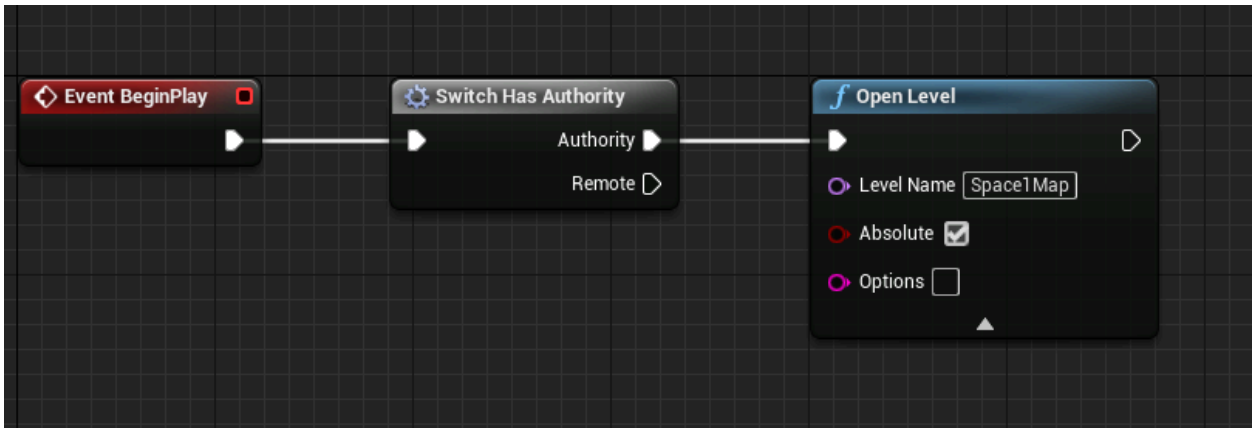
Intro To Unreal Multiplayer in Blueprints

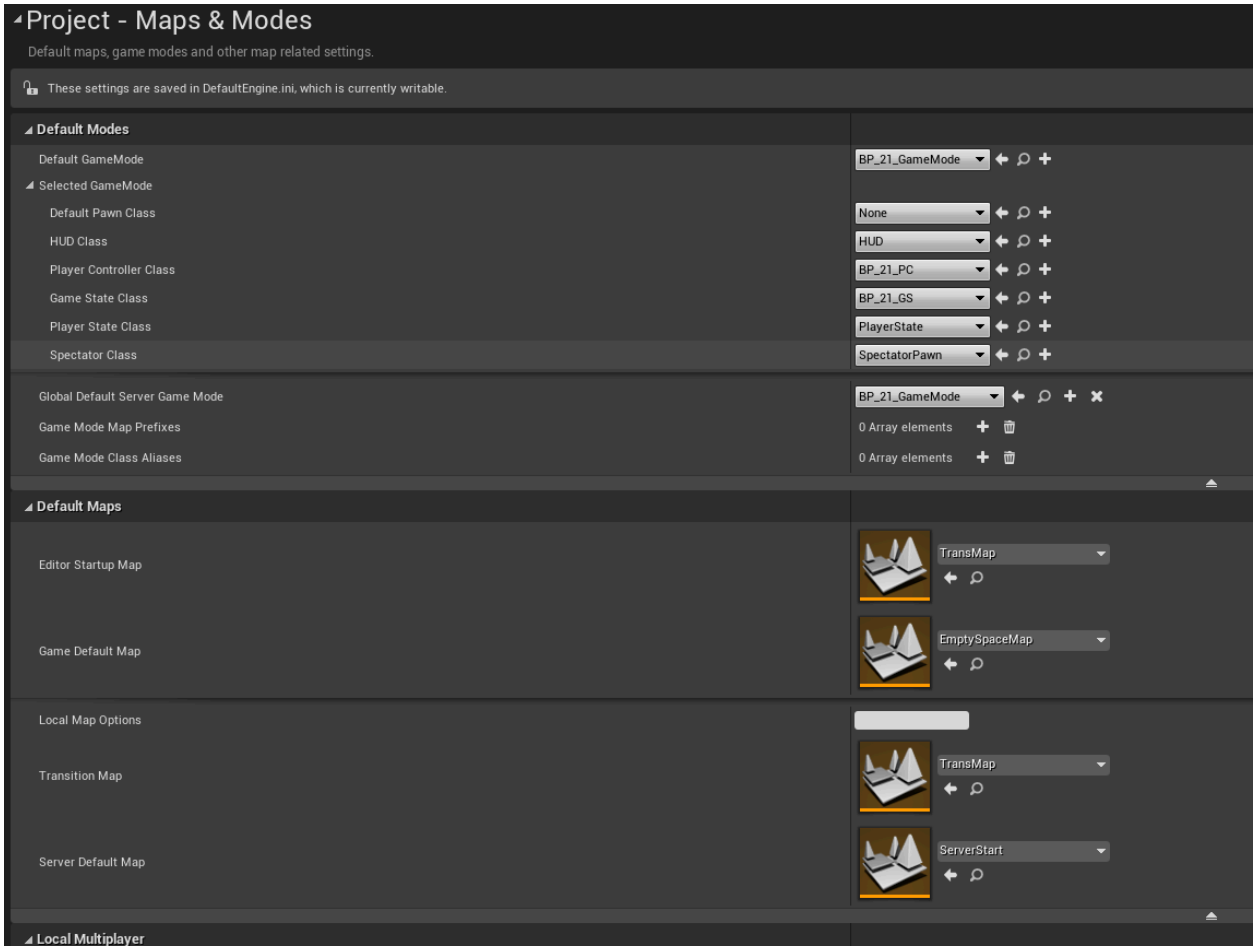
This assumes a dedicated server, but it will work with listen servers. This also ignores the “Sessions” subsystem. There will be no mention of prediction, verification, cheat-protection, or smoothing. It is a quick and dirty setup for learning and testing. Let me know on the discord if anything is incorrect or confusing: https://discord.gg/jHqWn3Yr

Setup

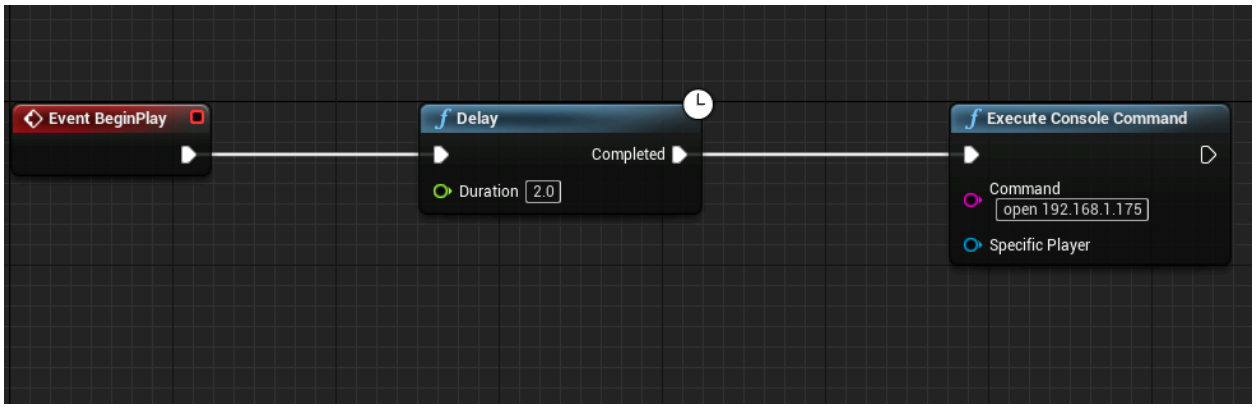
Create a new blank level. In the level blueprint, drag a line from EventBeginPlay, and add a SwitchHasAuthority. Drag a line from Authority and add an OpenLevel. Type the name of the level the server should load. This is probably where you will add “sessions” stuff in the future.



In “Project Settings,” click the “Maps & Modes” tab. Ensure the “Server Default” map is the one you created above.



Decide which map will be the default for the clients, and set that to “Game Default Map.” In the level blueprint, drag a line from EventBeginPlay and add an ExecuteConsoleCommand. Type “open [server-ip]” where [server-ip] is the ip address of the machine the server is running on. You may want to add a short delay.

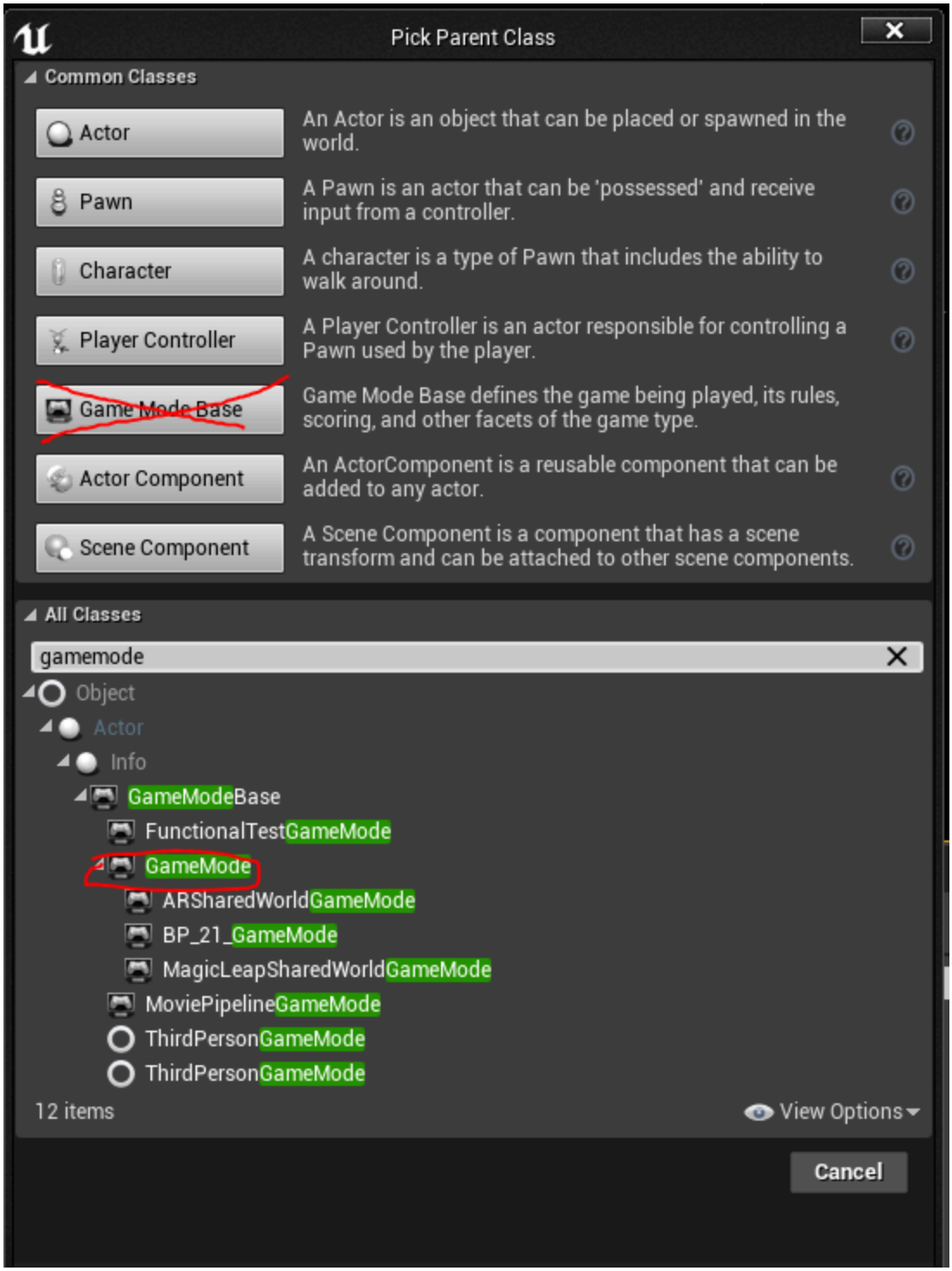


Gameplay Framework

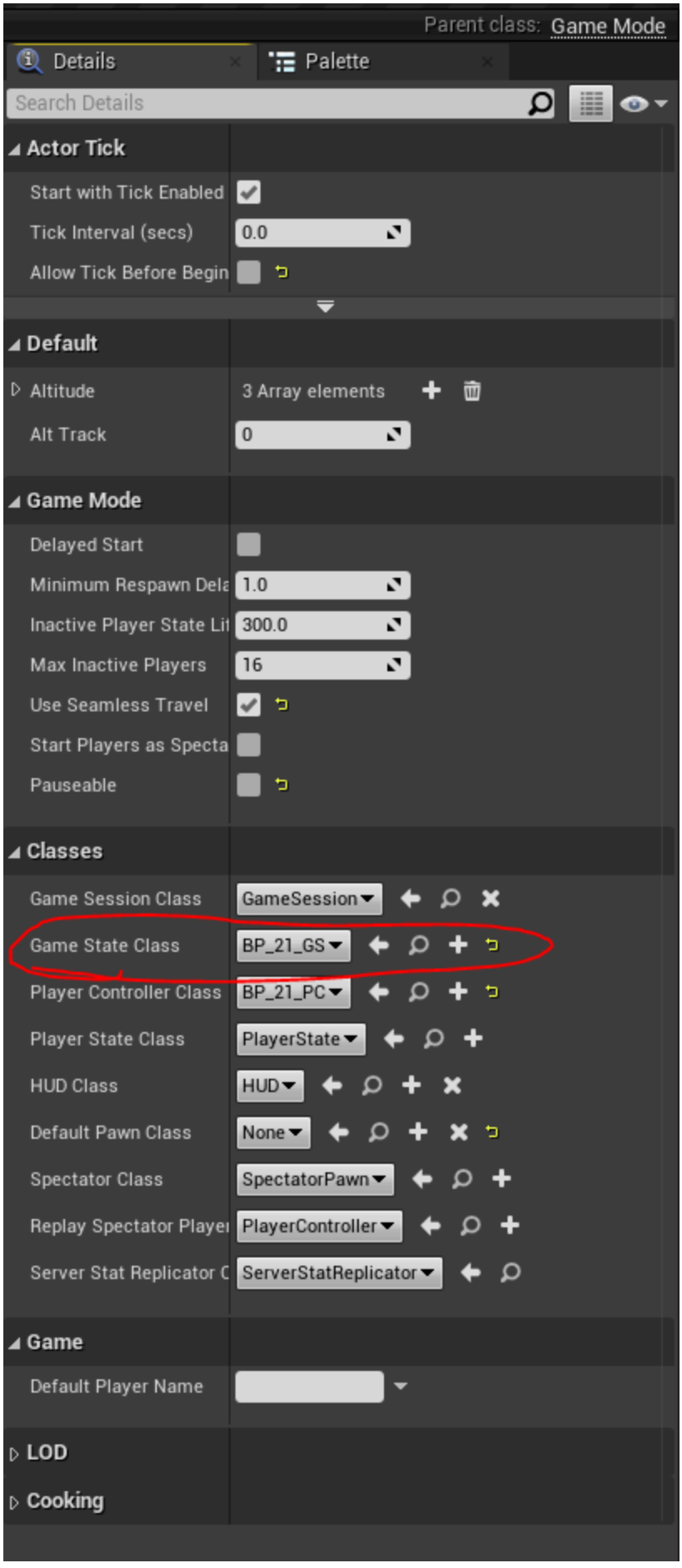
https://docs.unrealengine.com/en-US/InteractiveExperiences/Framework/QuickReference/index.html

Please review the reference at the url above.

The “Game Mode” is like the server’s brain. When we want the server to do stuff that doesn’t exist inside some other actor, we do it here. It is also the first stop a client’s player controller makes when connecting to the server. Make a fresh game mode from “GameMode,” and update “Maps & Modes” accordingly.

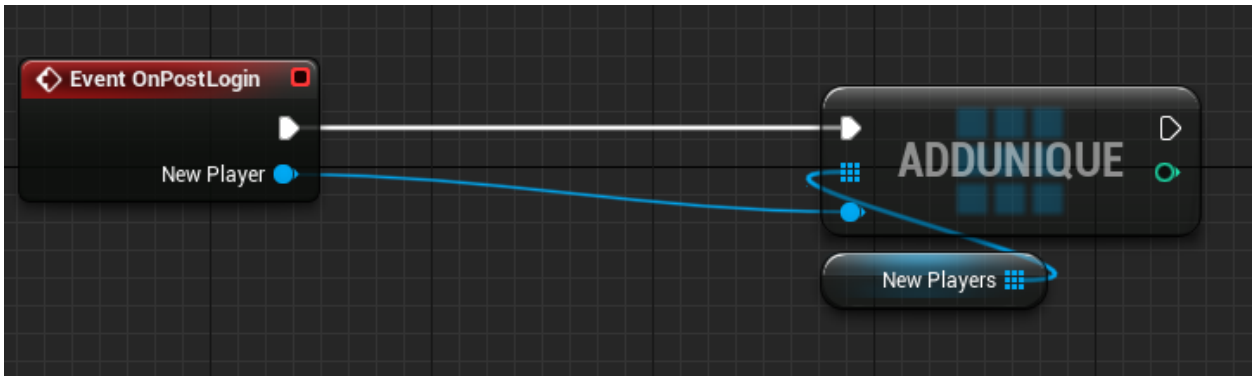


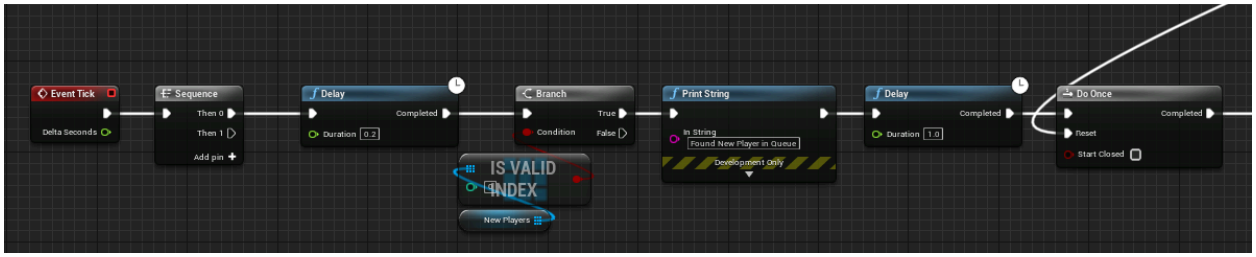
Do the same for “GameState,” and select your new GameState in the GameMode “Class Defaults.


Parent class: Game Mode
Details
Palette
Search Details
Actor Tick
Start with Tick Enabled
Tick Interval (secs)
0.0
Allow Tick Before Begin
Default
Altitude
3 Array elements
Alt Track
0
Game Mode
Delayed Start
Minimum Respawn Dela
1.0
Inactive Player State Lif
300.0
Max Inactive Players
16
Use Seamless Travel
Start Players as Specta
Pauseable
Classes
Game Session Class
GameSession
Game State Class
BP_21_GS
Player Controller Class
BP_21_PC
Player State Class
PlayerState
HUD Class
HUD
Default Pawn Class
None
Spectator Class
SpectatorPawn
Replay Spectator Player
PlayerController
Server Stat Replicator C
ServerStatReplicator
Game
Default Player Name
LOD
Cooking

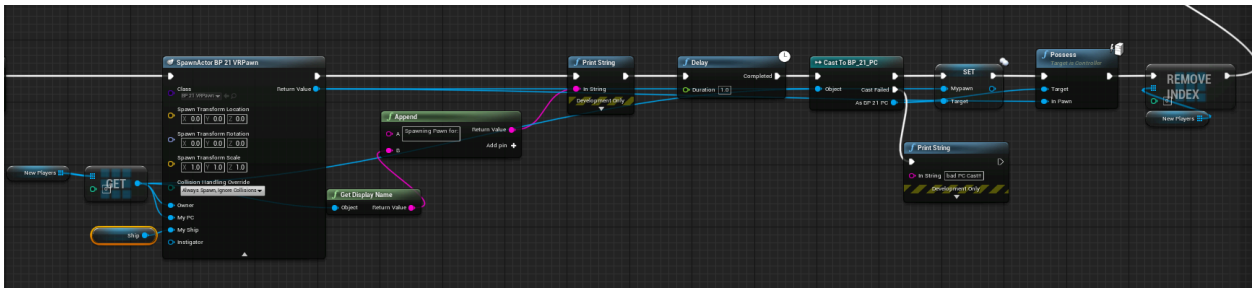
When a client connects to the server, this triggers an event in the GameMode called “EventOnPostLogin,” and it contains the info for that client’s PlayerController.

Sometimes the GameMode is much slower than you’d expect in dealing with multiplayer stuff. If you tell it to do too many things at once, stupid things happen. I like to use some workarounds involving delays and checks for this reason. When I get a new player connection, I add the PlayerController to an array because that is a fast operation and I can deal with him later without gumming up the rest of the logic.

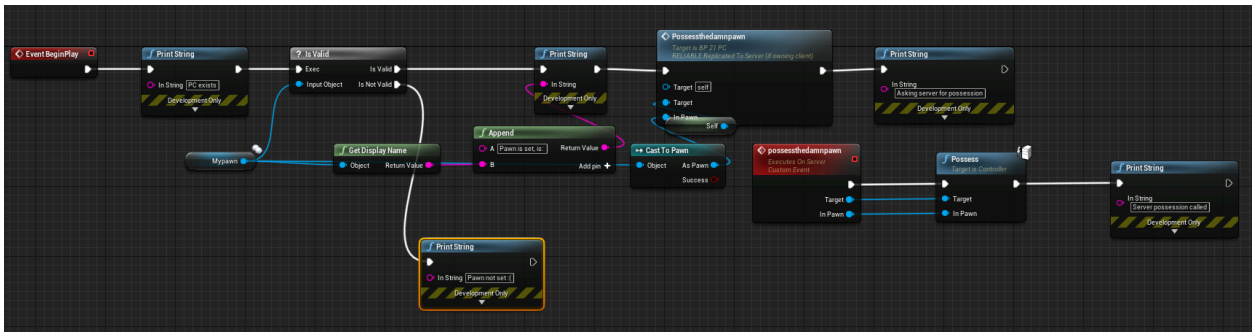




Now that we have the new player controller and some time to deal with it, we’ll spawn a Pawn for it to possess (set the GameMode’s default pawn to none, we like to do things explicitly).



Sometimes this doesn’t work properly, so I’ve added a backup possession function in the PlayerController. This function includes replication, which I’ll explain next.



Replication

Communication Rules

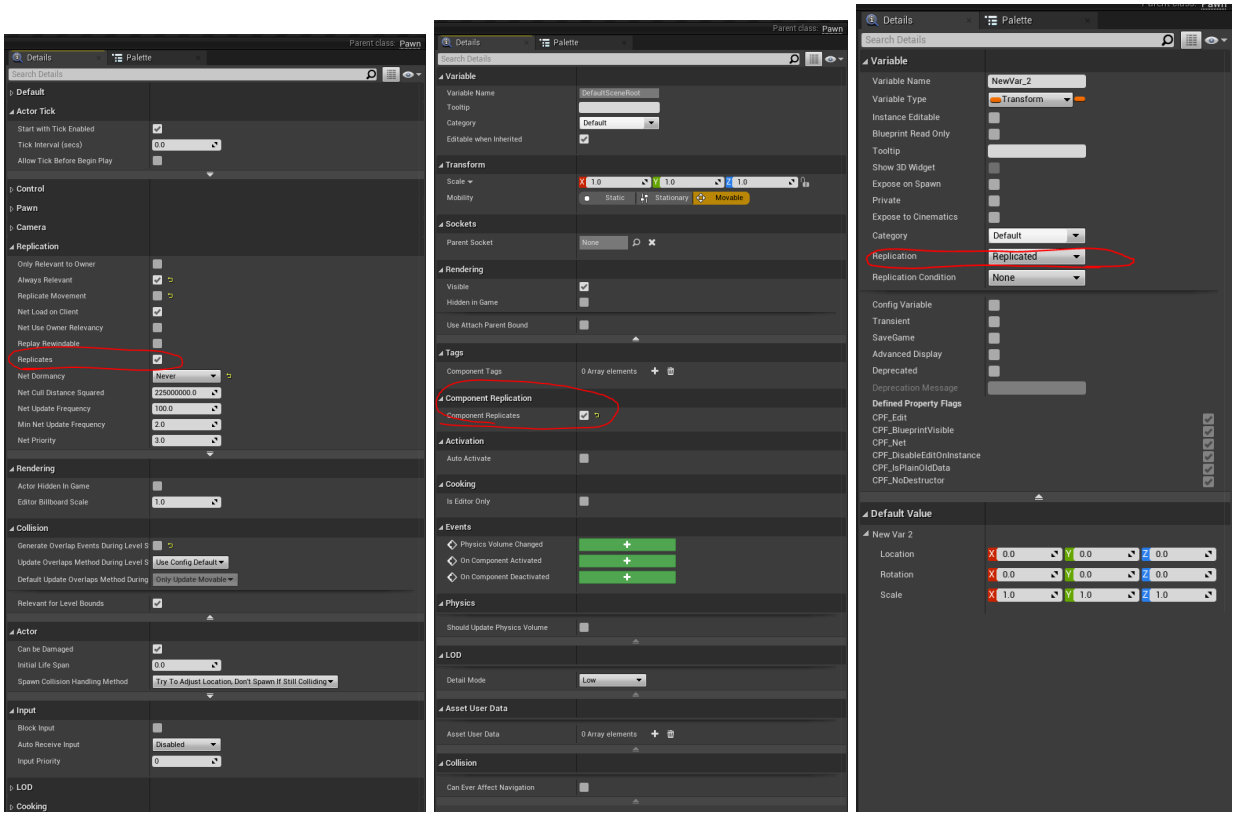
Clients do not talk to each other, they only talk to the server.

Only an actor owned by the client may send a message to the server (though any replicated actor receives messages from the server).

Object Replication

If a variable, component, or actor* is marked “replicate,” any time the server alters one of these objects it will update the changes to all the clients. It does not continuously sync the client-values for these objects with the server-values, it updates them on the client when they are changed on the server. The values may diverge if the client updates the value after the server does.

> *here ‘actor’ refers strictly to the root component or actor properties, not other components or arbitrary variables; it is worth noting that in order for any component or variable (or event) to be replicated, its parent actor must also be replicated



Actor, Component, and Variable Replication.

Event Replication

I do not like replicating variables or components using the above method if I can help it; I like to use event replication because I know exactly what, how, and where things will be replicated. Any “Custom Event” can be replicated.

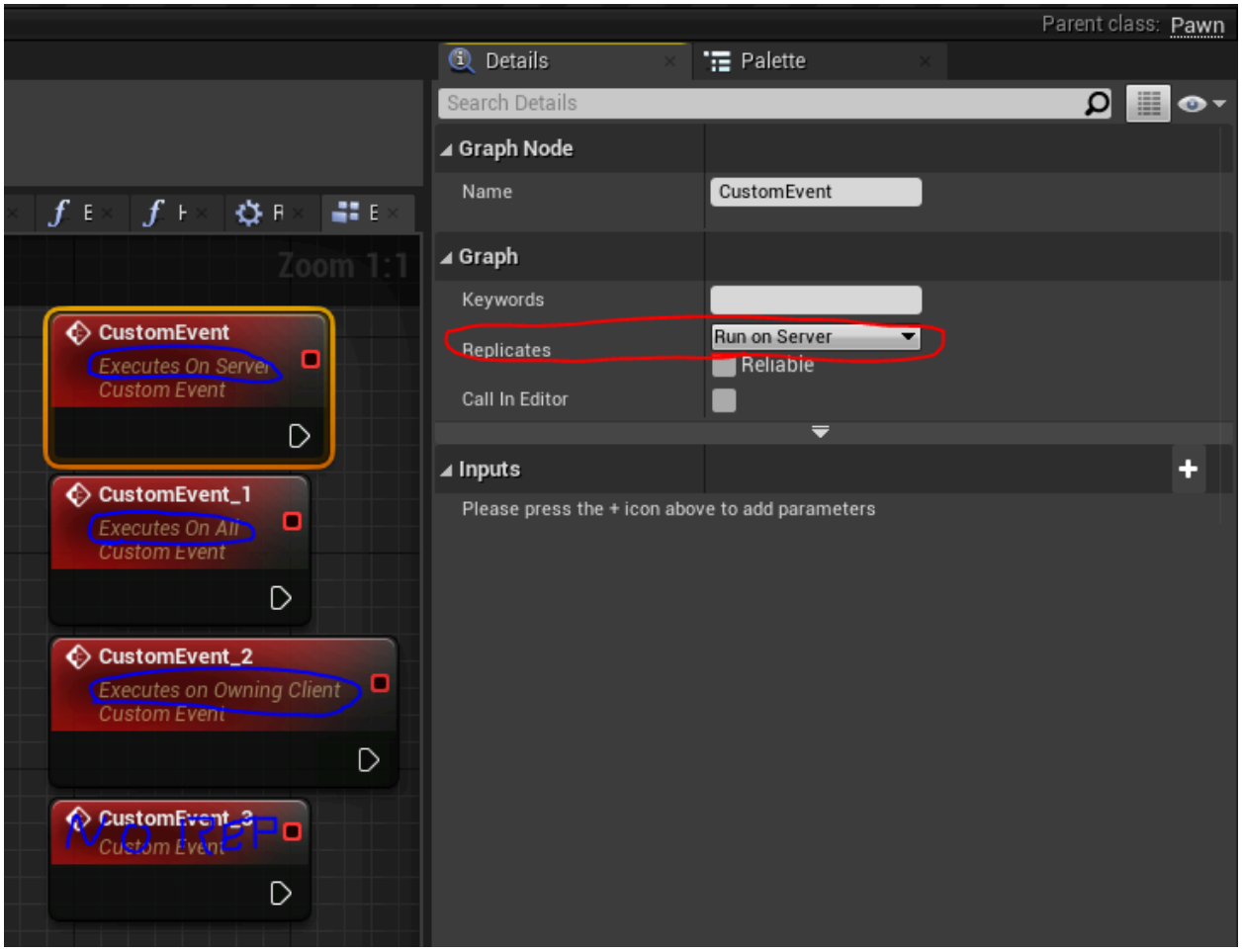
To add a custom event, right click on a blank spot in the Event Graph, begin typing “addcustomevent” until it finds it for you, then select it. It will spawn a red card for you. Give it a name. This custom event can now be passed variables and called as if it were a function or a macro. Before we continue with event replication, let’s quickly review what the difference is between Macros, Functions, and Events.

> A **macro** is a set of blueprint instructions that you expect to re-use frequently, but do not need to work in any other special way. My understanding is that when the blueprints are compiled, it actually treats any macros as if you had just copied and pasted the relevant code (right click on the macro and select “Expand Node,” this is how it looks to the program). I recommend using them for math operations or other similar things that don’t cause side effects (you’ll notice that by default, a new macro does not give you an execution pin - this is on purpose).

A **function** is its own instanced little world. Multiple copies of the same function can be running at the same time without bothering each other. It has its own local variables that don't exist until the function is called, are only available to that instance of the function, and are gone forever once the function completes. This goes for flow control as well. If you have a DoOnce node inside of a function, that instance of the function will indeed only execute the node once, but the next time you call the function it will forget all about that. I do not recommend putting side effects into Functions either, but this is more of a personal stylistic preference.

Events are not instanced. They have exactly one home on the event graph, and they start their own execution lines. If you call the same event more than once while it's still running, it can potentially interfere with itself (this may be desired). You can use events as if they were GOTO commands to hop around in the flow of your event graph, but this is considered dangerous. EventTick and OnEventBeginPlay are both examples of built-in events.

Events can be replicated. The server may replicate the event to all the clients, or only to the client who owns the actor the event is being called from. An actor owned by the local client may replicate an event from itself to the server.

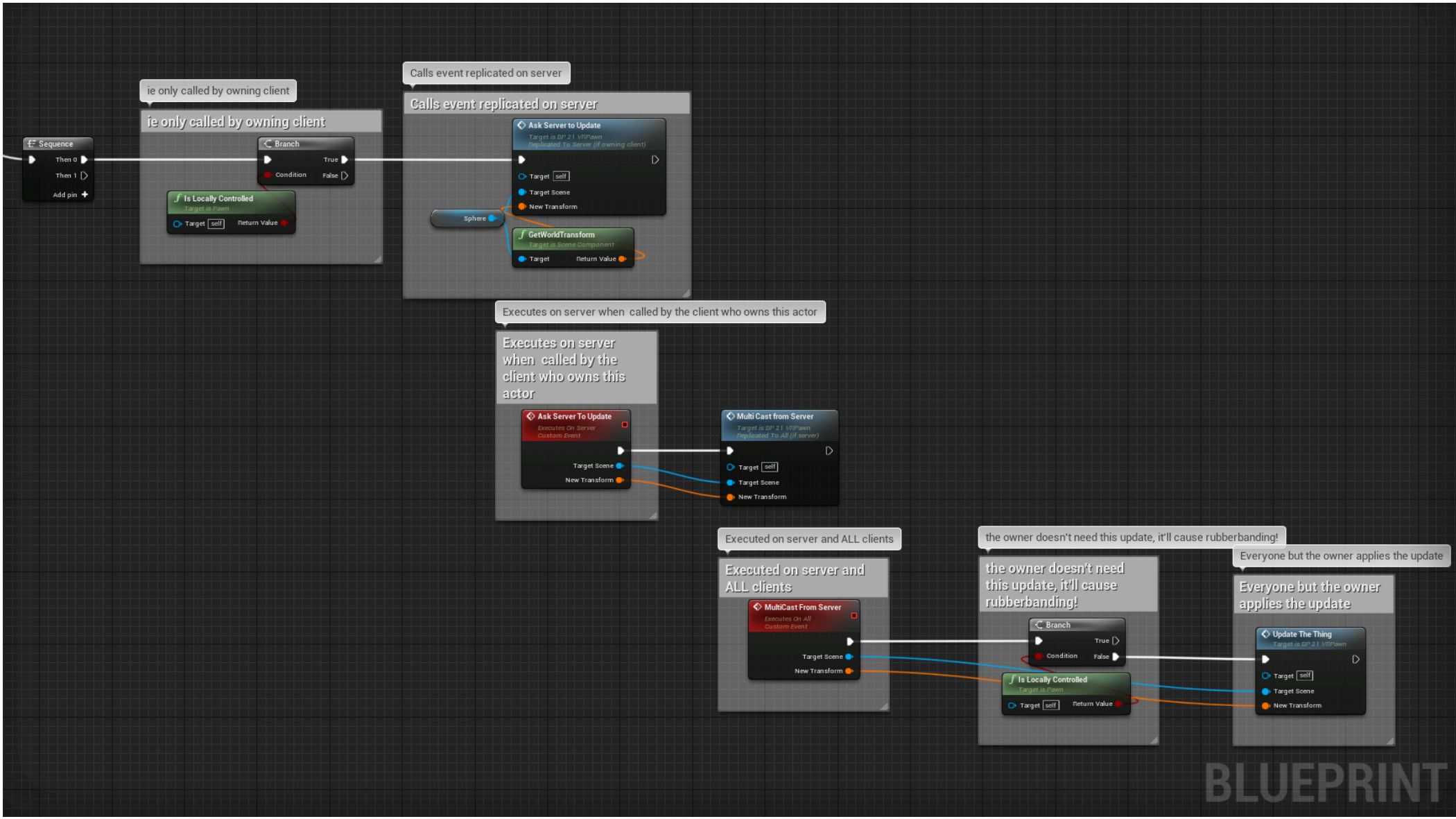


By default, events are not replicated, you need to select the appropriate option as shown above.

Run On Server: Can only be called on the client that owns the actor. It executes everything locally, and then the server does the same. If any part of the event touches a replicated variable or object, that will then be replicated to all the clients (including the one that called it, this can cause rubber-banding). Object Replication and Event Replication are separate, so just because you don't multicast an event doesn't mean its side effects won't be multicast. This is why I prefer to run everything explicitly through events and not objects.

Multicast: Can only be called on the server. If you call it from a client, it will execute locally (I think) and otherwise not do anything. You will sometimes need to run two events in sequence: one from the owning client to the server, and one from the server to all the clients (and if you don't want rubberbanding, you will need to do something about it).

Run On Owning Client: Can only be called on the server. Will run on the server and on the client that owns the actor only.



Making three custom events per class of update is a small price to pay for not having to model the state of every variable and object in three

different locations. On the other hand, straight variable replication appears to be substantially faster, so consider using that if you need speed over clarity.