Introduction sur le multijoueur sur UE4

(NB : J'espère que ce tuto vous sera utile et qu'il sera compréhensible par tous. Si vous avez des questions n'hésitez pas à m'envoyer un message sur Facebook « Antoine Gargasson » ou par mail antoine.gargasson@gmail.com)

(NB2 : Ce tuto est réalisé sur la version 4.8.3 de UE4)

Avant de commencer à entrer dans le vif du sujet, il est important de bien comprendre comment une structure serveur/clients fonctionne. Je vous propose dans ce tuto de nous pencher sur les bases du multijoueur sur UE4.

Il faut également savoir quelques petits termes :

- **Client** : C'est un joueur lambda qui se connecte sur le jeu.

- **Serveur** : Ça peut être un joueur, dans ce cas c'est un joueur/serveur. Sinon c'est un serveur dédié (non joueur). C'est l'entité qui contrôle tout ce qui se passe sur une partie.

- **Propriétaire** : Un propriétaire est un joueur qui joue sur sa propre instance du jeu. Il est propriétaire de ce qu'il créé dans sa partie. C'est une notion importante pour la suite car suivant qui est le propriétaire d'un objet, les clients pourront ou non interagir avec un objet. (voir section sur **l'autorité**)

Résumé du Framework

- L'objet **GameMode** (mode de jeu)existe seulement sur le serveur. Dans le cas d'un jeu solo le joueur est considéré comme serveur.

- Le **GameState** (état du jeu)existe sur le serveur et les clients, ainsi le serveur peut utiliser les variables répliquées du GameState pour garder tous les clients à jour sur les données du jeu.

- Le **PlayerController** (Contrôleur du joueur)existe sur le serveur Pour chaque joueur connectés sur le jeu. Sur les clients, seul le joueur possédé par le client possède un PlayerControllers. Cela signifie que les PlayerControllers ne sont pas le meilleur endroit pour stocker les propriétés répliquées qui intéressent tous les clients. A la place, préférez le PlayerState.

- Le **PlayerState** (état du joueur)existe pour chaque joueur connectés au jeu sur le serveur et chaque clients. Cette classe peut être utilisée pour stocker les propriétés répliquées qui seront utilisées par tous les clients et pas seulement le client propriétaire. On y trouvera par exemple le score individuel actuel de chaque joueur.

- Les **Pawns** (incluant les **Characters**) va aussi exister sur le serveur et tous les clients, et peut aussi contenir des variables répliqués et des événements. Le choix d'utiliser le PlayerState ou le Pawn pour certaines variables ou événements va dépendre de la situation. Mais la principale chose à garder en mémoire est que le PlayerState va persister tant que le joueur sera connecté alors que le Pawn non. Par exemple, si le Characters d'un joueur meurt pendant une partie, le Pawn peut être détruit et un nouveau sera créé quand le joueur réapparaîtra.

Ce qu'il faut donc retenir :

Les variables répliquées qui doivent être **remises à zéro** sur le joueur **meurt** ou disparaît (vie, mana, etc.) sont à mettre sur le **Character**.

Les variables locales qui sont **inchangées** lorsque le joueur **meurt** (inventaire) sont à mettre sur le **PlayerController**

Les variables répliquées qui restent inchangées et qui sont **indépendantes** de l'action d'un **joueur** (score) sont à mettre sur le **Player State**.

La réplication des acteurs

Une chose importante à comprendre est que les acteurs sont seulement répliqués du serveur aux clients. Il est impossible d'avoir un acteur répliqué d'un client au serveur. Bien sûr les clients ont quand même besoin de pouvoir envoyer des données au serveur et ils le font uniquement à partir des événement répliqués "Run On Server" (Exécuter sur le serveur).

L'autorité

Pour chaque acteur dans le monde, un des joueur connecté (client) est considéré comme ayant l'autorité sur cet acteur. Pour chaque acteur qui existe sur le serveur, le serveur a l'autorité sur cet actor, incluant tous les acteurs répliqués. Ce qui veut dire que quand la fonction "HasAuthority" (a l'autorité) est exécutée sur un client, et que la cible est un acteur répliqué sur ce client, cette fonction renverra faux. Vous pouvez aussi utiliser le "Switch Has Authority" (choix de l'entité autoritaire) comme une façon rapide d'exécuter des tâches différentes sur le serveur et le client sur un acteur répliqué.

Les variables

Dans le tableau ci contre, vous pourrez voir les différents types de réplication des variables sur vos acteurs.

None - Non répliquée	C'est la sélection par défaut pour une nouvelle variable. Cela signifie que la valeur ne sera pas envoyée par le réseau aux clients.
Replicated - Répliquée	Quand le serveur réplique cet acteur, il enverra cette variable aux clients. La valeur de cette variable sur le client qui la réceptionne sera automatiquement mise à jour. Ainsi, la prochaine fois qu'elle sera utilisée, elle reflétera la valeur sur le serveur. Bien sûr, quand on joue dans un réseau conséquent, la mise à jour mettra un peu plus de temps à être effectuée, cela dépendra de la latence sur le serveur. Souvenez vous que les variables répliquées vont seulement dans un sens, du serveur aux clients. Pour envoyer des données d'un client à un serveur, voir la section événements.
RepNotify - Répliquée avec	La variable va être répliquée comme dans l'option "répliquée" mais en plus elle va créer une fonction "OnRep_function" dans votre blueprint. Cette fonction va être appelée automatiquement par le moteur sur le client et sur le serveur uniquement quand la valeur de la variable va changer. Vous

notification	êtes libre d'implémenter cette fonction où vous voulez.

Sources :

https://docs.unrealengine.com/latest/INT/Gameplay/HowTo/Networking/ReplicateVariable/Blueprints/index.html

Création et destruction

Quand un acteur répliqué est créé sur le serveur, cela est communiqué aux clients et ils vont également créer automatiquement une copie de cet acteur. Mais, puisqu'en général la réplication n'est pas effectuée du client au serveur, si un acteur répliqué est créé sur un client, cet acteur existera seulement sur le client qui l'a créé. Ni le serveur, ni les autres clients vont recevoir une copie de cet acteur. Le client qui créé va avoir l'autorité sur cet acteur. Cela peut être quand même utile pour des acteurs cosmétiques qui n'ont pas d'effets sur le gameplay. Mais, pour les acteurs qui ont un effet sur le gameplay et qui doivent être répliqués, ces derniers doivent être créés sur le serveur. C'est utile pour créer des objets uniquement destinés au joueur comme des munitions ou du loot.

De même lorsqu'il s'agit de détruire un acteur répliqué. Si le serveur en détruit un, tous les clients vont détruire leur copie. Les clients sont libres de détruire les acteurs sur lesquels ils ont autorité, c'est à dire les acteurs qu'ils ont eux même créés, tant qu'ils ne sont pas répliqués aux autres joueurs et qu'ils n'ont pas d'effets sur eux. Si un client essaye de détruire un acteur dont il n'a pas l'autorité, la demande de destruction sera ignorée. Le point clé ici est la même que pour la création d'acteurs : **Si vous devez détruire un acteur répliqué, faites le sur le serveur.**

Source :

https://docs.unrealengine.com/latest/INT/Gameplay/HowTo/Networking/ReplicateActor/Blueprints/index.html

La réplication des fonctions (blueprint)

Il y a 3 principaux types de fonctions répliquées : La **Multicast**, la **Run On Server** et la **Run On Ownin Client**. La fonction Multicast doit être appelée sur le serveur, où elle est exécutée, et est envoyée automatiquement à tous les clients. Les fonctions serveur sont appelées par un client et ensuite exécutées uniquement sur le serveur. Les fonctions client sont appelées par le serveur et ensuite exécutées uniquement sur le client propriétaire.

https://docs.unrealengine.com/latest/INT/Gameplay/HowTo/Networking/ReplicateFunction/Blueprints/index.html

Appartenance

Un concept important à comprendre quand on travail sur du multijoueur, et spécialement avec les événements répliqués, est quelle connexion est considérée comme le possesseur d'un acteur ou d'un composant. (https://docs.unrealengine.com/latest/INT/Gameplay/Networking/Actors/OwningConnections/index.html). Par exemple, sachez que les événements "Run on Server" (exécuter sur le serveur) peuvent seulement être appelés d'un acteur (ou ses composants) que le client possède. Habituellement, cela signifie que vous pouvez uniquement envoyer un événement "Run on Server" à partir de cet acteur ou à partir d'un composant d'un de ces acteurs :

- Le PlayerController du client,
- Un Pawn que le PlayerController du client possède, ou
- Le PlayerState du client.

Sachez également que pour un événement "Run on owning client" (exécuter sur le client possédé), ces événements doivent aussi être exécuter sur un de ces acteurs. Autrement, le serveur ne saura pas à quel client envoyer l'événement et il sera uniquement exécuter sur le serveur !

Événements

Not Replicated - Non répliqué	Par défaut. Signifie qu'il n'y aura pas de réplication pour cet événement. Si c'est appelé sur un client, il sera uniquement exécuté sur ce client et si c'est appelé sur le serveur, il sera uniquement exécuté sur le serveur.
Multicast - Envoyé à tous	Si un événement multicast est appelé sur le serveur, il sera répliqué à tous les clients connectés indépendamment de à qui appartient cet objet. Si un client appel un événement multicast, il sera traité comme s'il n'était pas répliqué et sera uniquement exécuté sur le client qu'il l'a appelé. Rappelez vous bien que l'on doit passer par le serveur absolument !
Run on Server - Exécuté sur le serveur	Si cet événement est appelé depuis le serveur, is sera uniquement exécuté sur le serveur. S'il est appelé depuis un client, sur un objet que le client possède, il sera répliqué au serveur et exécuté sur le serveur. Un événement "Run on server" est la principale méthode pour un client pour envoyer des informations au serveur.
Run on Owning Client - Exécuté sur	Si c'est appelé depuis le serveur, cet événement vas s'exécuter sur le client à qui appartient l'objet cible. Depuis que le serveur peut posséder des acteurs, un événement "Run on owning client" peut être exécuté sur le serveur en dépit de son nom. S'il est appelé depuis un client, l'événement vas être traité comme s'il n'était pas répliqué et sera uniquement exécuté

le client possédé	sur le client qui l'a appelé.

Le tableau suivant vous indique où sera exécuté la fonction en fonction d'où elle est envoyé et de son type :

DEPUIS LE SERVEUR :

	Non répliquée	Multicast (A tous)	Exécuté sur le serveur	Exécuté sur le client qui possède
Le client possède la cible	Serveur	Serveur et tous les clients	Serveur	Client qui possède
Le serveur possède la cible	Serveur	Serveur et tous les clients	Serveur	Serveur
Cible non possédée	Serveur	Serveur et tous les clients	Serveur	Serveur

DEPUIS UN CLIENT :

	Non répliquée	Multicast (A tous)	Exécuté sur le serveur	Exécuté sur le client qui possède
Target owned by invoking client	Client appelant	Client appelant	Serveur	Client appelant
Target owned by a different client	Client appelant	Client appelant	Ignorée	Client appelant
Server-owned target	Client appelant	Client appelant	Ignorée	Client appelant
Unowned target	Client appelant	Client appelant	Ignorée	Client appelant

Prise en compte de l'arrivé d'un joueur en cours de partie

Une chose à garder en mémoire quand vous utilisez un événement répliqué pour communiquer au GameState les changements est comment il interagit avec un jeu qui supporte l'arrivé d'un joueur en cours de partie. Si un joueur rejoins une partie en cours, tous les événements répliqués qui ont eu lieu avant que le joueur est rejoint ne seront pas exécutés pour le nouveau joueur. Le soucis ici est que si vous voulez que votre jeu

supporte le join-in-progress (rejoindre en cours de partie), c'est généralement mieux de synchroniser les données importantes via des variables répliquées. Une solution qui est souvent utilisée est que le client qui réalise des actions dans le monde, notifie le serveur sur les actions via un événement "Run on server" et dans l'implémentation de cette fonction, le serveur met à jour certaines variables répliquées basées sur ces actions. Ensuite les autres clients, qui n'ont pas réalisé l'actions, voient tout de même le résultat grâce aux variables répliquées. De plus, n'importe quel client qui rejoint en cours après que l'action est eu lieu va aussi voir l'état correct du monde car ils recoivent les valeurs les plus récentes des variables répliquées du serveur. Si le serveur avait à la place seulement envoyé des événements, le joueur qui aurait rejoint en cours de partie n'aurait pas su des actions qui ont été effectuées !

Confiance (Reliability)

Quelque soit l'événement répliqué, vous pouvez choisir quel niveau de confiance il convient : Confiant ou pas confiant.

Les événements confiants sont assurés d'atteindre leur destination (en considérant que les lois sur l'appartenance plus haut soient respectées), mais elles consomment plus de bande passante et de la latence potentiellement pour garantir ce succès. Essayez d'éviter d'envoyer trop d'événements de confiance trop souvent, comme à chaque frame, car la mémoire interne du moteur dédiée aux événements de confiance pourrait être surchargé. Quand cela arrive, les joueurs concernés seront déconnectés !

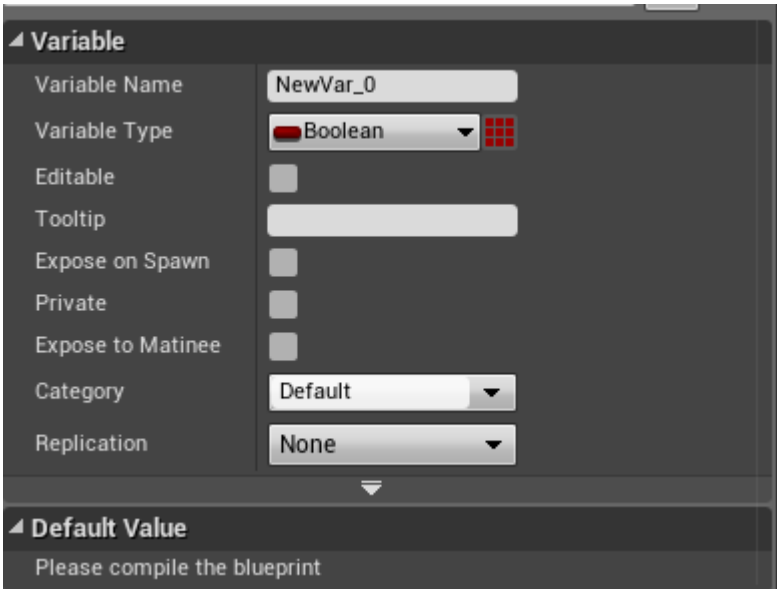
Les événements non confiants ne sont pas assurés d'atteindre leur destination, dans le cas ou des paquets sont perdu sur le réseau ou si le moteur détermine qu'il y a beaucoup de trafic haute priorité à envoyer par exemple. En conséquence, les événements non confiants utilisent moins de bande passante que les confiants et peuvent être appelés plus souvent sans crainte.

Toutes ces informations peuvent être retrouvées sur le site dédié à la documentation de Unreal Engine à l'adresse suivante : https://docs.unrealengine.com/latest/INT/

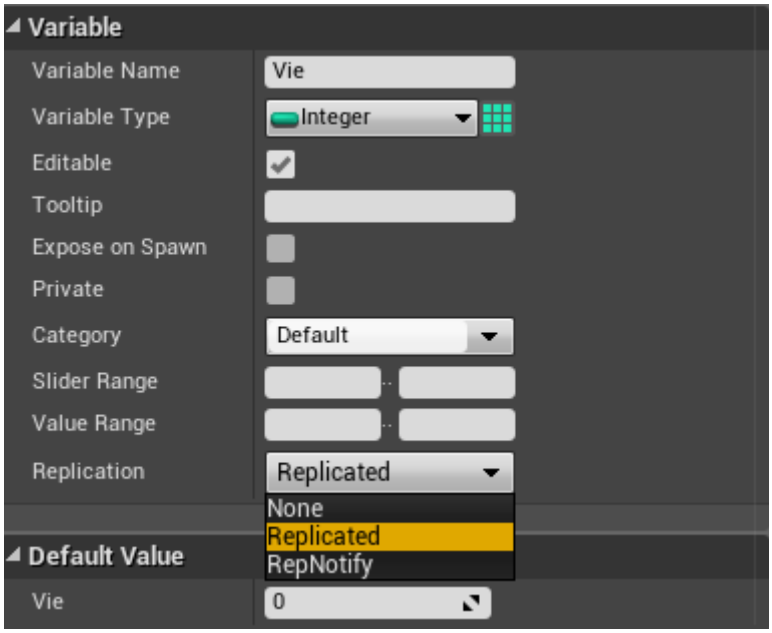
La pratique

Passons à la pratique ! Je vais vous montrer où se situe les changements de type et comment tout est représenté !

Commençons par les variables. Lorsque vous créez votre variable vous obtenez quelque chose comme suit :



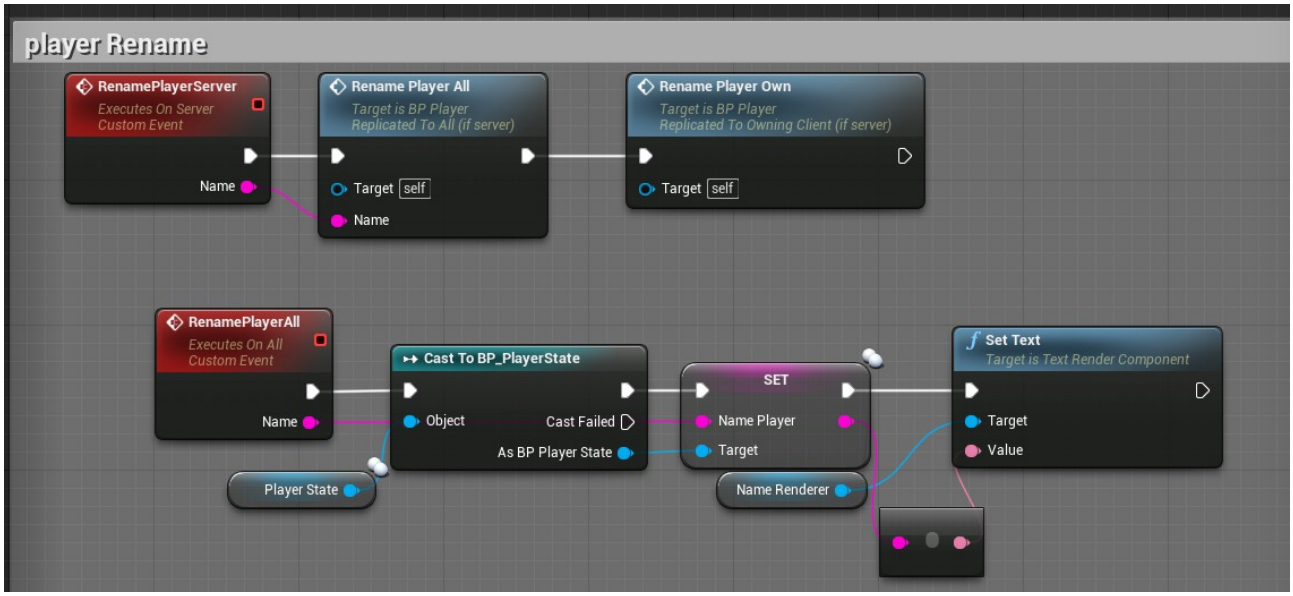
Vous nommez donc votre variable et changez son type à convenance. Vous vous rendez ensuite dans la partie qui nous intéresse : Replication.



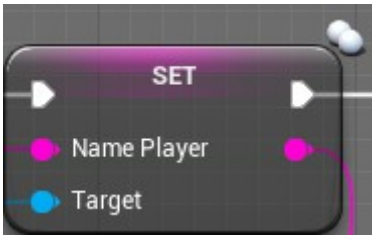
Vous avez le choix entre ce qui vous avez vu plus haut : Non répliquée, Répliquée ou Répliquée avec notification.

En ce qui concerne les événements. Ci dessous vous pouvez voir comment un client envoi des données aux autres clients. Il appel tout d'abord une fonction exécutée sur le serveur (RenamePlayerServer) qui appellera une fonction multicast (sur tout les clients) (RenamePlayerAll). Tous mes clients auront donc bien l'information du changement de nom.

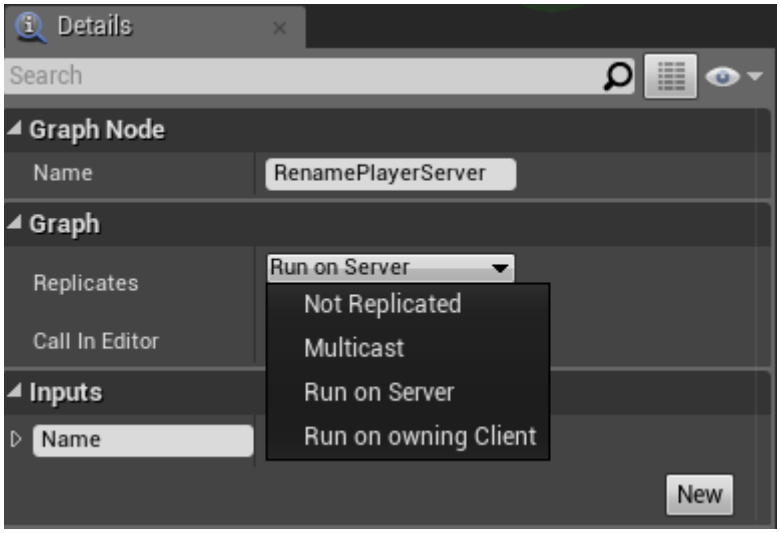
Il est important d'avoir une convention de nommage pour vous y retrouver. Ici j'ai choisi de mettre le nom de la fonction et ensuite où elle est exécutée.



Quand votre variable est répliqué, vous pouvez apercevoir deux petites boules au dessus des cases l'utilisant :

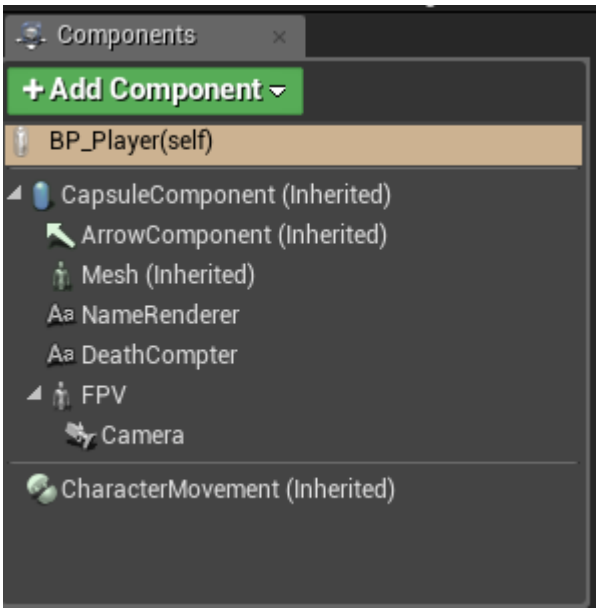


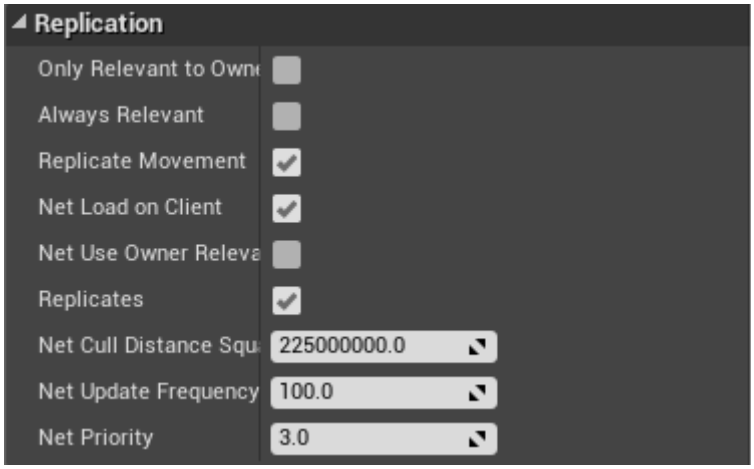
Comme pour les variables, si vous voulez changer le type d'exécution, rendez-vous dans les details dans la section Replicates.



Ici pourrez choisir le type d'exécution de votre fonction et son niveau de confiance (juste sous le Replicates).

Si vous souhaitez que votre acteur entier soit répliqué sélectionnez le et rendez-vous dans la partie Replicates.





Si vous cochez Replicates, votre acteur sera répliqué aux autres client dans le monde mais ses variables ne seront pas répliquées si vous ne le spécifiez pas en les créant.

Pour toute éventuelles questions, remarques, détails ou demandes, n'hésitez pas à prendre contact avec moi ! (Cf : début du tuto)