Unreal Engine 5 - Lesson 10 - Networking

—

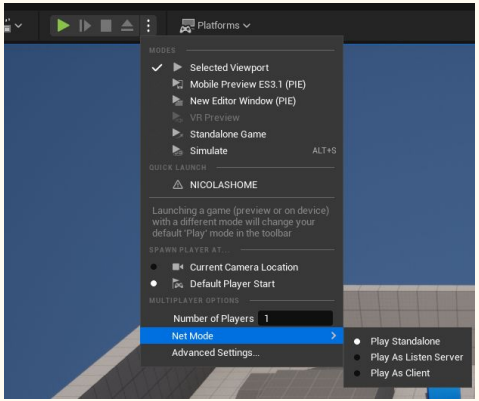
Nicolas Serf - Gameplay Programmer - Wolcen Studio
serf.nicolas@gmail.com

Summary

- Networking through Unreal Engine
- Plan it early
- Client-Server Model
  - Server Type
- NetMode
- Replication System
  - Actor
  - Ownership
  - Configuration
  - Relevancy
  - Property & Priority
- RPC
  - Reliable
  - WithValidation
  - Requirements & Caveats
  - General thoughts
- Property Replication
- Authority & ROle
  - Locally controlled
- Travelling in Multiplayer
- Final words

Networking through Unreal Engine

- **Built-in**
  - One of the **compelling** feature of the engine
  - Networking is **fully integrating** as a base in the engine
  - It offers easy **way** to **test** produce in **networking environment**
- **Integrated in Game Framework**
  - Obviously if you plan to make a multiplayer game in Unreal
  - But the Game Framework environment is **built** to take into consideration that **networking** is a pillar of many games
  - It allows to run **multiple** instance of an **editor** game window, each one handling input in it own windows
  - Serialization, opening sockets, sendings packet, connection... etc... This is all features that are integrated directly into the engine game code.
- **Server -client model**
  - Unreal's network model is based on an **authoritative server** on which **clients** connect
  - It **maintains** the **authoritative** state of the world
    - **When something changes, it change on the server**
    - The server **replicated** this changes into client

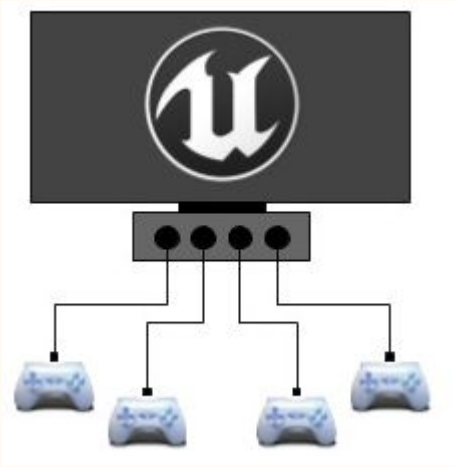


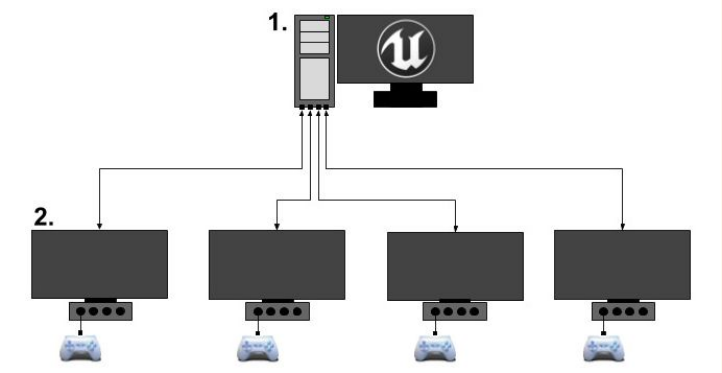
Plan for multiplayer early

- If there is a possibility that your project might need **multiplayer features** at any time, you should **build** all of your gameplay with multiplayer in **mind** from the start of the project.
- If your team consistently implements the **extra steps** for creating multiplayer, the process of building gameplay will not be much more **time-consuming** compared with a single-player game. In the long run, your project will be **easier** for your team as a whole to **debug** and **service**. Meanwhile, any gameplay programmed for **multiplayer in Unreal Engine will still work in single-player**.
- However, refactoring a codebase that you have **already** built **without** networking will require you to comb through your entire project and **reprogram** nearly **every** gameplay function. Team members will need to **re-learn** programming best practices that they may have taken for granted up to that point. You also will not be prepared for the **technical bottlenecks** that will be introduced by **network speed** and **stability**.
- Introducing networking **late** in a project is **resource-intensive** and **cumbersome** compared with planning for it from the outset.

Client-Server Model

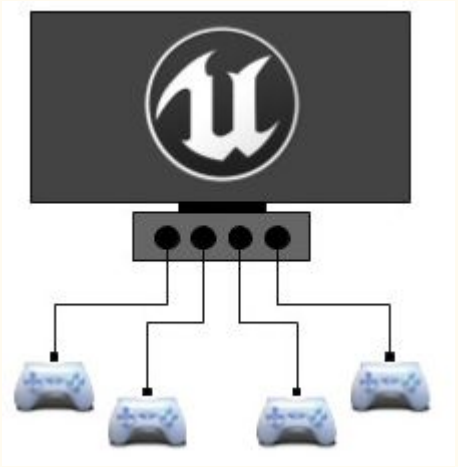
- In a **single-player** or **local multiplayer game**, your game is run **locally** on a **standalone game**. Players connect **input** to a single computer and control everything on it directly, and everything in the game, including the Actors, the world, and the user interface for each player, exists on that **local machine**.
- In a network multiplayer game, Unreal Engine uses a **client-server** model. One **computer** in the network acts as a **server** and **hosts** a session of a multiplayer game, while all of the other **players'** computers connect to the server as **clients**. The server then shares **game state information** with each connected client and provides a means for them to communicate with each other.
- The server, as the **host of the game**, holds the **one true**, **authoritative** game state. In other words, the server is where the **multiplayer** game is **actually happening**. The clients each **remote-control Pawns** that they own on the server, sending **procedure calls** to them in order to make them perform **ingame actions**. However, the server does not stream visuals directly to the clients' monitors. Instead, the server **replicates information** about the game state to each client, telling them what Actors **should exist**, how those Actors should **behave**, and what values different **variables** should have. Each client then uses this information to **simulate** a very **close approximation** of what is happening on the server.





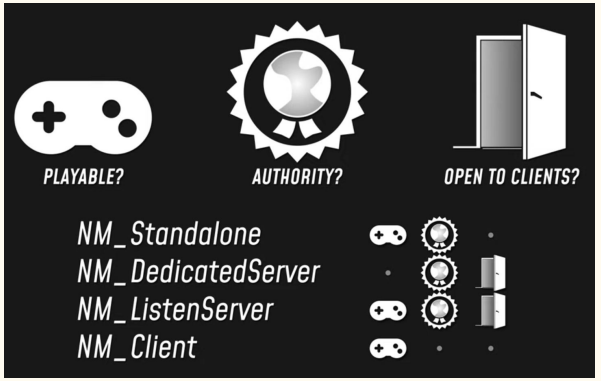
Server Types

- Network mode **describes** a computer's **relationship** to a network **multiplayer** session. An instance of a game can take on any of the following network **modes**
  - **Standalone** :
    - The **game** is **running** as a **server** that does **not accept** connections from remote clients. Any players participating in the game are strictly **local players**. This mode is used for **single-player** and **local multiplayer** games. It will run both server-side logic and client-side logic as appropriate for the local players.
  - **Client**
    - The game is **running** as a **client** that is **connected** to a **server** in a network multiplayer session. It will **not run** any **server**-side logic.
  - **Listen Client** :
    - The game is running as a **server hosting a network** multiplayer session. It **accepts connections** from remote clients and has local players directly on the server. This mode is often used for **casual cooperative** and **competitive multiplayer**.
  - **Dedicated Server** :
    - The game is running as a **server hosting** a network multiplayer session. It accepts **connections** from **remote clients**, but has **no local players**, so it discards graphics, sound, input, and other other player-oriented features in order to run more efficiently. This mode is **often used for games requiring more persistent**, **secure**, or **large-scale** multiplayer.



Net Mode

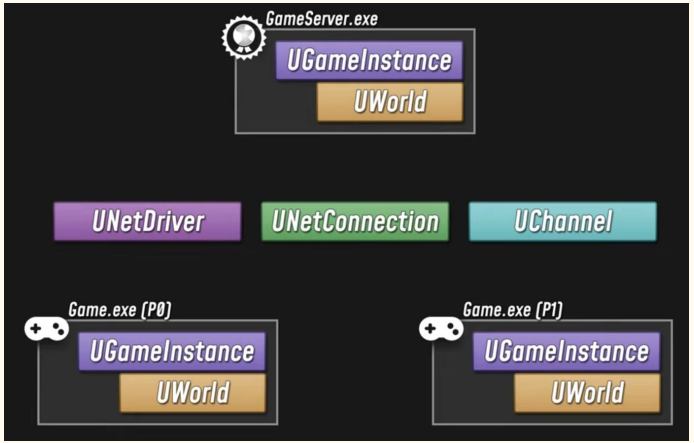
- It is a **property** of the **UWorld** which can be
  - **NM_Standalone**
  - **NM_DedicatedServer**
  - **NM_ListenServer**
  - **NM_Client**
- There is 3 factors to take into consideration to know how different this NetMode are
  - **Playable** ? Does our **GameInstance** has a **LocalPlayer** and **processing** inputs and **rendering** into viewport ?
  - **Authority** ? Does our **GameInstance** has the **authoritative copy** of the world with **Game Mode** in it ?
  - **Open to clients** ? If it is a **server**, **is it open** for **remote connection** attempts ?
- Remember that **each** executable has a **UGameInstance** that browse to an **URL** (local or remote) to build a **UWorld**

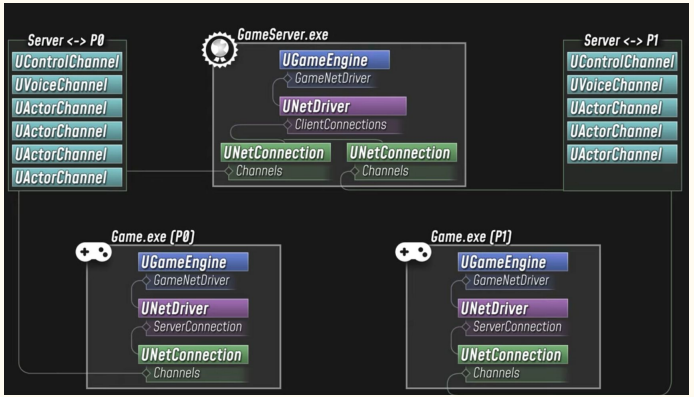


Replication system

- When we are in a **multiplayer scenario**, **replication system** is one of the **core system** to make online works.
- **Each clients** and the **server** all have their **UGameInstance** with their **UWorld** create.
    - **Replication** is responsible to **ensure** that the **different instance** of the game are **in-sync**
    - Each **UWorld** build their own picture of their **shared world** and **replication** is here to make sure they all **have** the **same informations**

- **Replication** system relies on 3 important classes
    - **UNetDriver** : It is created by UGameEngine when server or client boot up.
        - On server, it then listen for messages from remote processes
        - On client : It sends a connection request to the server
    - **UNetConnection** : When **UNetDriver** from server and client have made contact, it establishes a **UNetConnection** inside **UNetDriver**
        - **Server** has 1 **UNetConnection** per client
        - Client has only a single **UNetConnection**
    - **UChannel** : Within every **UNetConnection**, there is a number of different **UChannel**.
        - **ControlChannel** : Exchanges connection control messages
        - **VoiceChannel** :
        - **UActorChannel** : 1 for each actor being replicated across the connection

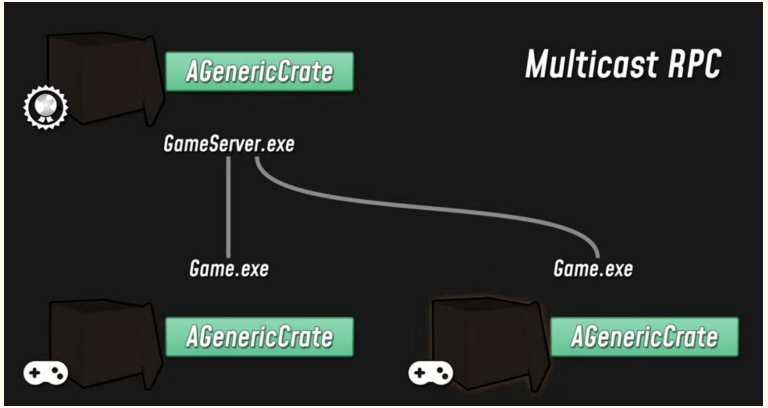




Replication system - Actor

- As state on the slide before, **replication** happens at **actor level**
- If you need an **actor** to stay in **sync** over network, you configure that actor to be **eligible** for **replication.**
    - When we do so, the **server** will **open** an **ActorChannel** in that player **NetConnection**.

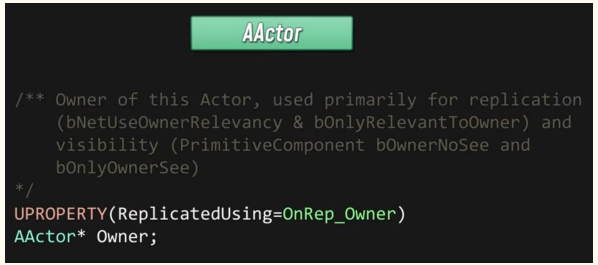
- What is happening to an actor that is being replicated to a client
    - **Lifetime** of the actor is kept in sync, meaning the **creation** and **destruction** of the actor is handle by that
    - **Property replication** is the second one. When an actor has properties marks to be replicated, it will be automatically propagated from server to clients
    - Finally, **RPC** which stands for **Remote Procedure calls.**
        - When a function is designate as Multicast and the function is called on the server, the server will send a message on every client to whom that Actor is currently replicated, in order to ensure that client will also call this function on it copy.
        - There is also **Client RPC** and **Server RPC**

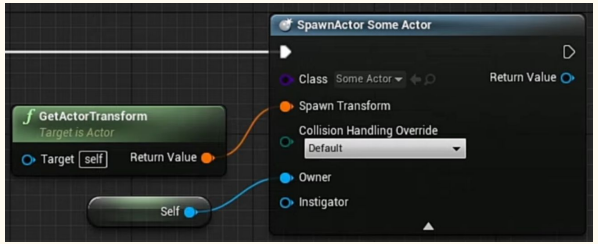


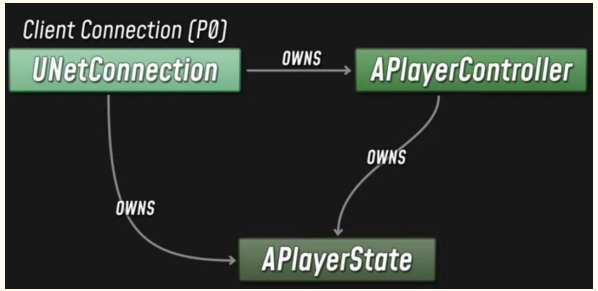
Replication system - Ownership

- Another important **concept** in replication **Ownership**
- Each actor can have **another Actor** designated as its **Owner**
  - You basically set the **owner** on **spawn**
  - You can also call **SetOwner**() at **runtime**

- **APlayerController** class has a special importance related to Ownership
  - Each **UNetConnection** represent a **Player**
  - Once the player is **fully logged** into the game, a **Player Controller** is **associated** with it
    - From the server POV, the **UNetConnection** owns that **PlayerController**
    - By extension, the **UNetConnection** owns every **AActor** that can **trace** their **ownership** back to that **PlayerController**
  - **APlayerController automatically** own the APawn they are **possessing**. So from there, we can create a tree of ownership that goes from **APlayerController** into every Actor related to the **APawn.**







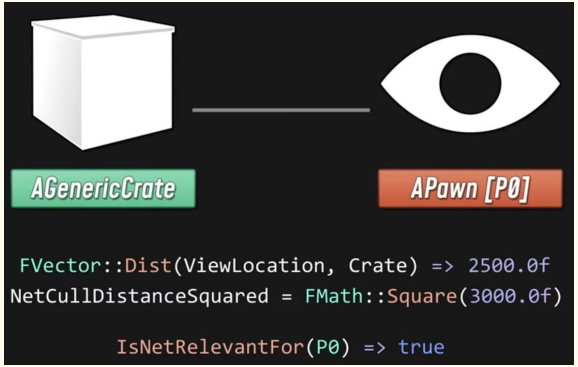
Replication system - Configuration

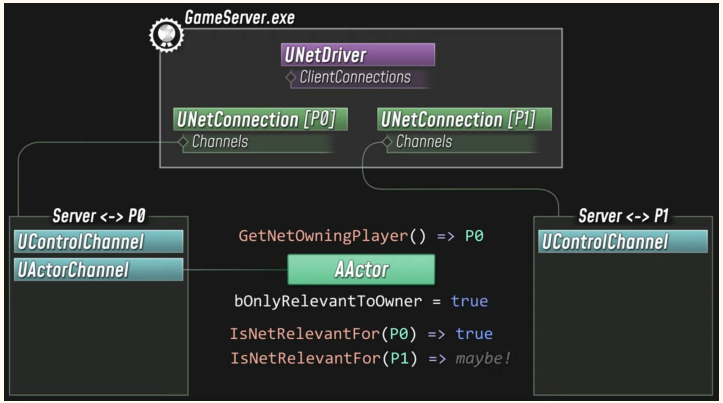
- For an Actor to be **consider** into replication, it needs to have its **bReplicated flag** sets to **true**
  - It is basically set in the **constructor**, or in the blueprint **details panel**
  - It can be **turn on** and **off** at **runtime** through **SetReplicated()**
  - **bReplicated** to true **DOESN'T** means it will be **replicated**, it means that the Actor is **eligible** for it
  - It means that the server can open a **UActorChannel** in the **UNetConnection** for replication

```
/** If true, this actor will replicate to remote machines.
    @see SetReplicates()
*/
UPROPERTY(EditDefaultsOnly, BlueprintReadOnly, Category=Replication)
uint8 bReplicates:1;
```

Replication system - Relevancy

- Actor's **relevancy** determine which connection that will happen for at what times
  - When actor is **eligible** for replication, from time to time, server's **net driver check** that actor against **each client** connection to determine if it's **relevant** for that client : **IsNetRelevantFor()**
  - Some AActor are **always relevant**, through the property **bAlwaysRelevant**
    - It means that while **eligible**, the server will **replicate** that actor to **all client** at **all times**. For example the **GameState** and **PlayerState** are always relevant
- Ownership is important on relevancy
  - An Actor that is **owned** (instigated) by a particular **player** will **always** be **considered** relevant for the corresponding client
  - Some Actors, like **PlayerControllers** are **configured** to **only** be **relevant** to their **owner**.
    - It means that they'll **never** replicate to **non-owning clients**
  - We can **configure** an actor to **inherits** relevancy from its **Owner**
- If **none** of the **special flags** are **set** and the client **doesn't own the actor**, the **default** behavior is applied
  - If Actor is **hidden** and its root component has **collision disable** => **Not relevant**
  - **Otherwise**, **relevancy** is based on **distance** from the player that **correspond** to the **client connection**.
    - If Squared distance to the player is less than **NetCullDistanceSquared**, then the actor is relevant to that player



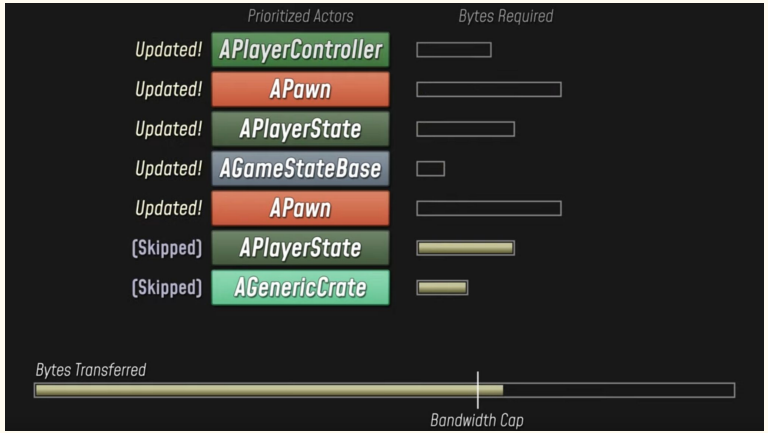


Replication system - Frequency & Priority

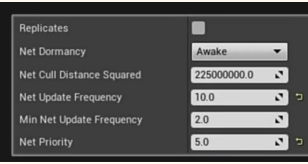
- When an actor is **replicated**, **frequency and priority** will determine how **often** the server
  - **Sends** update to client from whom that actor is relevant
  - Setting **NetUpdateFrequency** will tell the **system** how many **times** per **second** the server will check an actor an **potentially** send new **data** to **clients**

- Real world networks have **extremely variable latency** and **bandwidth** can become a **limiting factor**
  - It means that even by **sending update 60 times per seconds,** you can still **not see** perfectly **smooth** results on the other end

- Server's **UNetDriver** uses some simple **load balancing** in order to **mitigate** bandwidth **saturation**
  - At any point, **UNetDriver** has a **finite amount** of bandwidth to work with
  - UNetDriver **sort relevant** actors according to **priority** and then it **runs** network update until its used up it available **bandwidth**
  - Actors **closer** to the player has higher priority
  - Actors **not updated in a while** has higher priority
  - This **weight** is set in engine code, but we can also **set net priority** on an actor which will act as a final weight **multiplier** to that computation



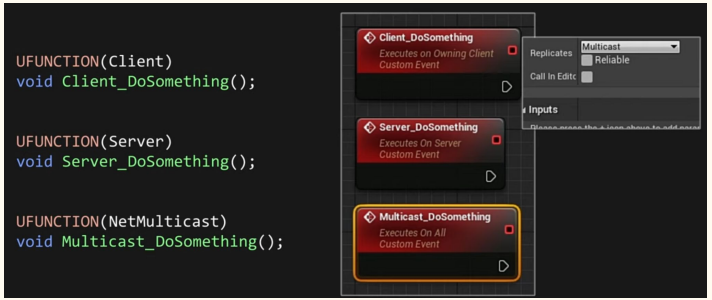

```
APulsatingSphere::APulsatingSphere()
    : Super()
{
    NetPriority = 5.0f;
}
```

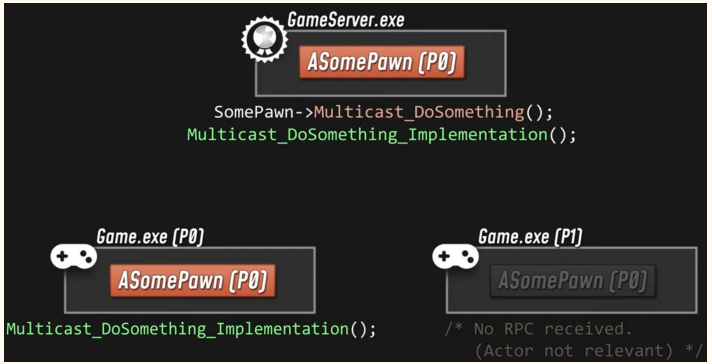


RPC

- Any **UFunction** can be designated as an RPC through
  - **Client**
    - Invoking a **client RPC** from the **server** will make the implementation **run** on the **owning client**
  - **Server**
    - Invoking a **server RPC** from the **owning client** will make the implementation **run** on the **server**
  - **Multicast**
    - Invoking a **multicast RPC** from the **server** will make the implementation **run everywhere**
      - **On the server**
      - **Then** on **all clients**

- **Relevancy** is a **factor** for **multicast** since **non-owning client** may **not** have an **open** channel for the actor
  - In that case, the **client will not receive the RPC**
  - It means you **shouldn't rely** on **multicast** RPC to **replicate persistent state** changes to **clients**

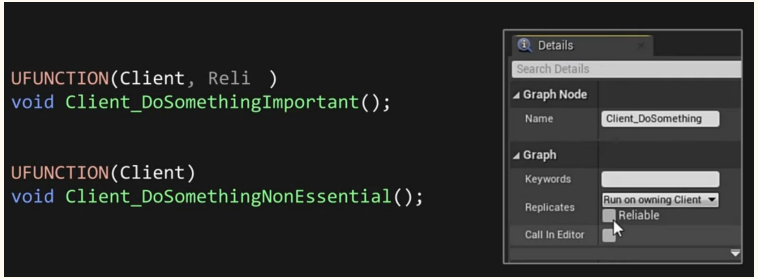




RPC - Reliable

- RPC can be declared
  - **Reliable**
    - **Guaranteed** to **arrive**
    - In a given Actor, they are **guaranteed** to **arrive** in the **order** in which they were **called**
    - **Necessary** if function call is **critical** to gameplay
    - Do **not overused** as it can **overload** the bandwidth, and leads to bottlenecks
  - **Unreliable**
    - They **can be dropped** if bandwidth is **saturated**
    - They are **not guaranteed** to **arrive**
    - They are **not guaranteed** to **arrive** in **order**

- In c++, the body of your **function** needs to be **defined** with **_Implementation suffix**, **even if the header declaration is not written** like this
  - Unreal engine **declare** that **automatically** through **reflection** and **auto generated source files.**
  - This is the function called on the **remote process** whereas the **function** called **locally** is an auto generated one




```
void ASomePawn::OnAttackReleased()
{
    float ChargeStrength = 0.0f;
    if (ChargeStrengthPerSecond > 0.0f && MaxChargeStrength > 0.0f && ChargeStartTime >= 0.0f)
    {
        const float ChargeDuration = GetWorld()->GetTimeSeconds() - ChargeStartTime;
        ChargeStrength = FMath::Min(MaxChargeStrength, ChargeDuration * ChargeStrengthPerSecond);
    }
    Server_InitiateAttack(ChargeStrength);
}

void ASomePawn::Server_InitiateAttack_Implementation(float ChargeStrength)
{
    UE_LOG(LogRepsiCore, Log, TEXT("Attack! (Charge: %0.2f)"), ChargeStrength);
}
```

RPC - WithValidation

- Another **specifier** for the function macro is **WithValidation**
- Like **_Implementation**, if you **declare** that, you'll **need** in the cpp files to create an defined a function with **_Validate suffix**
  - It returns a **boolean** which indicate if the values are **trustworthy**
  - It is a **cheat detection** in the case where the **server uses data sent from the client** in a way that **affects** gameplay

- **WARNING** : If a **server RPC fails validation**, the **client** who sent that RPC will be **kicked**

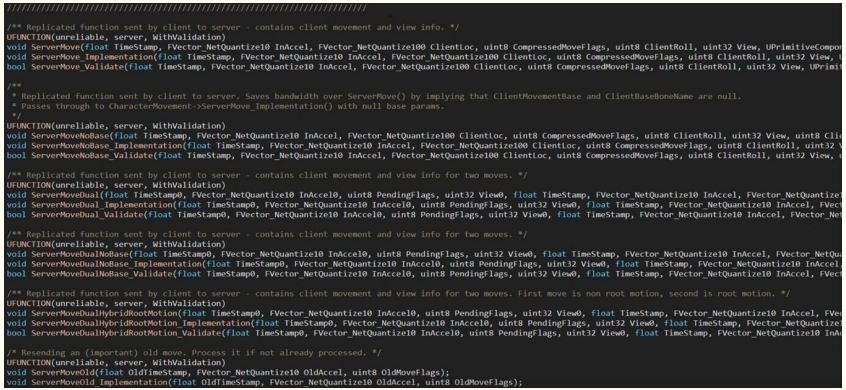
```
void ASomePawn::OnAttackReleased()
{
    float ChargeStrength = 0.0f;
    if (ChargeStrengthPerSecond > 0.0f && MaxChargeStrength > 0.0f && ChargeStartTime >= 0.0f)
    {
        const float ChargeDuration = GetWorld()->GetTimeSeconds() - ChargeStartTime;
        ChargeStrength = FMath::Min(MaxChargeStrength, ChargeDuration * ChargeStrengthPerSecond);
    }
    Server_InitiateAttack(ChargeStrength);
}

bool ASomePawn::Server_InitiateAttack_Validate(float ChargeStrength)
{
    return ChargeStrength <= MaxChargeStrength;
}

void ASomePawn::Server_InitiateAttack_Implementation(float ChargeStrength)
{
    UE_LOG(LogRepsiCore, Log, TEXT("Attack! (Charge: %0.2f)"), ChargeStrength);
}
```

RPC - Requirements and Caveats

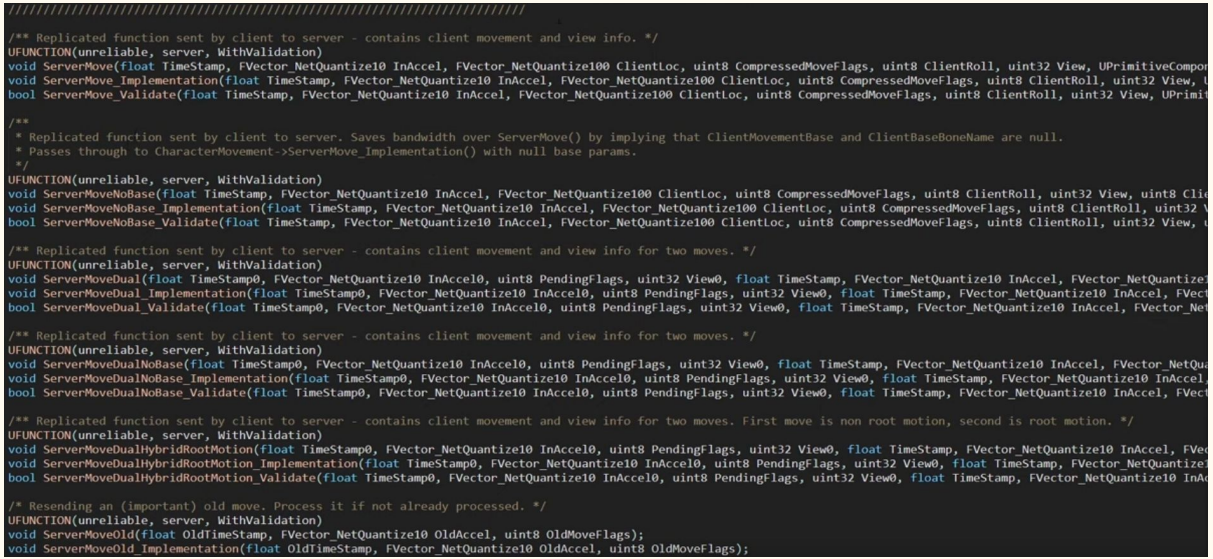
- There are a **few requirements** that need to be met for RPCs to be **completely functional**:
  - They must be called from **Actors**.
  - The Actor must be **replicated**.
  - If the RPC is being **called from server** to be executed on a **client**, only the client who actually **owns** that Actor will **execute** the **function**.
  - 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.
  - **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**.



RPC - General thoughts

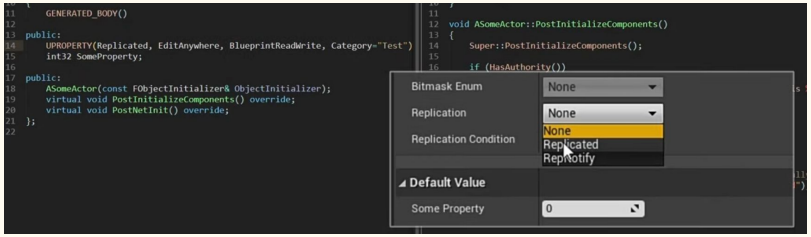
- RPC are the **only way to get data from** the **client** to the **server** by way of the owning connection
- RPC are typically **reserved** for **high-priority** and **time-critical** network code
    - For example, **CharacterMovementSystem** makes liberal use of RPC to send position update back and forth
        - Because **movement** is **critical** and **movement prediction** and **correction requires up-to-date informations** with as little latency as possible

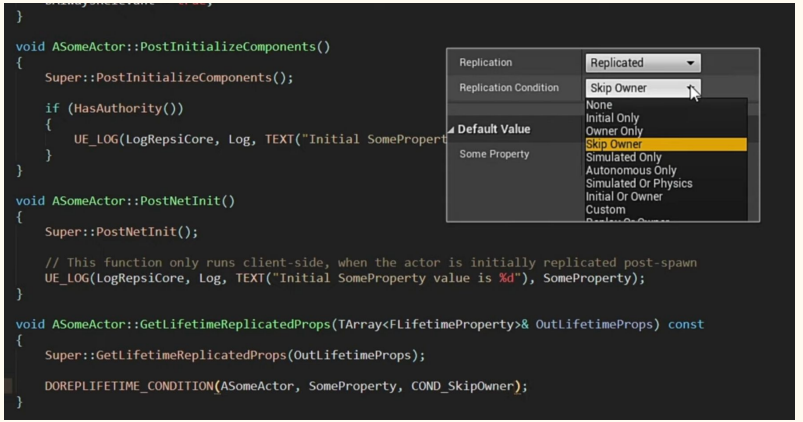
- In other cases, try to use as much as possible **Property Replication**



Property Replication

- It is the **standard** way of the **Unreal's replication** system
- **Load balancing** and **prioritization features** make it far more scalable
- **RPC** are **NOW** when **Property replication** are **Eventually**
- When you change a **replicated property** on the server, you can **count** on all **clients eventually being in sync** with the server
    - If at the point of change, the **client is not relevant** to receive the information (to far e.g), it **will not**... until it **becomes relevant** and get the update
- Property replication **respect** update **frequency** and **bandwidth limits**
- In order to replicate a property, we can through **Blueprint details panel** or with **Replicated** attribute on **UPROPERTY** macro
    - In addition to that in c++, you need to declare a **GetLifetimeReplicatedProperty** function in the cpp file.
        - In this function, you'll **specify** which **properties** should be replicated and under **what conditions**
        - **DOREPLIFETIME** is the macro to **replicate** to all clients at **all times**
        - But **we can add condition like replicated only to owning client**, **non-owning client**, etc...
- If you want to **run code** when replicated property is **update**, you can **declare** a **UFUNCTION** with prefix **OnRep_XXX**, **XXX** being the **property name**.
- Note : In c++, if you want to **RepNotify** logic to run on server, you need to manually call the **OnRep_XXX function**





Authority & Role

- An Actor can have few roles
  - **ROLE_None**
  - **ROLE_SimulatedProxy**
  - **ROLE_AutonomousProxy**
  - **ROLE_Authority**
- In most cases, you'll need to more generally think **"Do I have authority over this Actor ?"**
- In Actor class, you can call HasAuthority() function
  - If you have **Authority**, you have the **final say in updating** the state of an actor, either because
    - The game is in **single player mode** like **NM_Standalone**
    - The code is being **executed** by the **server**
    - The actor **only exists** on the **client**
  - If you **don't have Authority**, your code is **running on the client** but the actor is **being replicated from the server**
    - In that case, your **client's copy** of the actor is a **proxy** for the authoritative version on the server
    - It will almost always be a **SimulatedProxy**
    - **AutonomousProxy** enters when we are talking about players
      - **PlayerController** being **replicated** to the **owning client** is an **AutonomousProxy**, and the associated **Pawn** is also an **AutonomousProxy**
      - For other clients, this Pawn is a **SimulatedProxy**

```
enum ENetRole
{
    ROLE_None,              HasAuthority() => false
    ROLE_SimulatedProxy,    HasAuthority() => false
    ROLE_AutonomousProxy,   HasAuthority() => false
    ROLE_Authority          HasAuthority() => true
};
```

	GameServer.exe NM_DedicatedServer	Game.exe [P0] NM_Client	Game.exe [P1] NM_Client
AGameModeBase	AUTHORITY	(not replicated)	(not replicated)
APlayerController [P0]	AUTHORITY	AutonomousProxy	(not replicated)
APlayerController [P1]	AUTHORITY	(not replicated)	AutonomousProxy
APawn [P0]	AUTHORITY	AutonomousProxy	SimulatedProxy
APawn [P1]	AUTHORITY	SimulatedProxy	AutonomousProxy

Authority & Role - Locally controlled

- When dealing with **player related code** there is one more **question** to be **aware** of : **Is the player locally controlled ?**
  - When a Pawn is **locally controlled**, then the **player** corresponds to the **GameInstance** where your code is running
  - If not, it's a **remote client's player**

Travelling in Multiplayer

- In Unreal Engine (UE), there are two main ways to travel: **seamless** and **non-seamless**. The main difference, is that **seamless** travel is a **non-blocking operation**, while **non-seamless** will be a **blocking call**.
- When a client executes a **non-seamless travel**, the client will **disconnect** from the **server** and then **re-connect** to the **same server**, which will have **the new map ready to load**.
- It is recommended that **Unreal Engine multiplayer games** use **seamless** travel when possible. It will generally result in a **smoother experience**, and will **avoid** any issues that can occur during the **reconnection 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
- There are three main function that drive travelling:
    - UEngine::Browse
    - UWorld::ServerTravel
    - APlayerController::ClientTravel.

- **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 travelling 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)

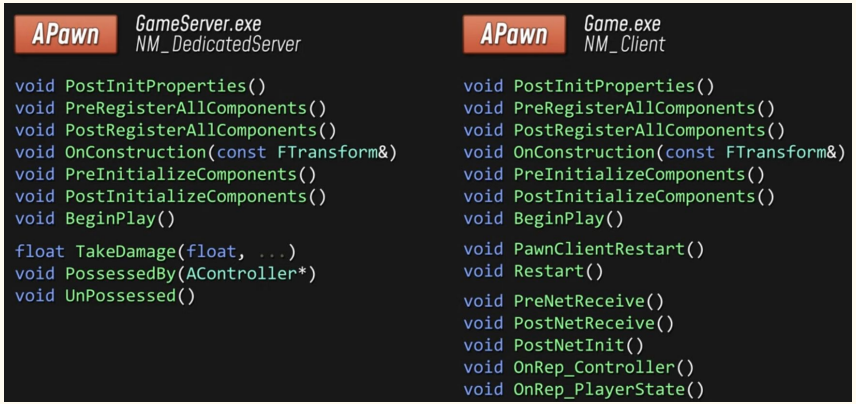
Travelling in Multiplayer - Seamless travel

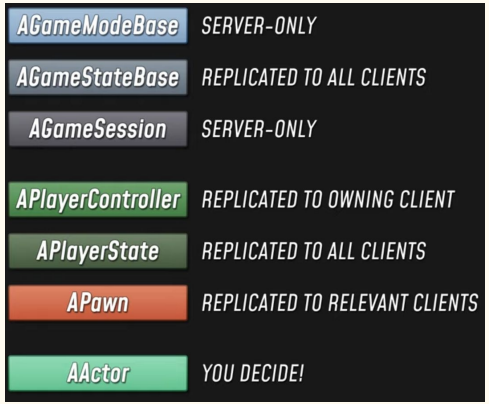
- **Enabling seamless travel**
  - To **enable seamless travel**, you need to setup a **transition map**. This is configured through the **UGameMapsSettings::TransitionMap** property. By default this property is **empty**, and if your game leaves this property empty, an **empty map will be created** for the transition map.
  - 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.
  - So now we can **travel** from the **current map** to the **transition map**, and then from there we can travel to the final map. Since the transition map is very small, it doesn't add much extra overhead while it overlaps **the current and final map**.
  - Once you have the transition map setup, you set **AGameModeBase::bUseSeamlessTravel to true**, and from there seamless travel should work
- **Seamless travel flow**
  - Mark actors that will persist to the transition level (more below)
  - Travel to the transition level
  - Mark actors that will persist to the final level (more below)
  - Travel to the final level
- **Persisting Actors across 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 AGameModeBase::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

Final words

- Dealing with **multiplayer** and replication is **not simple**
- It require an **extra step** in thinking about your **design** in order to make **maintainable and clean code**
- You need to think about how **datas** will be **transferred between clients** and **server**, how functions will be **executed**
- It may **feel hard to get** at first but will get **easier to manage with experience** and by **understanding** how the engine works

- A reason to use the **Game Frawework** is that it is designed for multiplayer **out of the box**.
    - It comes also to the **cost** that if you want to **overwrite** some function from the game framework, **you'll need to understand how they are integrated in network environment** to ensure that your code inside the function is logic





Time to.... highlight a concept

Technological survey

Practice

- **General**
  - We'll create a mini-project example for today practice
  - We'll skip the mini-project but feel free to add multiplayer on the stealth game & in your XCOM projects
  - Create a ping-pong
    - Listen server
    - Clients has an HUD with 1 button which is simply "Send"
    - Based on the last message received, when a client click on Send, it will either send Ping or Pong
    - When a client receive a Ping, the color of the mesh it is represented by will turn red
    - When a client receive a Pong, the color of the mesh it is represented by will turn yellow
    - When a ping/pong is received, show a message on the HUD that specify if it is a Ping or a Pong
    - When a ping is received, locally trigger a VFX from a niagara system
    - When a pong is received, replicated a VFX and create an actor in the area around the player