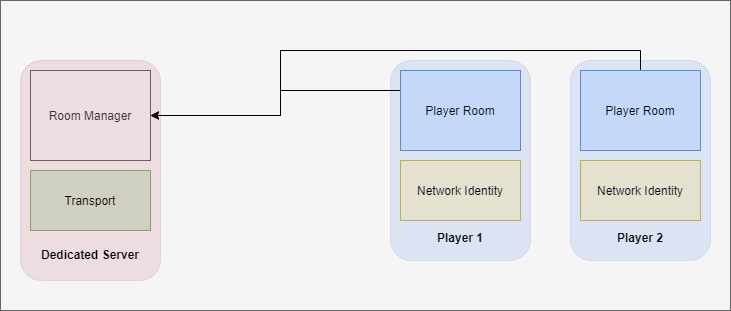
Document d’analyse Networking - Snow

# Mirror

## Présentation

Mirror est un plugin pour Unity3D qui permet de gérer la partie réseau d’un jeu. Il permet d’avoir dans un seul code la gestion des clients et du serveur, ainsi que des classes de base à partir desquelles on peut créer ses propres classes pour avoir un comportement personnalisé. Il permet de gérer à la fois un mode P2P ou un mode Serveur Dédié.



Mirror se décompose en :

* Un Serveur

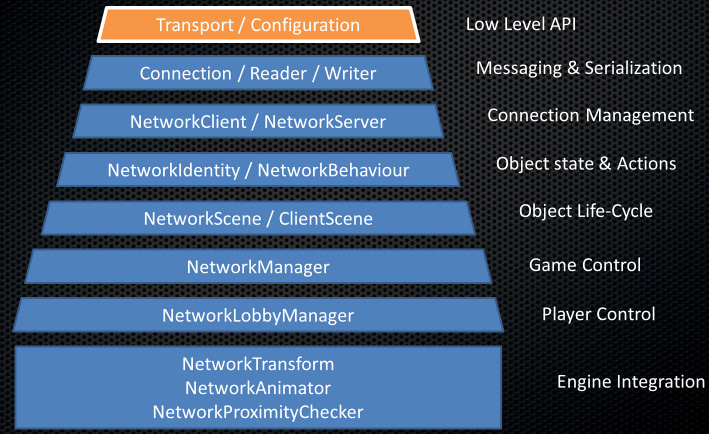
Le Serveur est l’instance du jeu à laquelle les autres joueurs vont se connecter. Il permet de gérer les comportements en ligne tel que les scores, placer les objets et envoyer les informations aux joueurs.

* Des Clients

Les Clients sont les instances du jeu à partir desquelles les joueurs vont pouvoir jouer. Ils pourront se connecter à un serveur pour jouer en ligne

## Fonctionnement

L’API de Mirror se compose des différentes couches suivantes.



La partie la plus basse est la partie transport. C’est ici qu’on défini le protocole avec lequel on veut communiquer entre serveur et joueurs.

Les classes NetworkClient et NetworkServer sont les classes de base qui correspondent respectivement au client et au serveur.

Le NetworkIdentity est un component qui permet de gérer les identifiants réseaux. Il fonctionne de pair avec la classe NetworkBehaviour qui se substitue au MonoBehaviour pour les objets réseaux.

Les classes NetworkScene et ClientScene se chargent de la gestion des scènes. Par défaut c’est le NetworkScene qui envoie les informations à chacun des ClientScene des joueurs.

Le NetworkManager gère à haut niveau la partie réseau. Il va permettre de connecter les joueurs au serveur, de gérer les scènes etc…

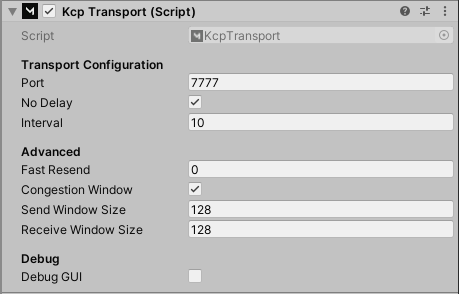
Le NetworkRoomManager (anciennement Lobby) permet de gérer les comportements spécifiques d’une room et des RoomPlayer. Il hérite de NetworkManager.

## Transport

Mirror permet de choisir le transport sur les couches de bas niveau. Ils s’intègrent par des component indépendants attachés au Network Manager. Mirror utilise par défaut KCP Transport qui communique en TCP, mais il permet de gérer la communication sur la couche de transport avec d’autres components :

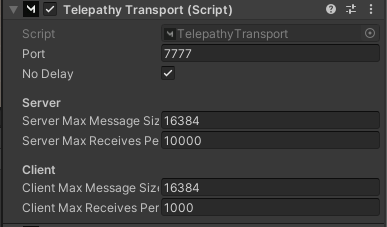
* KCP Transport (TCP)

KCP est le nouveau transport par défaut de Mirror. Il est plus rapide est plus stable que l’ancien composant Telepathy. KCP est a 100% en C# et possède néanmoins la rapidité d’un transport natif en C. Il fonctionne sur toute les plateformes exceptée WebGl.



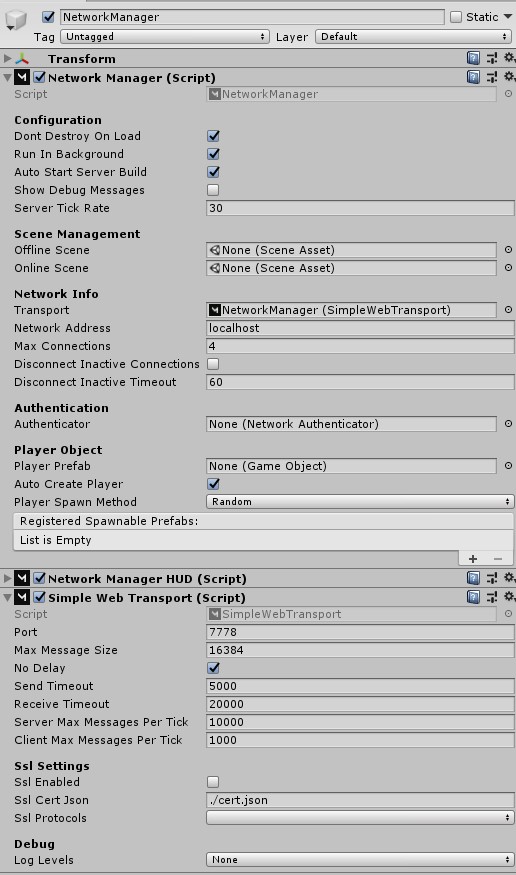
* Libuv2k (TCP)
* Telepathy (TCP)

Telepathy est composant de transport en TCP en C#. Il était jusqu’à récemment utilisé dans Mirror comme composant de transport par défaut. C’est un transport rapide et stable. Il utilise un processus par connexion ce qui permet d’exploiter entièrement les processeurs multicœurs.



* SimpleWebTransport (WebGl)

Le transport pour le WebGl est SimpleWebTransport. Il permet la communication avec le protocole websocket dans les builds Unity en WebGl. Il prend en charge le protocole wss pour les pages en https.



* Ignorance (UDP)

Ignorance est un transport en UDP qui se base sur la bibliothèque de network ENET en C à partir d’une branche personnalisée de ENet-Csharp. Il permet une gestion de séquences UDP reliable et unreliable pour les OS 64bits (Windows, Mac et Linux) et mobile (iOS, Android).

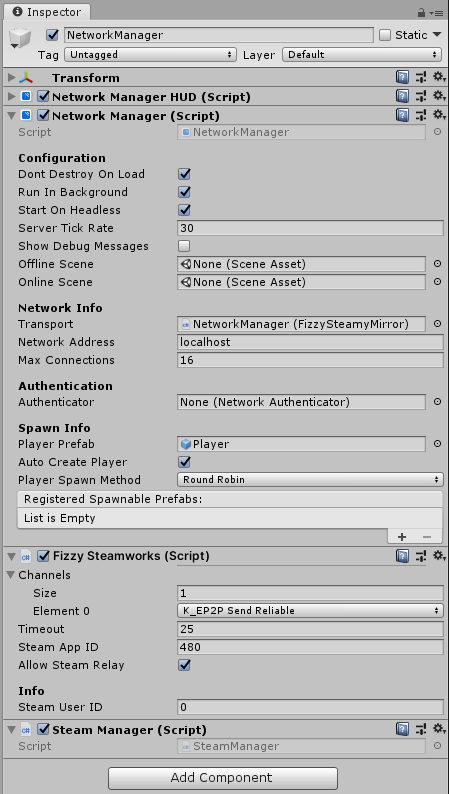
Ignorance a été créé dans le but de remplacer l’ancien système de network LLAPI qui est désormais obsolète.

* LiteNetLibTransport (UDP)

Ce transport permet une utilisation simple et légère d’une communication en UDP.

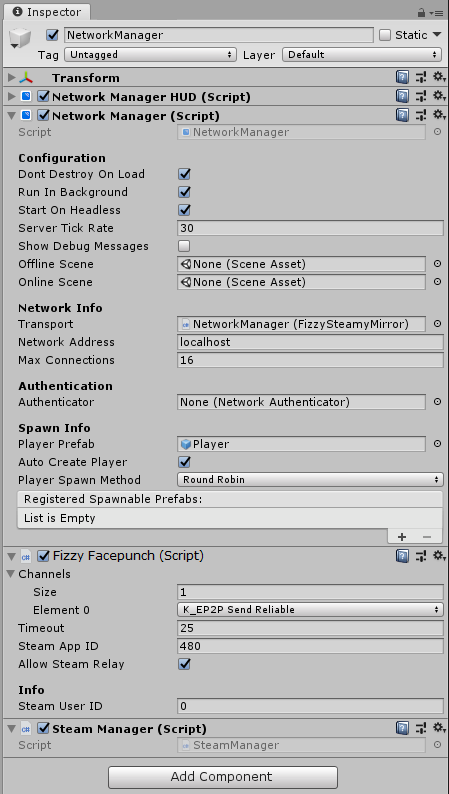
* FizzySteamworks (Steam)

FizzySteamworks est un transport qui se base sur l’utilisation du service P2P de Steam à travers le wrapper Steamworks.Net



* FizzyFacepunch (Steam)

FizzyFacePunch est un transport qui se base sur l’utilisation du service P2P de Steam à travers le wrapper FacePunch.Steamworks



## Synchro

Network Identity/Network Behaviour

Dans Mirror la synchronisation se fait de plusieurs manières différentes. Dans un premier temps c’est le component NetworkIdentity qui permet aux GameObjects d’être reconnus sur le serveur et de leur attribuer leurs propriétés réseau (leur networkId, isLocalPlayer…). Le NetworkIdentity existe sous la forme d’un Component à ajouter sur le GameObject.

La classe NetworkBehaviour permet de gérer toutes les commandes de synchronisation. Pour ajouter des comportements réseax à un objet, il faut que le script de celui-ci hérite de NetworkBehaviour et pourra ainsi accéder aux méthodes et tags réseaux. Il fonctionne de pair avec le NetworkIdentity.

Pour qu’un objet réseau fonctionne correctement il doit posséder le component NetworkIdentity et un script héritant de NetworkBehaviour.

Il existe des tags qui permettent de de gérer la synchronisation de différentes manières :

[ServerCallBack]/[Server]

Les tags **[ServerCallBack]**/**[Server]** permettent de bloquer l’utilisation de la méthode concernée si elle est appelée sur un client. **[ServerCallBack]** enverra un warning si un client tente d’y accéder alors que Server ne reverra rien. Ces tags sont donc utilisés pour la partie serveur.

[Command]

Ce tag permet d’appeler depuis un client cette méthode sur le serveur. Il permet de gérer les actions qui doivent avoir lieu sur le serveur et non pas sur le client (instancier un objet serveur par exemple)

[ClientRpc]/[TargetRpc]

Ce tag permet au serveur d’appeler depuis un objet présent sur le serveur une méthode sur l’objet client. Il permet de gérer des comportements qui doivent avoir lieu sur les clients et pas sur le serveur (animations)

Comme les ClientsRpc sauf qu’il permet depuis un client d’atteindre un objet sur un autre client à travers sa connexion.

[SyncVar]

Ce tag permet, sur des variables, de définir qu’elles doivent être partagées sur toutes les instances de l’objet et donc être mises à jour régulièrement.

NetworkTransform

Le NetworkTranform est un composant qui permet de gérer la synchronisation d’un GameObject à travers le réseau sous l’autorité du Serveur. Pour fonctionner il doit également posséder un NetworkIdentity qui permet au Serveur de reconnaitre le GameObject sur le réseau et lui attribuer un Netork ID.

## Room

Room Manager

Le RoomManager gère l’ensemble de la Room. Il hérite de la classe NetworkManager à laquelle il rajoute les fonctions de gestion de Room. Le RoomManager utilise la classe NetworkManagerHUD pour gérer l’UI. A travers cette UI le RoomManager gère différents modes :

* Un mode client, il appelle alors la méthode NetworkManager.StartClient
* Un mode Hôte, dans lequel il gère un serveur et un client à la fois. Il appelle la méthode NetworkManager.StartHost
* Un mode Serveur, dans lequel il gère un serveur dédié. Il appelle alors la méthode NetworkManager.StartServer

Tout comme le NetworkManager il fonctionne sur des boucles d’appel (OnServerSceneChange) qui permettent la personnalisation des comportements des classes de base de Mirror.

Il permet la gestion des spawns des GameObjects. La gestion des spawns se fait de 2 manières différents, soit par l’instanciation réseau sur le serveur puis la mise à jour sur les clients, soit par l’instanciation de GameObjects lors du chargement d’une Scène.  
Le RoomManager possède également un slot Player pour ajouter le préfab du joueur. Lorsqu’un client se connecte le RoomManager lui crée un GameObject à partir du prefab Player.  
A la différence du NetworkManager le RoomManager possède également un slot pour un RoomPlayer, qui correspond à un Profil Joueur.

Le RoomManager gère également les Scènes. Il possède 2 slots distinct pour y ajouter des Scènes :

* La OfflineScene, qui correspond à la scène sur laquelle le joueur va se retrouver lorsqu’il n’est pas/plus connecté à un serveur. On y trouve les options de connection.
* La OnlineScene, qui correspond à la scène lancée lorsque le client est connecté à un serveur. On y retrouve les options de jeu du serveur. Le RoomPlayer est instancié au moment de la connexion à un serveur, il apparait donc sur la OnlineScene. De plus sur cette Scène le RoomManager va vérifier l’option « Ready » chez les joueurs pour savoir s’il doit lancer la partie.

En plus de ces 2 Scènes une 3eme scène existe, héritée de NetworkManager, la GameplayScene. Celle-ci correspond à la scène sur laquelle le jeu va se lancer. C’est sur celle-là que le RoomManager va instancier le Player.

Pour finir le RoomManager permet de gérer la manière dont les joueurs vont être placé sur les points d’apparition. Soit de manière aléatoire parmi les points d’apparition avec la méthode Random, ou bien en plaçant les joueurs les uns après les autres en itérant sur les différents points d’apparition.

Room Player

Le Room Player permet de gérer les informations du joueur de manière globale (pseudo, stats, xp…). Il indique au RoomManager lorsque le joueur a passé la variable « Ready » à True. Il se crée lors de la connexion avec le Room Manager.

## Gestion des scènes

Les scènes sont gérées par le RoomManager qui prend les scènes en entrée en tant que prefabs. Il se compose d’une RoomScene, d’une OfflineScene et d’une GameplayScene. Le Game Manager permet de changer de scènes de manière dynamique entre les 3 trois scènes qu’il gère.

On peut également changer de scène en accédant au Server Manager qui possède une méthode prévue à cet effet.

## Intégration

L’intégration des différents modules avec Mirror est bien supportée pour le New Input System ainsi que pour les animations qui sont gérées côté serveur puis mise à jour sur les clients à l’aide de RPC