPLIFETIME_CONDITION(ATestPlayerCharacter, Health, COND_OwnerOnly);
```

Condition	Description
COND_InitialOnly	This property will only attempt to send on the initial bunch
COND_OwnerOnly	This property will only send to the Actor's owner
COND_SkipOwner	This property send to every connection EXCEPT the owner
COND_SimulatedOnly	This property will only send to simulated Actors
COND_AutonomousOnly	This property will only send to autonomous Actors
COND_SimulatedOrPhysics	This property will send to simulated OR bRepPhysics Actors
COND_InitialOrOwner	This property will send on the initial packet , or to the Actor's owner
COND_Custom	This property has no particular condition, but wants the ability to toggle on/off via SetCustomIsActiveOverride

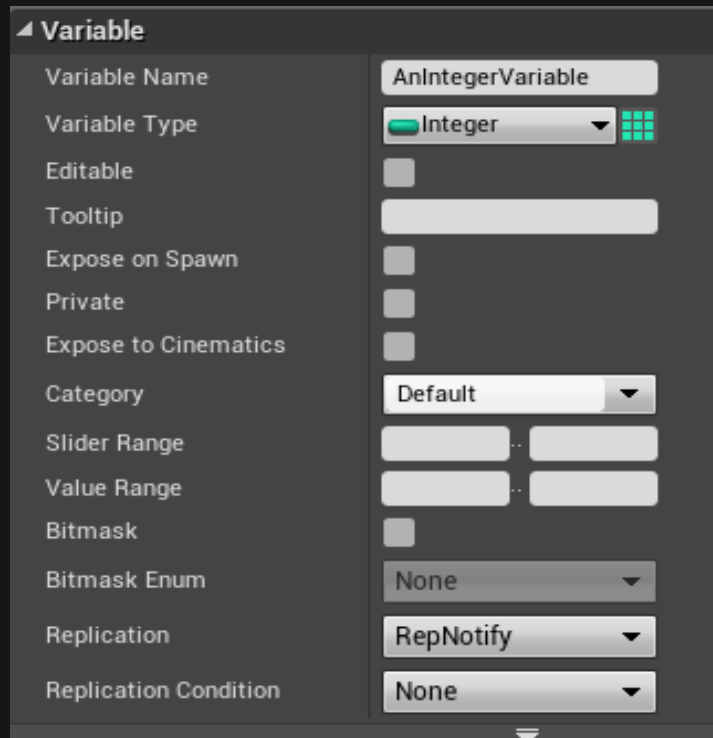
It's important that you understand that the whole replication process only works **from Server to Client** and **NOT** the other way round.

We will learn later how to get the Server to replicate something, that the Client wants to share with others (for example: **PlayerName**).

A different version of Replication is the so called “**RepNotify**” version.

It makes use of a function that will be called on **all instances** when receiving the updated value.

With this, you can call logic that needs to be called **AFTER** the Value has been replicated.



In Blueprints, this function will be created **automatically** once you select “**RepNotify**” in the Replication Drop-Down menu:



The C++ Version needs a bit more, but works the same:

```
/* Header file inside of the Classes declaration */  
// Create RepNotify Health variable  
UPROPERTY(ReplicatedUsing=OnRep_Health)  
float Health;  
  
// Create OnRep Function | UFUNCTION() Macro is important! | Doesn't need to be virtual though  
UFUNCTION()  
virtual void OnRep_Health();
```

```
/* CPP file of the Class */  
void ATestCharacter::OnRep_Health() {  
    if(Health < 0.0f)  
        PlayDeathAnimation();  
}
```

With '**ReplicatedUsing=FUNCTIONNAME**', we specify the function that should get called when the Variable is successfully replicated. This functions needs to have the '**UFUNCTION()**' macro, even if it's empty!

Remote Procedure Calls

Other ways for Replication are so called “**RPC**”s. Short form for “**R**emote **P**rocedure **C**all”.

They are used to call something on another instance. Your TV Remote does the same with your Television.

UE4 uses it to call functions from **Client** to **Server**, **Server** to **Client** or **Server** to **a specific group**.

These RPCs can't have a return value! To return something, you need to use a second RPC.

But this only works under certain rules.

They are listed in this table, which can be found in the official Documentation:

- **Run on Server** – Is meant to be executed on the Server Instance of the Actor
- **Run on owning Client** – Is meant to be executed on the Owner of this Actor
- **NetMulticast** – Is meant to be executed on all Instances of this Actor

Requirements and Caveats

There are a few requirements that need to be met for RPCs to be completely functional:

1. They must be called from Actors.
2. The Actor must be **replicated**.
3. If the RPC is being called from **Server** to be executed on a **Client**, only the Client who actually **owns** that Actor will execute the function.
4. If the RPC is being called from **Client** to be executed on the **Server**, the Client must **own** the Actor that the **RPC** is being called on.
5. **Multicast RPCs** are an exception:
 - If they are called from the Server, the Server will execute them locally, as well as execute them on all **currently connected Clients**.
 - If they are called from Clients, they will only execute locally, and will not execute on the Server.
 - For now, we have a simple throttling mechanism for Multicast events:
 - A Multicast function will not replicate more than twice in a given.
 - Actor's network update period. Long term, we expect to improve on this.

RPC invoked from the Server

Actor Ownership	Not replicated	NetMulticast	Server	Client
Client-owned Actor	Runs on Server	Runs on Server and all Clients	Runs on Server	Runs on Actor's owning Client
Server-owned Actor	Runs on Server	Runs on Server and all Clients	Runs on Server	Runs on Server
Unowned Actor	Runs on Server	Runs on Server and all Clients	Runs on Server	Runs on Server

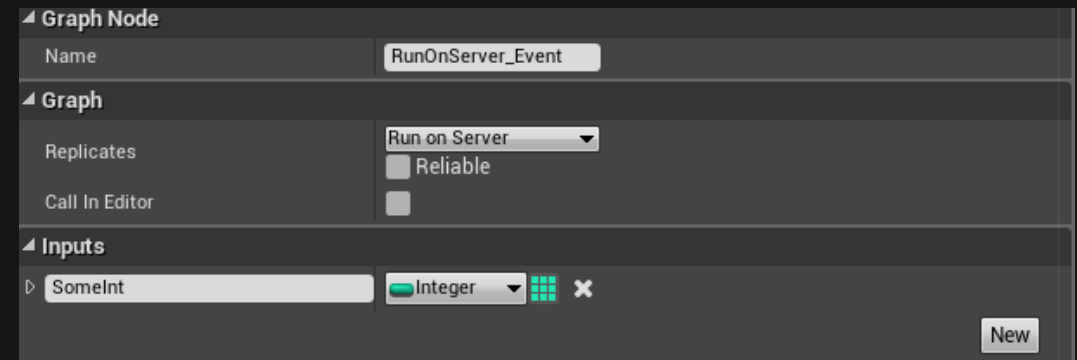
RPC invoked from a Client

Actor Ownership	Not replicated	NetMulticast	Server	Client
Owned by invoking Client	Runs on invoking Client	Runs on invoking Client	Runs on Server	Runs on invoking Client
Owned by a different Client	Runs on invoking Client	Runs on invoking Client	Dropped	Runs on invoking Client
Server-owned Actor	Runs on invoking Client	Runs on invoking Client	Dropped	Runs on invoking Client
Unowned Actor	Runs on invoking Client	Runs on invoking Client	Dropped	Runs on invoking Client

RPCs in Blueprints

RPCs in Blueprints are created by creating **CustomEvents** and setting them to **Replicate**.

RPCs can't have a return value! So functions can't be used to create them.



The '**Reliable**' check box can be used to mark the RPC as '**important**', making sure it will 99.99% executed and not dropped due to connection issues or so.

Note: Don't mark every RPC as **Reliable**!

RPCs in C++

To use the whole Network stuff in C++, you need to include “**UnrealNetwork.h**” in your project Header!
RPCs in C++ are relatively easy to create, we only need to add the specifier to the **UFUNCTION()** Macro.

```
// This is a Server RPC, marked as unreliable and WithValidation (is needed!)  
UFUNCTION(Server,unreliable, WithValidation)  
void Server_PlaceBomb();
```

The CPP file will implement a different function. This one needs '**_Implementation**' as a suffix.

```
// This is the actual implementation (Not Server_PlaceBomb). But when calling it, we use "Server_PlaceBomb"  
void ATestPlayerCharacter::Server_PlaceBomb_Implementation() {  
    // BOOM!  
}
```

The CPP file also needs a version with '**_Validate**' as a suffix. Later more about that.

```
bool ATestPlayerCharacter::Server_PlaceBomb_Validate() {  
    return true;  
}
```

The other two types of RPCs are created like this:

Client RPC, which needs to be marked as '**reliable**' or '**unreliable**'!

```
UFUNCTION(Client,unreliable)  
void ClientRPCFunction();
```

and **Multicast RPC**, which also needs to be marked as '**reliable**' or '**unreliable**'!

```
UFUNCTION(NetMulticast,unreliable)  
void MulticastRPCFunction();
```

Of course, we can also add the '**reliable**' keyword to an RPC to make it reliable.

```
UFUNCTION(Client,reliable)  
void ReliableClientRPCFunction();
```


Validation (C++)

The idea of **Validation** is, that if the Validation function for an RPC detects that any of the parameters are bad, it can notify the system to **disconnect** the Client/Server who initiated the RPC call.

Validation is required by now for every ServerRPCFunction. The '**WithValidation**' keyword in the **UFUNCTION** Macro is used for that.

```
UFUNCTION(Server,unreliable,WithValidation)
void SomeRPCFunction(int32 AddHealth);
```

Here is an example of how the '**_Validate**' Function can be used:

```
bool ATestPlayerCharacter::SomeRPCFunction_Validate(int32 AddHealth) {
    if(AddHealth > MAX_ADD_HEALTH) {
        return false;           // This will disconnect the caller!
    }
    return true;                // This will allow the RPC to be called!
}
```

Note: Client to Server RPCs require the '**_Validate**' function to encourage secure Server RPC functions and to make it as easy as possible for someone to add code to check each and every parameter to be valid against all the known input constraints!

Ownership

So, **Ownership** is something important to understand. You already saw the table, which contained stuff like “**Client-owned Actor**”.

Server or Clients can 'own' an Actor.

An example is the PlayerController, which is owned by the Client (or Listen-Server).

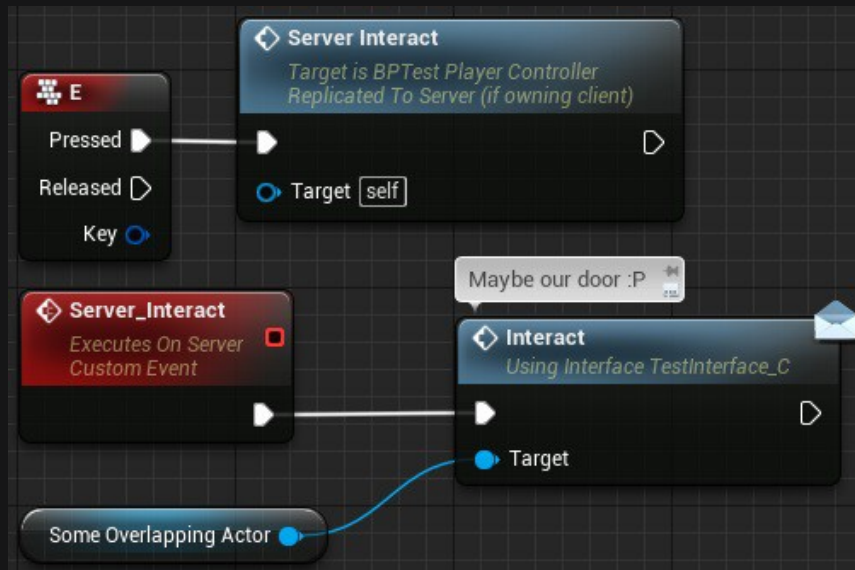
Another example would be a spawned/placed Door in the Scene. This will mostly be owned by the Server.

But why is this a problem?

If you check out the table again, you will notice that, for example, a Server RPC will be dropped if a Client calls it on an Actor that he does **NOT** own.

So the Client can't call “**Server_OpenDoor**” on the Server owned door. But how do we work around that?

We use a Class/Actor, that the Client actually owns. And this is where the PlayerController starts to shine.



So instead of trying to enable Input on the Door and calling a Server RPC there, we create the Server RPC in the PlayerController and let the Servercall an Interface function on the Door (for example: **Interact()**).

If the Door implements the correct Interface, it will then call the defined logic and hopefully replicate the open/closed status correctly.

Note: Interfaces are nothing Multiplayer specific and I highly recommend to look them up if you don't understand what they are!

Actors and their Owning Connections

As already mentioned in the Class overview, the PlayerController is the first Class that the Player actually '**owns**', but what does that mean?

Each **Connection** has a PlayerController, created specifically for that Connection. A PlayerController, that is created for that reason, is owned by that Connection. So when we determine if an Actor is owned by someone, we actually query up to the **most outer** Owner and if this is a PlayerController, then the Connection that owns the PlayerController also owns that Actor.

Kinda simple, or? So what would be an example for that?

The Pawn/Character. They are possessed by the PlayerController and at this time, the PlayerController is the Owner of the possessed Pawn. This means, the Connection that owns this PlayerController also owns the Pawn.

This is only the case **WHILE** the PlayerController is possessing the Pawn. Un-possessing it will result in the Client no longer owning the Pawn.

So why is this important and for what do I need this?

- RPCs need to determine **which** Client will execute a Run-On-Client RPC
- Actor **Replication** and connection **Relevancy**
- Actor **property Replication** conditions when the Owner is involved

You already read that RPCs react differently when getting called by Client/Server depending on the Connection they are owned by.

You also read about **conditional Replication**, where (in C++) variables are only replicated under a certain condition.

The following topic describes the Relevancy Part of the list.

Actor Relevancy and Priority

Relevancy

So, what is '**Relevancy**' and why do we need that?

Imagine having a Game with Levels/Maps big enough that Players could actually be '**un-important**' towards others.

Why would **Player A** need to see the mesh of **Player B** if they are miles apart from each other?

To improve bandwidth, **Unreal's Network Code** makes it possible that the Server only tells Clients about Actors that are in that Client's **relevant set**.

Unreal applies the following rules (in order) to determine the **relevant set** of Actors for a Player.

These tests are implemented in the virtual function '**AActor::IsNetRelevantFor()**'.

1. If the Actor is '**bAlwaysRelevant**', is owned by the Pawn or PlayerController, is the Pawn, or the Pawn is the Instigator of some action like noise or damage, it is **relevant**
2. If the Actor is '**bNetUserOwnerRelevancy**' and has an **Owner**, use the **Owner's relevancy**
3. If the Actor is '**bOnlyRelevantToOwner**', and does not pass the first check, it is **not relevant**
4. If the Actor is **attached** to the **Skeleton** of another Actor, then its **relevancy** is determined by the relevancy of its base
5. if the Actor is hidden ('**bHidden == true**') and the **root component** does **not collide** then the Actor is **not relevant**
 - If there is no **root component**, '**AActor::IsNetRelevantFor()**' will log a warning and ask if the Actor should be set to '**bAlwaysRelevant = true**'
6. If '**AGameNetworkManager**' is set to use **distance based relevancy**, the Actor is **relevant** if it is closer than the **net cull distance**

Note: Pawn and PlayerController override '**AActor::IsNetRelevantFor()**' and have different conditions for relevancy as a result.

Prioritization

Unreal uses a **load-balancing technique** that **prioritizes** all Actors, and gives each one a fair share of the bandwidth based on how important it is to Gameplay.

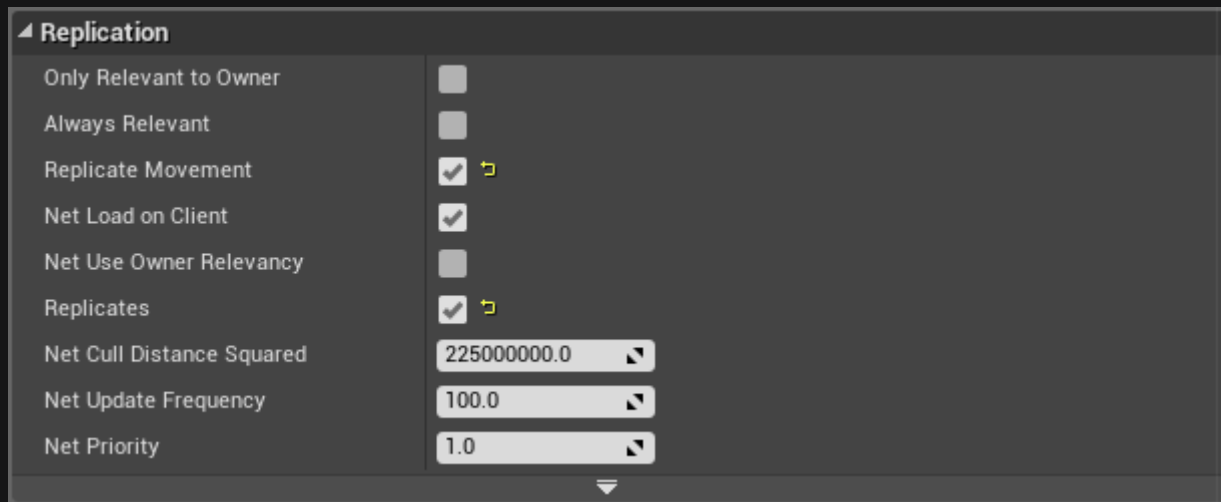
Actors have a float variable called “**NetPriority**”. The **higher** this number, the **more bandwidth** that Actor receives relative to others. An Actors with a “**NetPriority**” of **2.0** will update exactly **twice** as frequently as an Actor with a “**NetPriority**” of **1.0**.

The **only** thing that matters with priorities is their **Ratio**. So obviously you cannot improve **Unreal's Network Performance** by increasing all of the priorities.

The current priority of an Actor is calculated using the virtual function '**AActor::GetNetPriority()**'.

To avoid starvation '**AActor::GetNetPriority()**' multiplies '**NetPriority**' with the time since the Actor was last replicated.

The '**GetNetPriority**' functions also consider the **relative location** of and distance between the **Actor** and the **Viewer**.



Most of these **Settings** can be found in the **Class Defaults** of the **Blueprint** and, of course, be setup inside the C++ Class of each Actor Child.

```
bOnlyRelevantToOwner = false;  
bAlwaysRelevant = false;  
bReplicateMovement = true;  
bNetLoadOnClient = true;  
bNetUseOwnerRelevancy = false;  
bReplicates = true;  
  
NetUpdateFrequency = 100.0f;  
NetCullDistanceSquared = 225000000.0f;  
NetPriority = 1.0f;
```

Actor Role and RemoteRole

We have two more **important properties** for Actor Replication.

These two properties tell you:

- Who has **Authority** over the Actor
- Whether the Actor is **replicated** or **not**
- The **Mode** of the Replication

First thing we want to determine, is who has Authority over a specific Actor.

To determine if the current running instance of the Engine has Authority, check that the **Role** property is **ROLE_Authority**. If it is, then this instance of the Engine is in charge of this Actor (whether it's replicated or not).

Note: This is not the same as Ownership!

Role/RemoteRole Reversal

Role and **RemoteRole** could be reversed depending on who is inspecting these values.

For example, if on the Server you have this configuration:

- `Role == Role_Authority`
- `RemoteRole = ROLE_SimulatedProxy`

Then the Client would see it as this:

- `Role == ROLE_SimulatedProxy`
- `RemoteRole == ROLE_Authority`

This makes sense, since the Server is in charge of the Actor and replicating it to Clients.
The Clients are just supposed to receive updates, and simulate the Actor between updates.

Mode of Replication

The Server doesn't update Actors every update. This would take way too much bandwidth and CPU resources. Instead, the Server will replicated Actors at a frequency specified on the '**AActor::NetUpdateFrequency**' property.

This means, that some time will pass on the Client between Actor updates. This could result in Actors looking sporadic or choppy in their movement. To compensate for this, the Client will simulate the Actor between updates.

There are currently two types of simulation that occur:

ROLE_SimulatedProxy

and

ROLE_AutonomousProxy

ROLE_SimulatedProxy

This is the standard simulation path, and is generally based on extrapolating movement based on the last known velocity.

When the Server sends an update for a particular Actor, the Client will adjust its position towards the new location, and then in between updates, the Client will continue to move the Actor based on the most recent velocity sent from the Server.

Simulating using last known velocity is just one example of how general simulation works.

There is nothing stopping you from writing custom code to use some other information to extrapolate between Server updates.

ROLE_AutonomousProxy

This is generally only used on Actors that are possessed by PlayerControllers.

This just means that this Actor is receiving inputs from a human controller, so when we extrapolate, we have a bit more information, and can use actual human inputs to fill in the missing info (rather than extrapolating based on the last known velocity).

Traveling in Multiplayer

Non-/Seamless travel

The difference between '**Seamless**' and '**Non-seamless**' travel is simple.

'**Seamless**' travel is a non-blocking operation, while '**Non-seamless**' will be a blocking call.

A '**Non-seamless**' travel for a Client means, that he disconnects from the Server and then re-connects to the same Server, which will have the new Map ready to load.

Epic recommends to use '**Seamless**' travel as often as possible, since this will result in a smoother experience, and will avoid any issues that can occur during the re-connection process.

There are three ways in which a '**Non-seamless**' travel must occur:

- When loading a Map for the first time
- When connecting to a Server for the first time as a Client
- When you want to end a Multiplayer Game, and start a new one

Main Traveling Functions

There are three main functions that drive traveling:

UEngine::Browse

- Is like a hard reset when loading a new Map
- Will always result in a '**Non-seamless**' travel
- Will result in the Server disconnecting current Clients before traveling to the destination Map
- Clients will disconnect from current Server
- Dedicated Server cannot travel to other Servers, so the Map must be local (cannot be URL)

UWorld::ServerTravel

- For the Server only
- Will jump the Server to a new World/Level
- All connected Clients will follow
- This is the way Multiplayer Games travel from Map to Map, and the Server is the one in charge to call this function
- The Server will call '**APlayerController::ClientTravel**' for all Client Players that are connected

APlayerController::ClientTravel

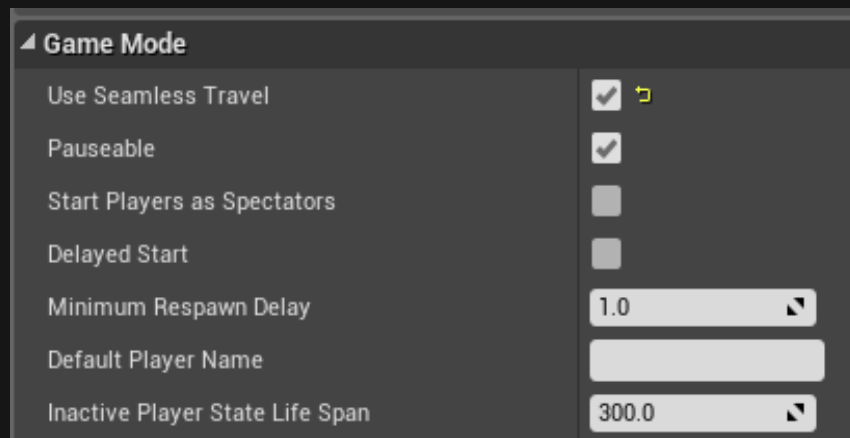
- If called from a Client, will travel to a **new** Server
- If called from a Server, will instruct the particular Client to travel to the new Map (But stay connected to the current Server)

Enabling Seamless Travel

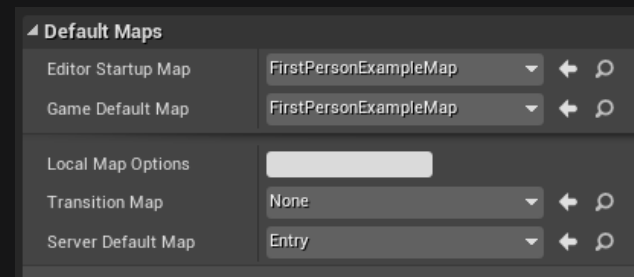
'**Seamless**' travel comes with a **Transition Map**. This is configured through the '**UGameMapsSettings::TransitionMap**' property. By default this property is empty. If your Game leaves this property empty, an empty Map will be created for it.

The reason the Transition Map exists, is that there must always be a World loaded (which holds the Map), so we can't free the old Map before loading the new one. Since Maps can be very large, it would be a bad idea to have the old and new Map in memory at the same time, so this is where the Transition Map comes in.

Once you have the transition map setup, you set '**AGameMode::bUseSeamlessTravel**' to true, and from there '**Seamless**' travel should work!



This can, of course, also be Setup via a GameMode Blueprint and via the **Project Settings** in the “**Maps and Nodes**” Tab.



Persisting Actors / Seamless Travel

When using '**Seamless**' travel, it's possible to carry over (persist) Actors from the current Level to the new one. This is useful for certain Actors, like Inventory Items, Players, etc.

By default, these Actors will persist automatically:

- The GameMode Actor (Server only)
 - Any Actors further added via '**AGameMode::GetSeamlessTravelActorList**'
- All Controllers that have a **valid** PlayerState (Server only)
- All PlayerControllers (Server only)
- All **local** PlayerControllers (Server and Client)
 - Any Actors further added via '**APlayerController::GetSeamlessTravelActorList**' called on **local** PlayerControllers

Here is the general flow when executing 'Seamless' travel:

1. Mark Actors that will persist to the Transition Level (read above)
2. Travel to the Transition Level
3. Mark Actors that will persist to the Final Level (read above)
4. Travel to the Final Level

Online Subsystem Overview

The **Online Subsystem** and its **Interfaces** exist to provide a clear abstraction to common online functionality across the available set of platforms in a given environment.

Platforms in this context would refer to **Steam**, **Xbox Live**, **Facebook**, etc. **Portability** is one of the main goals.

For what do I need a Subsystem?

By default you will use the '**SubsystemNULL**'. This allows you to host **LAN** Sessions (so you can find Sessions via a Server List and join them in your LAN network) or join directly via **IP**. What it doesn't allow you, is to host such a Session on the Internet.

Why?

Because you have no **Master Server** that provides the Client with a list of Servers/Sessions.

Subsystems, like for example Steam, will allow you to also host Servers/Session that are visible over the Internet. This is important to understand!

You can also create your own Subsystem/Master Server, but that requires a lot of coding outside of UE4.

Online Subsystem Module

Basic Design

The base module **Online Subsystem** is responsible for regulating how platform specific modules are defined and registered with the Engine. All access to a platform service will go through this module. When loaded, this module will in turn try to load the default platform server module specified in the '**Engine.ini**' – file:

```
[OnlineSubsystem]  
DefaultPlatformService = <Default Platform Identifier>
```

If successful, this default online interface will be available via the static accessor when no parameter is specified.

```
static IOnlineSubsystem* Get(const FName& SubsystemName = NAME_None);
```

Additional services are loaded on demand when called from this function with a proper identifier.

Use of Delegates

Much like **Unreal Engine 3 (UE3)**, the Online Subsystem will make heavy use of **Delegates** when calling functions with **Asynchronous** side effects. It is important to respect the Delegates and wait for the proper Delegate to be called before calling functions further down the chain.

Failure to wait for an Asynchronous task can cause crashes and unexpected, undefined behavior. Waiting for Delegates is especially important during connectivity failures such as a cable pull or other disconnect events. The amount of time a task may take to finish may seem instantaneous in the ideal case, but can be upwards to almost a minute in the timeout case.

The Delegate interface is fairly straightforward, with each Delegate clearly defined at the top of each interface header. Every Delegate has an **Add**, **Clear**, and **Trigger** function. (Although **Triggering** delegates **manually** is **discouraged**).

Common practice is to **Add()** the **Delegate** right before calling the appropriate function, and then **Clear()** the **Delegate** from within itself.

Interfaces

Not all platforms will implement all **Interfaces** and Game Code should plan accordingly.

Profile

Interface definition for the online services **Profile** services. Profile services are defined as anything related to a given **User Profile** and its associated metadata (**Online Presence**, **Access Permissions**, etc).

Friends

Interface definition for the online services **Friends** services. Friends services are anything related to the Maintenance of Friends and Friends Lists.

Sessions

Interface definition for the online services Session services.

Session services are defined as anything related managing a Session and its state.

Shared Cloud

Provides the Interface for **sharing files** already on the cloud (see User Cloud with other users).

User Cloud

Provides the Interface for per User Cloud file storage.

Leaderboards

Provides the Interface for accessing online Leaderboards.

Voice

Provides the Interface for **Voice Communication** over network while in the game.

Achievements

Provides the Interface for **Reading/Writing/Unlocking** Achievements.

External UI

Provides the Interface for accessing a given platform's **external Interfaces** if available.

Sessions and Matchmaking

Matchmaking is the process of matching Players with Sessions. A Session is basically an Instance of the Game running on the Server with a given set of properties, which is either **advertised**, so that it can be found and joined by Players wanting to play the Game, or **private**, so only Players who are **invited** or notified of it in some way can join.

Picture an **Online Game Lobby** that lists all of the Games currently being played.

Each Game in the list is a Session, or **individual** online match. Players are matched with Sessions either by searching or some other means and then join the Session to play the Match.

Basic Life-Time of a Session

- **Create** a new Session with desired Settings
- **Wait** for Players to request to join the Match
- **Register** Players who want to join
- **Start** the Session
- **Play** the Match
- **End** the Session
- **Un-register** the Players. Either:
 - **Update** the Session if you want to change the Type of Match and go back to **waiting** for Players to join
 - **Destroy** the Session

Session Interface

The **Session Interface**, **ISession**, provides platform-specific functionality for setting up the pieces behind the scenes that are necessary in order to perform Matchmaking as well as other methods of allowing Players to find and join online games. This includes **Session Management**, finding Sessions through search or other means, as well as joining and leaving those Sessions. The Session Interface is created and owned by the Online Subsystem. This means it only exists on the Server.

Only one Session Interface will ever exist at a time - the Session Interface for the platform the engine is currently running on. While the Session Interface performs all of the Session handling, the Game doesn't generally interact directly with it. Instead, the GameSession, **AGameSession**, acts as a game-specific wrapper around the Session Interface and the game code makes calls to it when it needs to interact with the Session.

The GameSession is created and owned by the GameMode, and also only exists on the Server when running an online game.

Each game can potentially have multiple GameSession types, but only one will ever be used at a time.

The most common case for a game having more than one type of GameSession is to add a type of GameSession for when the game is using a Dedicated Server.

Session Settings

The **SessionSettings**, defined by the '**FOnlineSessionSettingsclass**', are a set of properties that determine the characteristics of the session.

In the base implementation, these are things like:

- Number of Players allowed
- Is the Session advertised or private
- Is the Session a LAN match
- Is the Server dedicated or Player-hosted
- Are invites allowed
- Etc.

Using the Online Game Lobby Example, each of those games is a Session and has its own SessionSettings. For example, some Sessions may be **Player versus Player** (PvP) while others are **Cooperative Multiplayer** (Co-Op).

Different Sessions could be playing different maps or playlists, require different numbers of players, etc.

Session Management

(click header to get to the detailed tutorial for Sessions in C++)

All Session Nodes are **Asynchronous** tasks (Clock Symbol) and will call “**OnSuccess**” or “**OnFailure**” once they're finished. In the meanwhile, you can use the top most exec output.

Creating Sessions

In order for Players to find a Session and potentially join it, you need to create a Session and set up its properties, as well as decide which of those properties will be visible. Registering a Dedicated Server Session via Blueprint might not be supported at the current Engine Version. In C++ you should do the normal C++ Session creation inside the AGameSession by overriding the '**RegisterServer**' function!

Create a Session via Blueprint



For creating a Session in Blueprints, you can use the '**CreateSession**' Node that Epic already exposed. It doesn't offer many options, but this can be extended via **Plugins** from the **Forum**.

Create a Session via C++

[My Wiki-Post about Using Sessions via C++!](#)

Updating Sessions

Updating is done when you want to change the Settings of an existing Session and is performed using the **'IOnlineSession::UpdateSession()'** function.

For example, the Session may be set up currently to only allow **8 Players**, while it needs to allow **12 Players** for the next match. To update the Session, **'UpdateSession()'** would be called passing it **new** Session Settings that specify a **12 Player** maximum.

When the **request to update** a Session has completed, the **'OnUpdateSessionComplete'** delegate is fired. This provides the opportunity to perform any configuration or initialization necessary to handle the Session Settings changing.

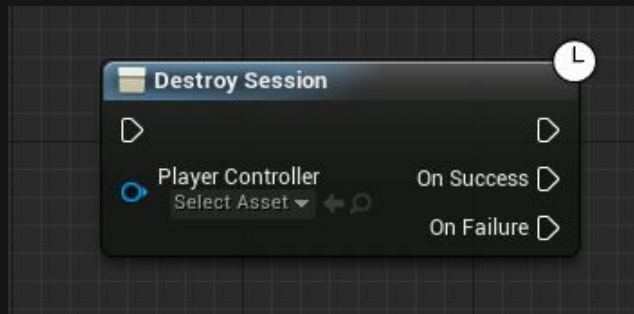
Updating a Session is generally performed **between Matches** on the Server, but it is also performed on the Client in order to keep the Session information in sync.

There is **no Blueprint Version** of this function yet, but it can be extended via **Plugins** from the **Forum**.

Destroying Sessions

When a Session ends and is no longer needed, the Session is destroyed using the '**IOOnlineSession::DestroySession()**' function. When the destruction operation has completed, the '**OnDestroySessionComplete**' Delegate is fired, enabling you to perform cleanup operations.

Destroy Session via Blueprint



For Blueprints, you can use the '**DestroySession**' Node, which Epic already exposed.

Destroy Session via C++

[My Wiki-Post about Using Sessions via C++!](#)

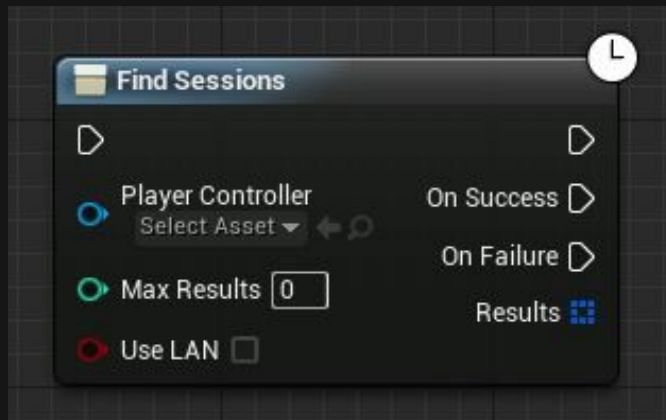
Searching Sessions

The simplest way to find Sessions is to search for Sessions that match some desired subset of Settings.

This could be in response to the Player choosing a collecting of filters in a User Interface, or it could be done automatically behind the scenes, based on the Player's skill and other factors, or it could be a combination of both methods.

The most basic form of Searching for Sessions is the classic Server Browser that shows all available games and allows the Player to filter those based on the type of Game they want to play.

Searching Session via Blueprint



For Blueprints you can use the '**FindSessions**' Node that Epic already exposed for you. You can specify the Number of results and if you want to search for LAN or Online Games.

More can be specified using **Plugins** from the **Forum**.

Searching Session via C++

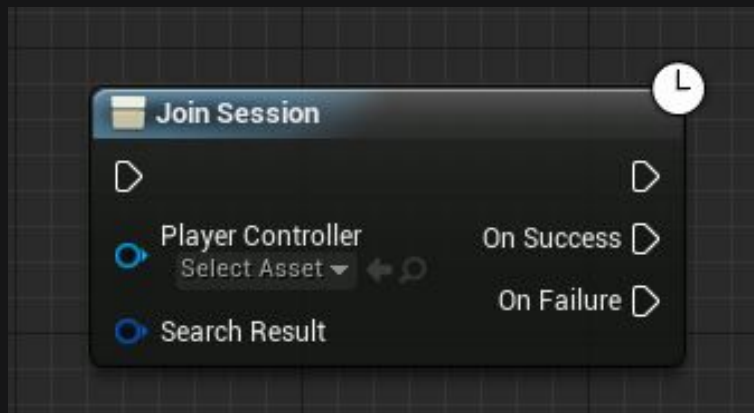
[My Wiki-Post about Using Sessions via C++!](#)

Joining Sessions

Once you have determined a Session for the Player to join, the process of joining is initiated by calling '**OnlineSession::JoinSession()**' and passing it the **Number of the Player** and the **Name** and **SearchResult** of the Session to join. When the joining process completes, the '**OnJoinSessionComplete**' Delegate is fired. This is where the logic to get the Player into the Match is performed.

More information about this logic can be read below and on my Session **C++ Blog-Post**.

Joining Session via Blueprint



For joining a Session via Blueprint, you can use the '**JoinSession**' Node, that Epic already exposed. It needs a SessionResult, which you can get from the '**FindSession**' Node.

It returns on Array of SessionResults. The '**JoinSession**' Node will directly join the Map '**OnSuccess**'.

You don't have to deal with that.

Join Session via C++

[My Wiki-Post about Using Sessions via C++!](#)

Cloud-Based Matchmaking

Cloud-Based Matchmaking refers to built-in **Matchmaking Services** that are available, and generally platform specific. An example of this type of service is the **TrueSkill System** available through the **Microsoft Xbox Live Service**.

To initiate Matchmaking on Platforms that support it, you call '**IOOnlineSession::Startmatchmaking()**' and pass it the Controller Number of the Player to matchmake for, the SessionName, SessionSettings to use if creating a new Session, and Settings to Search against. The '**OnMatchmakingComplete**' Delegate is fired when Matchmaking is complete. This provides a bool specifying whether the process was successful and the Name of the Session to join in that case.

A Matchmaking action in process can be canceled by calling '**IOOnlineSession::CancelMatchmaking()**' and passing it the Controller Number of the Player and the SessionName that was passed to the call to start Matchmaking in the first place.

The '**OnCancelMatchmakingCompleteddelegate**' is fired when the cancel operation is completed.

Following and Inviting Friends

On platforms that support the concept of Friends, Players can **follow Friends** into a Session or **invite Friends** to join them in a Session.

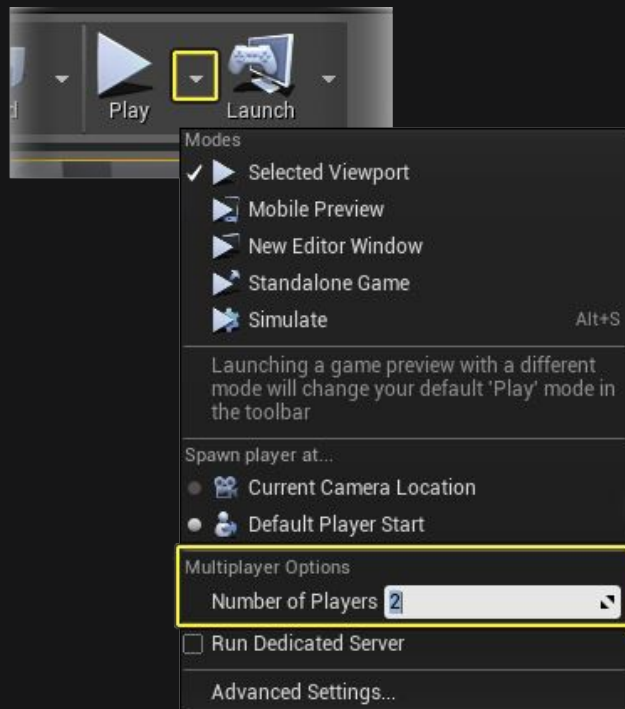
Following a Friend into a Session is done by calling '**ISession::FindFriendSession()**' and passing it the Number of the local Player that wants to join the Session and the ID of the Friend already in the Session. The '**OnFindFriendSessionComplete**' **Delegate** is fired when the Session is found and it contains a SearchResult that can be used to join the Session.

A Player can also invite one or more Friends to join them in their **current** Session using '**ISession::SendSessionInviteToFriend()**' or '**ISession::SendSessionInviteToFriends()**' and passing in the **Local Player Number**, **SessionName**, and **ID(s)** of the **Player(s)** to be invited.

When a Friend accepts an invitation, the '**OnSessionInviteAccepted**' **Delegate** containing the SearchResult of the Session to join is fired.

How to Start a Multiplayer Game

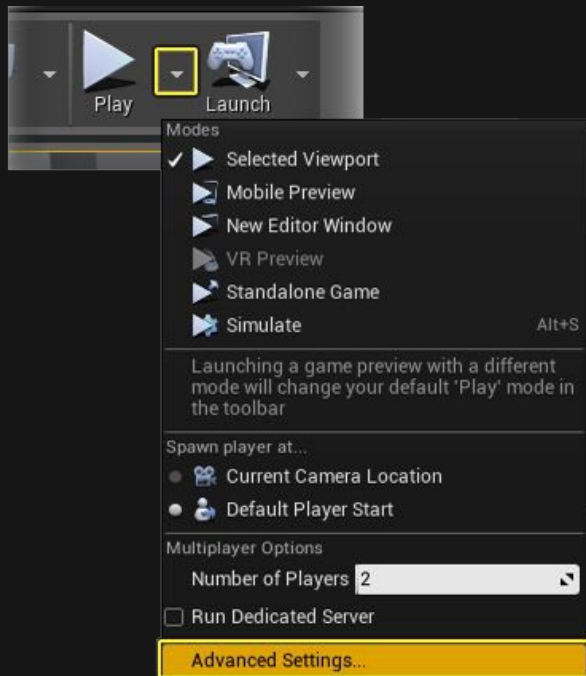
The easiest and straight forward way to start a Multiplayer Game is to set the **Number of Players**, in the Play Drop-Down Menu, to something higher than 1.



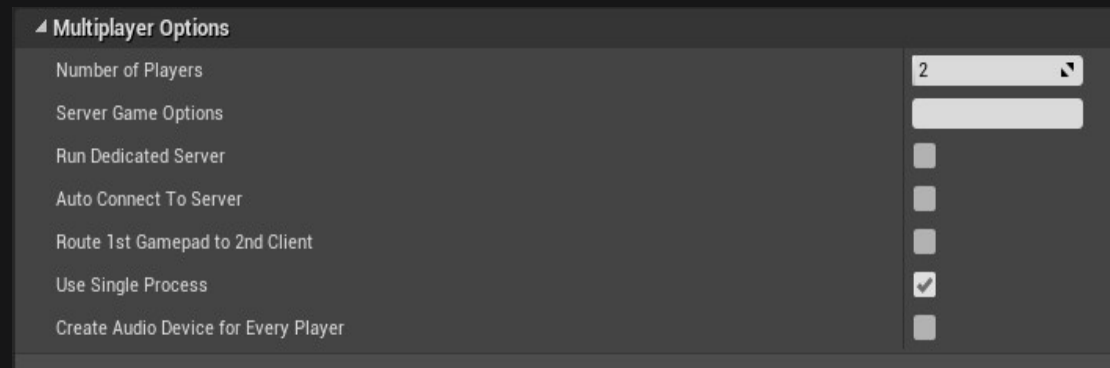
This will automatically create a network connection between Server and Clients. So even if you start the Game in your Main-Menu Level with **Number of Players** set to 2+, the Games will be connected!

This is always a network connection. It's **NOT** a local couch-coop Multiplayer connection. This needs to be handled differently and will **NOT** be covered at this point.

Advanced Settings



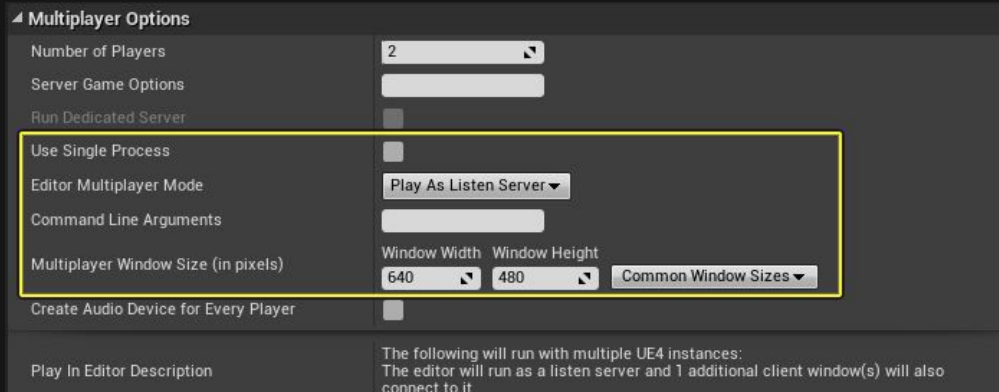
Advanced Settings allow you to specify some more options for the Play-Mode. One of the categories lets you set **Multiplayer Options**:



The following table will explain the **Multiplayer Options** one by one.

Option	Description
Number of Players	This option defines the number of players to spawn in the game when launched. The editor and listen server count as players, a dedicated server will not. Clients make up the remainder of players.
Server Game Options	Here you can specify additional options that will be passed to the server as URL parameters.
Run Dedicated Server	If checked, a separate dedicated server will be launched. Otherwise the first player will act as a listen server that all other players can connect to.
Auto Connect To Server	Controls, if the Game should directly connect the Clients to the Server. Means, when you just want to test Gameplay, you can check it and not care about setting up Menu for connecting. Other way round, if you have the Menu, you might want to deactivate the boolean.
Route 1 st Gamepad to 2 nd Client	When running multiple player windows in a single process, this option determines how the game pad input get routed. If unchecked (default) the 1st game pad is attached to the 1 st window, 2nd to the 2nd window, and so on. If it is checked, the 1st game pad goes to the 2 nd window. The 1st window can then be controlled by keyboard/mouse, which is convenient if two people are testing on the same computer.
Use Single Process	This spawns multiple player windows in a single instance of Unreal Engine 4. This will load much faster, but has potential to have more issues. When this is unchecked, additional options become available.
Create Audio Device for Every Player	Enabling this will allow rendering accurate audio from every player's perspective but will use more CPU.
Play in Editor Description	This is a description of what will occur when playing based on the currently applied Multiplayer settings.

Use Single Process

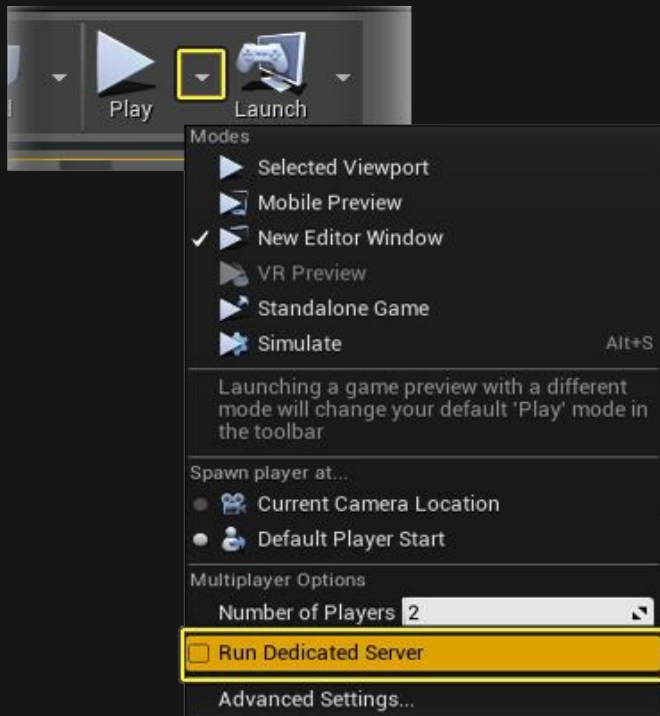


When **Use Single Process** is checked, multiple windows are spawned in a single instance of Unreal Engine 4.

When this option is unchecked, multiple instances of UE4 will be launched for each player that is assigned and additional options become available.

Option	Description
Editor Multiplayer Mode	This is the NetMode to use for Play In Editor (Play Offline , Play As Listen Server or Play As Client).
Command Line Arguments	Here you can assign additional command line options that will be passed to standalone game instances.
Multiplayer Window Size (in pixels)	Define the width/height to use when spawning additional standalone game instances.

Run as Dedicated Server



Without checking “**Run Dedicated Server**” the first Client will be a Listen-Server.

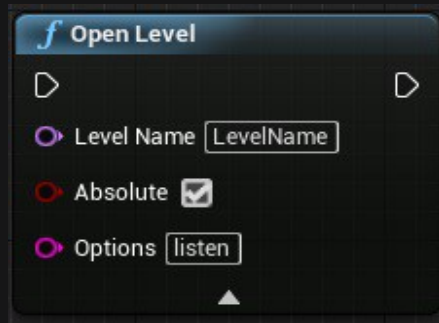
On the other hand, when marking it as **TRUE**, **all** Players will be Clients.

Start and Connect to a Server

The Subsystem and Session Chapter already explained how to setup a Session/Server via the Session System, but what about normal starting and connecting via IP? Pretty easy!

Blueprint

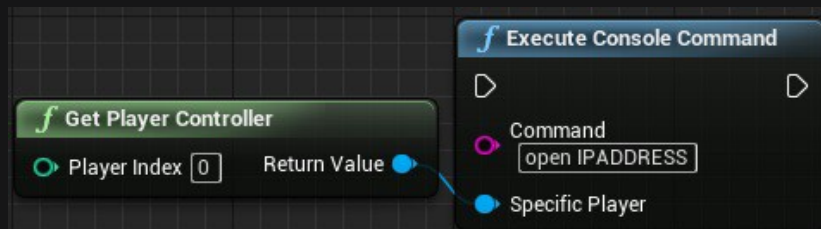
Start a Server



To start the Server, without the Session System, you simply use the '**OpenLevel**' Node and pass it the '**Level Name**' and the '**listen**' Option. You can also pass more Options, separated by a '?', which can be retrieved in the GameMode Class as explained.

A Dedicated Server, without Session System, already starts on the correct map, which you can specify in the '**Maps&Nodes**' Tab of your Project Settings.

Connect to a Server



To connect to a Server, you simply use the 'Execute Console Command' Node with the Command '**open IPADDRESS**', where '**IPADDRESS**' is replaced by the actual **IPADDRESS** of the Server.

This can be filled in through a Widget Text-Box, for example.

UE4++

Similar to Blueprints, you can use these two functions, that have the same result as the Blueprint Nodes.

Start a Server

```
UGameplayStatics::OpenLevel(GetWorld(), "LevelName", true, "listen");
```

Connect to a Server

```
// Assuming you are not already in the PlayerController (if you are, just call ClientTravel directly)  
APlayerController* PlayerController = UGameplayStatics::GetPlayerController(GetWorld(), 0);  
PlayerController->ClientTravel("IPADDRESS", ETravelType::TRAVEL_Absolute);
```

Starting via Command Line

Basic command lines (these commands use the editor and therefore don't require cooked data):

Type	Command
Listen Server	UE4Editor.exe ProjectName MapName?Listen -game
Dedicated Server	UE4Editor.exe ProjectName MapName -server -game -log
Client	UE4Editor.exe ProjectName ServerIP -game

Note: Dedicated Servers are headless by default. If you don't use “-log”, you won't see any window to present the Dedicated Server!

Connection Process

When a new Client connects for the first time, a few things happen:

First, the Client will send a request to the Server to connect.

The Server will process this request, and if the Server doesn't deny the connection, will send a response back to the Client, with proper information to proceed.

The following page will show the major steps of the connection process. This is a direct extract from the official Documentation.

The major steps are

1. Client sends a connect request.
2. If the Server accepts, it will send the current map
3. The Server will wait for the Client to load this map
4. Once loaded, the Server will then locally call **"AGameMode::PreLogin"**
 - This will give the GameMode a chance to reject the connection
5. If accepted, the Server will then call **"AGameMode::Login"**
 - The role of this function is to create a PlayerController, which will then be replicated to the newly connected Client. Once received, this PlayerController will replace the clients temporary PlayerController that was used as a placeholder during the connection process Note that **"APlayerController::BeginPlay"** will be called here.
It should be noted that it is **NOT** yet safe to call RPC functions on this actor. You should wait until **"AGameMode::PostLogin"** is called.
6. Assuming everything went well, **"AGameMode::PostLogin"** is called.
 - At this point, it is safe for the Server to start calling **RPC** functions on this PlayerController.

That's it so far.

This might extend in the near future, but right now I feel like I covered a lot of the basic and not so basic things for Multiplayer.

Hopefully this helped you to get started with Multiplayer and Networking in Unreal Engine 4 and we will see some cool new Multiplayer Games in the future!

Sources:

*1: Despite my own knowledge, I used the official Documentation, as stated at the beginning of the Compendium.

*2: The main network framework, as well as the two (recreated by myself) images of the distribution of common classes is documented by 'Nuno Afonso' and can be seen here: www.nafonso.com Thanks for this amazing page!

*3: Nearly all examples, Blueprint Pictures and C++ Code, are created by myself.