Architectures de développement adaptées aux MMOGs

Blineau Suliac
Ecole Nationale des Jeux et Médias Intéractifs Numériques
Angoulême, France
suliac.blineau@orange.fr

RÉSUMÉ

Dans ce document, après avoir fait un état de l'art des techniques actuelles de développement de MMOGs, nous allons présenter une solution adaptée aux jeux multijoueurs. L'architecture proposée se concentre sur l'optimisation des échanges réseaux lors d'une partie pour un RPG de type rogue-like à plusieurs joueurs avec un unique serveur de jeu. Nous proposons une architecture distribuée où de nombreux clients viennent se connecter au(x) serveur(s) et nous expliquerons comment nous avons conçu le projet pour notamment séparer la partie visuelle du client (gérée par Unity) de la partie logique du jeu (fait en C# sous forme de Dll) et gérer les échanges entre un serveur en C++ et des clients en C#. La solution prévoit la division d'un monde virtuel en cellules fixes de tailles égales pour gérer plus facilement la zone d'intérêt des joueurs. Nous gérons également l'optimisation des échanges par un système publish/subscribe d'événements. Nous nous attarderons finalement sur la gestion des données à synchroniser sous forme de TimeLine (avec un timestamp) pour faciliter et optimiser la technique de synchronisation utilisée : le 'local-lag'.

CONCEPTS CCS

• **Networks** → Network architectures;

MOTS-CLÉS

Network, LocalLag, DeadReckonning, Distributed Architecture, Interest management, Architecture non-déterministe, Loadbalancing, UDP, TCP, Serilisation

1 INTRODUCTION

Au cours des dernières années le nombre de jeux multijoueurs a explosé, et ceux *massivement* multijoueurs (pouvant accepter dizaines voir centaines de milliers de joueurs simultanément) n'ont jamais été aussi populaires. Il suffit de voir le succès de World of Warcraft (même après plus de 14 ans) ou League of Legend pour se rendre compte qu'ils sont au centre de l'attention de la scène vidéoludique. League of Legend estimait avoir **100 millions** de joueurs actifs par mois en 2016 [9] et World of Warcraft 10 millions de joueurs abonnés en 2014 [8]. De nombreux autres en possèdent probablement encore plus, mais il est intéressant de remarquer les spécificités des deux premiers.

League of Legend est un MOBA (*multiplayer online battle arena*), un jeu d'action-stratégie, qui voit s'affronter dix joueurs répartis en deux équipes de cinq. La tâche peut sembler simple : de nombreux jeux gèrent une dizaine de personnes relativement facilement. Mais il ne faut pas oublier que le League of Legend comptait une centaine de millions de joueurs. Il faut donc pouvoir supporter cette charge, qui est présente sur l'ensemble de l'infrastructure de Riot Games. Étant basé sur de la stratégie, avec un affrontement direct, il est impensable d'avoir des désynchronisations entre les joueurs sous peine de casser complètement l'expérience de jeu. Il faut donc veiller à optimiser les échanges tout en s'assurant de la synchronisation des clients.

World of Warcraft est un MMORPG (*massively multiplayer online role-playing game*) où vous incarnez une personnage fictif dans un monde virtuel et persistant. Les gens jouent dans ce même monde et peuvent se rejoindre et se rencontrer virtuellement. Il y a donc potentiellement plusieurs centaines de joueurs qui devront pouvoir interagir avec un monde virtuel. D'autant plus que certains points d'intérêt attire des foules considérables à des endroits très localisés. Comment gérer le fait qu'un joueur n'ait pas eu le temps de recevoir l'information qu'il est attaqué alors que pour un autre joueur c'est le cas ? À quel point différence de perception entre les différents clients est elle acceptable ?

Ainsi la grande problématique de ces expériences en ligne est de savoir comment gérer le nombre de personnes souhaitant rejoindre et interagir sur ces mondes virtuels et comment régir les échanges entre eux.

Il est nécessaire de prévoir des architectures avec plusieurs serveurs. Cela oblige donc les créateurs de MMOG à trouver des solutions pour répartir la charge sur différentes machines en temps réel : le Load Balancing. Mais un autre problème intervient aussitôt : comment savoir à quel serveur assigner des joueurs ? Certains se contentent d'un découpage statique des régions virtuelles [17, 18], quand d'autres le rendent dynamique [2, 4, 11, 12, 22]. Le Load-Balancing est très lié à une partie plus liée au développement : l'Interest Management. Cette technique est la gestion de la zone d'intérêt du joueur dans le jeu : c'est est très important, au-delà de l'aspect sécurité, pour l'optimisation des échanges. Filtrer en amont les paquets équivaut à moins de trafic [2, 5, 10–12, 21]. Finalement nous savons qu'il est important de gommer (ou de contrôler) les différences entre les vues des joueurs, il faut donc trouver des techniques pour compenser cela. D'une part il faut déterminer si l'on souhaite créer un système déterministe ou non [17] (c'est-à-dire que tout peut être prévu, avec peu d'informations, ou pas). Mais il faut également des techniques de compensation des différences entre les joueurs : c'est ici que le Dead Reckonning ou le local-lag vont jouer un rôle [19].

Ce document sera séparé en deux grandes parties. La première consistera en un état de l'art des différentes techniques utilisées. Nous évoquerons le Load Balancing, l'Interest Management et quelques techniques de compensation de désynchronisation. Ensuite nous verrons une proposition d'architecture adaptée à un projet de taille réduite en se concentrant principalement sur l'architecture générale de la solution, l'Interest Managment et la compensation des différences entre les clients.

2 ETAT DE L'ART

2.1 Load Balancing

Lorsque l'on commence à rechercher des informations sur le réseau dans les jeux vidéo très vite on tombe sur comment les développeurs équilibrent la charge sur différents serveurs. C'est ce qu'on appelle plus communément le Load Balancing.
Une des premières techniques pour gérer les importantes charges serveurs était tout simplement d'assigner des utilisateurs à des serveurs non surchargés[12]. Cela est fait grâce au NAT qui va choisir automatiquement un serveur cible libre. Même si cette technique a le mérite d'être relativement simple à mettre en place, elle n'est pas particulièrement adaptée pour optimiser les échanges entre les joueurs en évitant les communications interserveurs (plus coûteuses que les intra-serveurs).
Le Load Balancing est souvent très étroitement lié au découpage du monde virtuel. Ce découpage peut être statique ou dynamique. Une seconde technique consiste à découper le monde en zone fixes assignés à des serveurs. Cette méthode est par exemple utilisée sur le jeu Albion Online[18]. Albion Online est un MMO acceptant plusieurs milliers de joueurs simultanés. Évidemment ces joueurs ne sont pas tous aux même endroits et ils sont donc assignés à des zones instanciées correspondant à des serveurs différents. Chaque zone est gérée par un serveur et chaque serveur peut gérer une ou plusieurs zones.
Cependant ce genre de technique peut être sensible aux effets de "Crowding" c'est à dire de rassemblement de la population à des endroits très localisés. Cela est visible dans les MMO notamment RPG où certaines villes/capitales vont être un carrefour et un point d'intérêt certain. Pour gérer ce genre de cas, Albion Online fait des villes des zones à part entières. Cela ne règle pas vraiment le problème si trop de joueurs souhaitent aller dans la même ville Le but est probablement de simplifier le Load Balancing pour le rendre vraiment production-ready. Malgré l'apparente simplicité de la solution, elle semble être efficace puisque Albion Online est un jeu en production qui accueille des milliers de joueurs sur ses serveurs.
Une amélioration de cette technique est de découper les zones virtuelles ,non plus en région, mais en petites cellules[21]. Ces cellules de tailles beaucoup plus réduites sont, elles, assemblées dynamiquement pour créer des régions. Les cellules vont être chargées de donner des informations sur la charge qu'elle subisse en temps réel et on va avoir un Load Balancer qui va utiliser ces données pour regrouper les cellules en régions (dynamiquement, donc) et assigner les régions à des serveurs. Le Load Balancer peut ensuite adapter en temps réel les régions en assignant des cellules particulièrement sensibles à la charge à des serveurs plus "tranquilles". Le regroupement des cellules est souvent fait avec des algorithmes génétiques. Une des faiblesses de cette technique est qu'elle est adaptée pour créer une première fois des zones, mais le changement en temps réel pourrait poser des problèmes. Bien optimisée, elle a l'avantage de supporter même les effets de Crowding.
Pour optimiser ces changements de charge en temps réel, des projets proposent d'utiliser des algorithmes de partitionnement de l'espace. Certains utilisent ainsi des KD-Tree[4, 11], ou d'autres arbres de partitionnements[22]. L'avantage était que même si la majorité des MMO(RPG) sont en 3D, gérer le partitionnement en 2D est suffisant. Nous pouvons donc utiliser facilement des KD-Tree avec k=2. Chaque noeud du KD-Tree sera une région du monde et va stocker les coordonnées de découpage de l'espace le concernant. Les noeuds enregistrent aussi des valeurs pour le rééquilibrage de l'arbre : la capacité totale du noeud et sa charge actuelle. La capacité et la charge d'un noeud (non feuille) sont l'addition des valeurs correspondantes des enfants. Les deux noeuds enfants d'un noeud correspondront à la zone qui a des valeurs plus importantes que les coordonnées de découpe et a celle qui a des valeurs moins importantes. Les feuilles de notre arbre vont elles contenir la liste des joueurs présents dans leur région, mais pas de coordonnées de découpe. Chaque feuille est associée à un serveur particulier. Lors d'une surcharge du serveur, un noeud fait appel au Load Balancer pour recalculer les coordonnées de découpe du noeud pour réduire sa zone.
Il y a donc de nombreuses techniques pour gérer la charge d'un jeu multijoueur, et bien souvent cette gestion de charge passe également par une optimisation interne de la communication client serveur : l'Interest Management.

2.2 Interest Management

L'Interest Management est la gestion de la zone d'intérêt d'un joueur. Lors que nous incarnons un avatar qui parcourt les étendues virtuelles d'un monde ouvert, nous ne pouvons pas tout voir. Pour cette raison, seules les actions que l'on peut voir ou qui nous concernent (messages, éléments non diégétiques...) nous intéressent. Il est inintéressant pour un joueur de savoir qu'un autre joueur a tué un monstre à l'autre bout de la carte, et cela augmente l'utilisation de bande passante pour rien. Pour cette raison nous souhaitons créer autour de notre joueur une zone dans laquelle tout ce qui s'y passe nous est transmis mais rien de plus.
Cette gestion de zone d'intérêt n'est pas seulement géographique : certaines actions ne doivent pas nous être transmises du tout même si elles se passent dans notre zone. Cela touche surtout à la triche. Par exemple, si lors d'un combat entre deux joueurs, un des deux utilise un sort pour se rendre invisible, il faut qu'il soit **vraiment** invisible même au niveau du réseau ! Si ce n'est pas le cas, des personnes mal intentionnées qui modifierait le client pourrait afficher le joueur invisible.
Lors de l'étude des échanges entre les clients du RTS 0A.D. (clone d'age of Empires), il est intéressant de remarquer que, quel que que soit notre client, nous recevons l'intégralité des messages envoyés dans la partie. Comme la partie est gérée par un des joueurs c'est normal que l'un d'eux reçoive toutes les informations (puisqu'il est l'équivalent du serveur) mais là, c'est le cas pour l'intégralité des participants. Le problème est que comme le seul rempart contre la triche est de ne pas afficher les informations reçues, si quelqu'un souhaite modifier un client, il pourrait afficher en temps réel toutes les actions adverses alors qu'elles sont censées se passer dans le brouillard de guerre. Pour un jeu comme 0A.D. où les personnes se connaissent et ne vont probablement pas tricher cela ne pose pas vraiment de problèmes, mais si c'était le cas sur des jeux compétitifs comme OverWatch, cela pourrait avoir des répercussions financières.
Pour gérer cette zone d'intérêt, plusieurs possibilités s'offrent à nous. Nous pouvons juste déterminer un cercle autour du joueur[5]

et déterminer au niveau du serveur si ce dernier à le droit de recevoir une information. La technique est relativement simple et précise, mais nous oblige à itérer régulièrement sur les objets proches du joueur.
Bien souvent la gestion de la zone d'intérêt est liée à un système d'événements et de publish-subscribe. Dans le cas d'Albion Online cela a de l'importance car pour leur Interest Management, ils découpent leur carte en petites cellules (un peu comme pour le load balancing dans certains projets) et les différents objets vont venir s'abonner aux cellules environnantes. Ainsi la zone d'intérêt du joueur est dynamique mais basée sur des cellules déterminées statiquement. L'avantage de cette technique est que les cellules, qui contiennent une liste des joueurs associés, ne sont que des hashmaps. Ainsi lorsqu'un joueur va entrer dans une cellule, la cellule va prévenir tous les intéressés qu'il y a un nouvel objet dans les parages. Les différents autres clients pourront alors créer un avatar pour ce joueur par exemple. Il y a une petite subtilité tout de même puisque dans Albion Online, il y a deux zones d'intérêt : une plus interne et l'autre externe. La zone interne est la zone d'intérêt telle qu'on la conçoit, la seconde sert à se désabonner des objets qui en sorte. Le faite d'avoir les deux permet d'éviter qu'un objet à la lisière des cellules n'entre et ne sorte en permanence de notre zone d'intérêt, ce qui générerait beaucoup de trafic réseau. L'intérêt d'utiliser ces cellules est que cela revient à simplement abonner les clients aux événements des cellules.
Nous pouvons remarquer que l'Interest Management se rapproche beaucoup du Load Balancing mais concerne plutôt l'optimisation des échanges plus que l'optimisation de la charge. Mais malgré toute la bonne volonté du monde même en optimisant au maximum les échanges, il y aura toujours des messages qui n'arriveront pas à temps chez les participants. Il faut donc s'assurer de compenser les désynchronisations entre ces derniers.

2.3 Synchronisation

La vue qu'un joueur à de sa partie dépend de beaucoup de choses. Dans tous les cas si quelque chose n'est pas normal cela risque de frustrer le joueur : S'il est mort alors qu'il n'a pas eu le temps de voir son adversaire par exemple. Pour gérer ce genre de cas, de nombreuses techniques existent mais bien souvent elles ne viennent que changer la vision du joueur pour lui *donner l' impression* que le monde virtuel qui l'entoure est consistant.
Par exemple pour estimer des positions sans avoir à envoyer des informations tout le temps, on peut utiliser des techniques de prédictions comme le Dead Reckonning. Cela consiste tout simplement à calculer une position à partir d'anciennes. Notez que cela n'est qu'une **estimation**, si nous perdons quelques paquets concernant des informations de positions, et que nous extrapolons de futures positions, rien ne nous garantit que la position est juste, mais cela va donner une *impression* de normalité. Alors que des téléportations intempestives peuvent vraiment casser l'immersion. La question est donc de savoir à quel point un client peut être désynchronisé.
Dans le cas de jeu où il faut une grande précision (être sûr que si l'on voit un joueur, il soit vraiment à cet endroit) il est possible d'utiliser des techniques de commandes différées comme le "Bucket Synchronisation"[3] ou le "Local Lag" [13]. Ces techniques consistent à ajouter un délai aux actions des joueurs pour s'assurer que tous les autres aient bien les ait avant de les appliquer.
La technique de "Bucket Synchronisation" est celle utilisée dans la majorité des RTS de age of Empires à Starcraft 2. L'idée est que le joueur envoie seulement des actions durant un laps de temps, le serveur attend que **tous** les joueurs aient envoyé une action, puis elles sont validées par le serveur et envoyées à tous les clients. Grâce à cela on est sûr que les clients restent bien synchronisés et cette technique réduit énormément le trafic réseau puisque seules les demandes d'actions sont envoyées. Le revers de la médaille est que la latence générale ressentie est celle du joueur avec la pire latence, et il est régulier de ressentir des ralentissements lors des parties. Pour cette raison cette technique est impensable pour des MMOs. Les jeux les plus récents trompent les joueurs en lançant seulement sur l'écran du client source des animations ou un son de feedback. Ce simple retour suffit la majorité du temps à satisfaire le joueur qui n'a pas l'impression de jouer dans le vide.
Le local-lag ressemble à la technique du Bucket Synchronisation, mais sans attendre les autres joueurs. L'idée est que plutôt que d'attendre que tous les joueurs aient donné des signes de vie, on va différer l'ensemble des actions des joueurs d'une ou deux centaine de millisecondes. L'idéal étant que ce délai soit assez bas pour être peu perceptible et facilement compensable mais assez haut pour que tous les clients aient eu le temps de recevoir l'action.
Toute la magie de la programmation réseau vient donc non pas à s'assurer que les joueurs voient le jeu "réel" mais plutôt à donner l'impression que ce qu'ils voient est réel. Si la latence ressentie dans un RTS est "justifié" par le fait que les troupes soient humaines et mettent du temps à réagir, cela semble normal alors qu'un joueur qui se téléporte en permanence ne le sera pas.
Finalement toutes ces techniques de compensation de latence reposent énormément sur un ressenti du temps. Il est toujours question de calculer une valeur en fonction de son ancien état ou de prévoir complètement un nouvel état. Il est donc important de bien garder la trace chronologique de ces états, et pour cela nous pouvons utiliser des Timelines[19]. Le principe des Timelines est très simple, toutes les données qui doivent être synchronisées sont stockées sous une forme chronologique : on sauvegarde toutes les données avec un temps associé. En mettant le temps au centre de la synchronisation on s'épargne beaucoup de problèmes d'estimation. Le serveur possédera donc des Timelines pour des données et transmettra les changements aux différents clients qui auront leur copie locale. Par exemple, si nous souhaitons gérer la position d'un objet, il suffit de créer une Timeline sur le serveur, et sur les clients concernés. Lors du choix d'une destination il nous suffit d'enregistrer la position actuelle de l'objet comme étant celle au temps actuel et d'enregistrer la position de destination comme étant celle à laquelle le joueur sera dans un temps donné. Si la direction change en cours de route, il suffit de réenregistrer la position actuelle du joueur à l'instant présent et la nouvelle destination vient remplacer l'ancienne. Cela permet de favoriser l'interpolation (moins sujette à désynchronisation) à l'extrapolation. La force de ces Timelines réside dans le fait qu'avec ces dernières il suffit de faire des getters/setters des valeurs à un instant t et les fonctions d'interpolation et d'extrapolation qui les utilisent. Ainsi si l'on ajoute un élément à notre Timeline, la prochaine interpolation récupérera ce dernier et donnera la bonne nouvelle valeur.

3 CONTRIBUTION

Afin d'appliquer les différents points vus précédemment, nous allons proposer une possibilité d'architecture. Cette proposition se concentrera principalement sur l'implémentation de l'Interest Management et sur la technique de synchronisation le "local-lag" dans le cadre d'un vrai projet.

3.1 Projet et impact sur l'architecture

3.1.1 *Projet.*

Le jeu qui est produit est un RPG avec des composantes de rogue-like centralisé autour d'un unique serveur censé supporter plusieurs vingtaines de joueurs simultanément. Comme le load balancing n'est pas la priorité dans notre expérimentation, il est très intéressant de n'avoir qu'un serveur puisque cela nous oblige à vraiment optimiser les échanges.

La caméra est en vue du dessus comme dans les hack'n slash. Les déplacements se feront au clic de la souris avec la possibilité de garder le clic enfoncé pour changer rapidement d'orientation.

Le jeu se découpera en partie qui seront instanciées (comme dans League of Legend). Cela implique une gestion des parties :

- Création de partie de jeu par les joueurs
- Récupérer les parties existantes
- Rejoindre une partie
- etc...

Une partie peut accueillir autant de joueurs que souhaités, qui affronteront ensemble des vagues d'ennemis dont la force sera adaptée au nombre de participants.

Les joueurs pourront choisir une classe au début de chaque nouvelle partie. Lorsqu'ils vainquent des monstres, les joueurs gagnent de l'expérience leur permettant de développer un arbre de talents. Ils gagnent également des objets pour améliorer leur personnage et de l'or dépensable auprès d'un PNJ marchand.

Voici le déroulement simplifié d'une partie :

(1) Lors du début d'une partie, les joueurs choisissent une classe.
(2) Quand tout le monde a choisi, les joueurs sont téléportés sur une carte choisie aléatoirement avec des monstres.
   (a) Une fois les monstres vaincus, les joueurs peuvent ouvrir un coffre pour récupérer des objets pour améliorer leurs statistiques.
   (b) Tous les joueurs qui meurent sont éliminés et les coéquipiers doivent continuer l'aventure avec l'équipe restante.
(3) ils sont ensuite téléportés dans une salle où il n'y a pas d'ennemis mais un vendeur auquel ils peuvent acheter des objets.
(4) Enfin les joueurs sont de nouveau téléportés dans une salle avec une difficulté accrue.
(5) etc...

3.1.2 *Impact sur l'architecture.*

Comme le jeu est un RPG relativement lent (lancement de sorts), et que les joueurs sont en équipe, la "vitesse" du réseau n'a pas besoin d'être parfaite. Mais nous avons besoin d'assez de précisions pour ne pas donner de fausses informations aux clients.

Le Dead Reckoning peut poser problème dans le sens où il repose sur de l'interprétation du futur et peut donc se tromper. Dans notre projet nous ne souhaitons pas que les joueurs aient commencé un sort puis que ce dernier s'arrête au milieu car le joueur n'avait pas assez de mana. La technique du local-lag sera donc utilisée, couplée à un système de temporalisation des données (Timelines).

Avec une bonne gestion des échanges réseaux, les micros-lags que pourraient ressentir les joueurs devraient être prévisibles et compensables par des animations ou des feedbacks sonores. Si nous commençons à jouer un son lorsqu'un joueur lance un sort, ce dernier saura que son ordre a été envoyé alors qu'il ne sera effectif qu'après le délai du local-lag (entre 100ms et 200ms).

Il est donc indispensable de penser notre architecture en fonction et de s'assurer qu'aucun message superflu ne soit envoyé. Pour cela il nous faut bien séparer les différentes parties (de jeu) dans différentes parties de l'architecture afin d'éviter à tout prix d'avoir un *single point of failure*. Cela peut arriver si nous n'avons qu'une interface de communication côté serveur.

Comme nous avons besoin de minimiser au maximum les échanges, que le sentiment de lag n'est pas (ou peu) un problème (car controlé avec le local-lag) et que nous avons tout de même besoin de nous assurer que toutes les actions soient "précises"; le choix d'un système déterministe semble adéquat. Cela permettra de ne pas envoyer des états complets d'objets mais juste les changements ou les inputs. Il faut cependant faire attention au revers de la médaille qui est une désynchronisation plus facile si l'on gère mal les échanges. Mais cela sera compensé par l'utilisation du local-lag et surtout des Timelines qui permettront de corriger rapidement les erreurs.

Habituellement un système déterministe induit l'utilisation de lock-step pour le jeu en ligne : le fait que le serveur attende que tous les clients aient envoyé leurs messages avant de passer à la suite. Sauf que cela implique que la personne avec la moins bonne connexion impacte tous les autres. Cela n'est pas possible dans notre cas. Nous allons donc adapter ce déterminisme : nous n'allons plus attendre que tout le monde réponde, et nous allons préciser plus d'informations que les systèmes déterministes classiques mais sans préciser à chaque frame toutes les données. Par exemple lorsqu' un joueur se déplace, seul un message de changements de direction avec la position du changement sera envoyé. Nous laissons les différents clients interpoler eux-même la trajectoire du joueur. Il est important de noter que cette technique est adaptée au projet puisque le déplacement se fait au clic et qu'il n'y a pas besoin de réagir instantanément. Par exemple pour un FPS, cela ne serait pas adapté car il faut connaître la position **exacte** de tous les joueurs en temps réel pour que chaque vue de joueur soit "juste".

Notre jeu étant en temps réel, toute la communication en temps réel sera faite en UDP. Avec une particularité tout de même, nous devons être sûrs que les messages arrivent à destination (*reliable*) mais nous n'avons pas besoin qu'ils arrivent dans l'ordre (*not ordered*). En choisissant d' utiliser le local-lag nous assumons qu'une action est différée d'une centaine de ms : bien souvent un message arrivera donc avant d'être exécuté. Il n'est pas indispensable de savoir que le joueur perd X de Vie dans 50ms *avant* qu'un joueur lance un sort dans 100ms (tant que la première arrive bien à temps pour son exécution). Cela facilite donc grandement l'utilisation de l'UDP.

Nous pourrions utiliser le TCP, seulement ce dernier n'est pas du tout adapté au temps réel[7]. Comme le TCP est dit *reliable & ordered* il assure que tous les messages envoyés arrivent à destination **et** dans l'ordre. Sauf que si des paquets sont perdus, le TCP "bloque"

l'envoi des paquets suivants et ralentis considérablement la communication, créant des effets de goulot d'étranglement. Il est donc nécessaire de refaire un mini-protocol UDP qui s'assure que les messages arrivent bien à destination, peu importe leur ordre.
Le TCP sera tout de même utilisé pour tout ce qui touche à la gestion des connexions, des parties de jeu ou de l'échange de messages puisque dans ces cas, nous avons besoin d'un moyen sûr pour acheminer les messages (*ordered & reliable*). Utiliser le TCP et l'UDP apporte cet autre avantage qu'en gardant la connexion TCP ouverte durant une partie, les déconnexions sont très facile à déterminer. Il est ainsi facile de gérer ce cas sans avoir à faire nous-même l'envoi récurent de paquets UDP. Habituellement il est déconseillé d'utiliser les deux ensembles puisque les deux protocoles s'impactent mais, dans notre cas de projet d'expérimentation, cela ne pose pas trop de soucis. Ici nous privilégions donc la facilité sur les performances.
Au niveau de l'Interest Management, comme nous ne sommes pas sur un monde ouvert, l'intérêt d'en faire peut sembler moindre. Mais ce n'est pas le cas puisque les cartes sont assez grandes (une dizaine d'écran de joueur de hauteur et de largeur). Si nous voulons réussir à faire jouer plusieurs dizaines de joueurs sur une partie, impossible de se contenter d'envoyer toutes les informations à tout le monde. Étant donné que les cartes seront connues, le découpage en cellules des cartes est statique. Comme nous ne sommes pas sur un pur MMO, le découpage dynamique des cellules perd de son sens. Et même de grands MMO, tels qu'Albion Online, ont un découpage statique de leur monde[18]. Cela devrait être grandement suffisant pour notre cas.
Finalement pour s'assurer de bien découper notre architecture nous utilisons un système d'événements basé sur les deleguates (utilisation de la bibliothèque FastDeleguate [6])[1, 14]. Cela permet de garder nos différentes parties bien découpées.
Maintenant que nous avons expliqué nos choix techniques d'architecture nous allons vois plus précisément comment nous les avons implémenté.

3.2 Communication

Avant de commencer à parler d'architecture de serveur ou de client, nous allons voir comment nous avons géré la communication client-serveur. En effet la communication est primordiale dans un jeu multijoueur et demande un travail non négligeable souvent invisible en tant que tel.
le serveur étant en C++, il faut tout d'abord recréer les classes qui encapsulent les sockets proposées par la winsock api : TCPClient, UDPClient, TCPListener etc... l'idée étant de retrouver le style du C# mais version C++/winsock.
Une fois cela fait, il faut créer des interfaces de communications UDP et TCP qui sont en faite des thread lisant et écrivant en permanence sur les sockets.
Lorsque les interfaces de communications reçoivent des données, elles les stockent sous forme de tableau d'octets avant d'essayer de les désérialiser. Les données sont ainsi transformées en classes "Packet" qui contienent les informations d'un message qui est ensuite ajouté à une queue de message.
Cela nous amène à un autre point de la communication qui a été travaillé : la sérialisation. Pour cela nous avons une classe de base abstraite appelée "Packet". Celle ci contient des fonctions templates qui permettent de sérialiser les types primitifs sous forme de tableau d'octet grâce au bit shifting.
L'avantage d'utiliser le bit shifting est que cela nous évite de nous perdre avec l'endianness : on **sait** qu'utiliser les bitshifts revient à charger de la mémoire vers le processeur la donnée et que cela équivaut à transformer la donnée en big endian [15, 20]. Et comme nous savons que nous sérialisons en big endian, il est facile de désérialiser un paquet reçu.
La classe "Packet" va posséder des fonctions virtuelles pures "Serialize()" et "Deserialize()" et chaque type de message que l'on créer implémente ces fonctions car seul eux savent comment se sérialiser et désérialiser. Ainsi si nous avons un paquet contenant un int32 et un int8, lors de la sérialisation nous pourrions choisir d'ajouter au buffer d'octets d'abord l'int32 puis l'int8 et comme nous avons écrit la fonction, dans la foulée nous ferions la fonction de désérialisation en sachant que la première donnée est l'int32. Pour savoir à quel type de paquet nous avons affaire, nous avons créé un simple petit header : tous les paquets commencent par un identifiant de type de paquet (pour pouvoir les désérialiser), un identifiant unique et la taille totale du paquet (y compris le header) en octet.
Lorsqu'une interface de communication reçoit des données sur sa socket, elle désérialise le header, puis est capable de désérialiser le paquet en fonction de son type. À l'issue de ça, l'interface de communication ajoute le paquet à une queue d'événement.
Le management d'événements est la pierre angulaire de la communication dans le projet. Il est est partagé entre deux classes : l'"EventManagerHandler" et l' "EventManager". Comme nous avons plusieurs parties de jeu qui peuvent être simultanées, il n'est pas souhaitable que les messages des différentes parties se mélangent. Nous avons donc un "EventManager" par partie de jeu (et un supplémentaire pour le broadcast). L'"EventManagerHandler" est un simple singleton permettant d'accéder à l'"EventManager" de notre partie. La gestion des événements est relativement simple en elle-même : chaque type de paquet est associé à une liste de deleguates. À intervalles réguliers, les paquets dans la queue d'événements sont récupérés et tous les deleguates associés aux types sont appelés. Il suffit donc, dans le constructeur d'une classe, de venir s'abonner à un type de messages pour les recevoir.
Finalement un des points les plus critiques à gérer est la communication UDP. Comme expliqué rapidement précédemment, nos choix techniques font que nous avons besoin d'utiliser l'UDP de façon *reliable* mais pas *ordered*. Pour cela il nous faut envoyer des messages UDP jusqu'à ce que l'on reçoive une réponse... comme en TCP. Enfin pas tout à fait, ici nous allons nous contenter d'envoyer notre message UDP toutes les 50ms et ne pas bloquer le reste des messages. Lorsque l'on reçoit un message UDP, comme le header contient l'identifiant du message, on envoie en retour un message de type ACK avec l'identifiant. Lorsque l'on reçoit l'ACK il nous suffit de supprimer le message UDP pour ne pas le renvoyer. Cela veut dire qu'il nous faut sauvegarder les messages et les ré-ajouter à la file des messages à envoyer régulièrement. De nos jours le taux de paquets perdus est inférieur à 1%[16]. Comme nous ne souhaitons pas valider les paquets de type ACK (auquel cas cela serait sans fin : il faudrait valider les ACKs des ACKs etc...) nous allons nous contenter d'envoyer dix fois notre ACK. Même si par malheur personne ne reçoit le message UDP (déjà très peu probable car moins d'1% de chance de drop), ce dernier sera renvoyé. Aussi étonnant

que cela puisse paraître cette technique fut utilisée sur des grosses productions[7] : c'était même "pire" puisqu'il n'y avait pas de système de validation des messages. Les jeux concernés se contentaient d'envoyer une dizaine de fois chaque message en espérant qu'un moins un arrive à destination. Si on y réfléchit, ce n'est pas si grave puisque les techniques de synchronisation (deadreckoning etc...) son là pour compenser ce genre de cas. Pour rappel, dans notre cas ce n'est pas possible puisque nous souhaitons avoir un système le plus déterministe possible et nous avons donc besoin de chacun des messages (même s'ils arrivent en retard). Il est intéressant de remarquer que la technique utilisée, même si elle emprunte un peu au TCP reste beaucoup plus légère que le dernier.

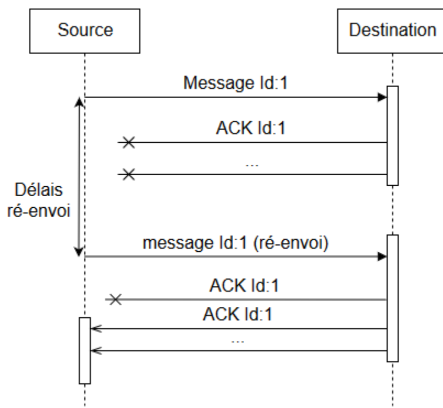

Figure 1: Communication UDP

3.3 Serveur

Voyons dans un premier temps comment nous avons créé notre serveur.
Notre serveur est découpable en deux : Le WorldServer et le GameServer. Le WorldServer, va gérer toutes les communications qui n'ont rien à voir avec la logique interne du jeu. C'est notamment elle qui va accepter les nouvelles connexions de joueurs, échanger les informations sur les parties en cours et créer ces dernières.
Conceptuellement cela se traduit par la création d'une classe qui est en lien avec des interfaces de communication TCP et UDP (chaque client associé possède une interface UDP et une TCP)
Le WorldServer est un thread qui se charge donc essentiellement d'accepter les nouvelles connexion. Lorsqu'il reçoit une demande de connexion TCP, il créé immédiatement une interface de communication avec la socket TCP et créé dans la foulée une autres avec une socket UDP. Un message TCP est envoyé au client avec les informations pour échanger en UDP (notamment le nouveau port).
L'autre objectif du WorldServer concerne tout ce qui touche à la gestion du lobby : récupérations des parties existantes, création de nouvelles, suppression de celles qui n'ont plus de joueurs etc... lors de la création d'une nouvelle partie, le WorldServer va créer un GameServer pour la durée de cette dernière et va passer les informations de communication avec le client au GameServer. Cela veut dire qu'à partir de ce moment le client ne communique plus qu'avec le GameServer ! C'est important de le noter puisque cela permet de décharger le WorldServer d'une part de la charge d'utilisateur. Ainsi, même si nous n'avons pas de Load Balancing poussé au niveau physique (plusieurs serveurs physiques), nous pouvons imaginer une amélioration du projet dans le futur où les serveurs (qui ne sont actuellement que des threads) soient des processus tournant sur différentes machines.
L'autre point important de notre serveur est donc le GameServer.

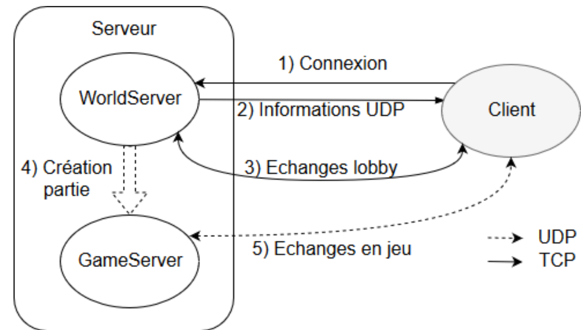

Figure 2: Echanges client-serveur

Ce dernier est composé d'au minimum deux autres parties : Le GameNetworkServer et le GameLogicServer.
Le premier est la partie qui va gérer les échanges avec les jours in-game, mais surtout va gérer l'Interest Management et filtrer les messages. Le second va encapsuler la logique même du jeu : vérifier et valider les requêtes des clients, gérer le calcul des collisions ou des dégâts etc... Lorsqu'une interface de communication lit des informations, elle ajoute le paquet à une queue d'événements. Le GameLogicServer est abonné à tous les événements qui concerne la logique : par exemple la requête d'un joueur de se déplacer. Une fois la requête traitée, la logique créée un paquet de type "Telle action s'est passée dans telle cellule". Le GameNetworkServer est abonné à ces paquets qui seront filtrés (Interest Management), transformés et renvoyés aux clients concernés.
Au sein de la logique, tout est géré avec un pattern Component très simplifié. Le GameLogicServer possède une hashmap de GameObject associées aux cellules. Ces GameObjects représentent tous les objets du jeu et ont eux même une liste de composants. Ces composants sont très importants puisqu'ils renferment toute la logique des objets. Par exemple nous allons avoir un composant pour gérer la vie d'un joueur ou sa position (par la suite nous verrons que pour les données nous utilisons un système de Timeline intégré aux composants).

3.4 Client

Au niveau du client, la subtilité principale est notre découpage. Pour ne pas avoir à faire de moteur de rendu, le moteur utilisé est Unity. Seulement, nous souhaitons pouvoir débugger facilement et potentiellement émuler de nombreux clients pour faire des tests de charges. Il ne serait pas optimisé de lancer X fois le jeu complet. Pour régler cela, la logique liée au client (envoi de messages au serveur, interpolation de valeur...) sera faite en C# sous forme de DLL. Ensuite il suffit de "plugger" un client de rendu qui charge la DLL. Cela nous permet d'avoir, certes, un client dont le rendu

est géré par Unity, mais également de faire facilement un client en console qui permettra de débugger ou tester facilement. Un tel projet nécessite donc un découpage complet entre la logique et la vue du client.
Grâce à l'Interest Management et aux techniques de gestion de la synchronisation, un client recevra essentiellement trois types de messages :

- ObjectEnter : qui va avertir de la création ou de l'apparition d'un objet dans la zone d'intérêt. Cela peut être un joueur, un NPC, un sort, un item...
- ObjectLeave : qui va avertir un client de la disparition d'un objet
- ObjectUpdate : qui va avertir un client de la mise à jour d'un objet, et bien souvent d'un composant en particulier. Par exemple si un joueur change de direction, il faudra mettre à jour les composants gérant la position.

Le client C# possède deux interfaces de connexions UDP et TCP ainsi que la liste des GameObjects clients. Il est également aussi en lien avec une Factory d'objets qui lui permet de créer des GameObjects à partir des informations contenues dans les messages "ObjectEnter". Ces objets clients sont donc des copies de ceux sur le serveur et contiennent seulement les informations à échanger avec le client de rendu.
Comme nous ne souhaitons pas faire confiance pour la logique (clientau rendu, le client C# (ayant son propre EventManager) va créer un message à destination de la vue. Le moteur de rendu ne sera abonnés qu'aux messages venant de la logique du client et sera chargé de gérer l'affichage des modèles, la gestion des animations, etc... Tout ce qui ne touche pas à la logique. Le client de rendu va donc également posséder une Factory pour générer ses ViewObjects avec les spécificités particulières du moteur de rendu. Par exemple, pour client sous Unity, dans la Factory nous allons pouvoir ajouter des animators controller particuliers ou des renderers d'Unity (que seul ce dernier peut connaître)
Le moteur de rendu sera quand même chargé de récupérer les commandes envoyées par l'utilisateur. Dans notre cas, par exemple, si un joueur appuie sur la touche A, le moteur de rendu va envoyer un message au client C# pour le prévenir. Ce dernier pourrait interpréter cela comme étant une commande de lancer de sort et va envoyer un message au serveur pour prévenir que le client souhaite lancer le sort avec un identifiant. le serveur va récupérer cette demande et vérifier que le joueur concerné qu'il n'est pas mort entre temps ou qu'il avait les ressources nécessaires. Une fois la demande validée, le serveur va mettre à jour ses GameObjects et envoyer des notifications de mise à jour aux clients. Lorsce que le client de base C# récupère les misse à jours il va également prévenir le moteur de rendu des changements.
Il y a tout de même un cas où nous "cassons" intentionnellement ce découpage par événements et messages. Certains composants sujets à beaucoup d'interpolation (comme la position par exemple) ont besoin d'avoir un accès le plus direct possible aux composants logiques du client afin d'avoir toujours la dernière valeur. Par exemple pour la position, si nous nous reposons que sur la queue d'événements pour la gérer, nous prenons le risque de la surcharger. Or la position doit être mise à jour aussi vite que possible sur la vue. Si la queue prend du retard à cause d'une surcharge, une ancienne valeur de position n'a plus de sens. Pour décharger un peu la queue d'événements, il a été définis que certains composant peuvent aller chercher directement (en lecture seulement) des informations. Cela permet d'assurer l'équilibre avec des événements plus importants mais moins fréquents.

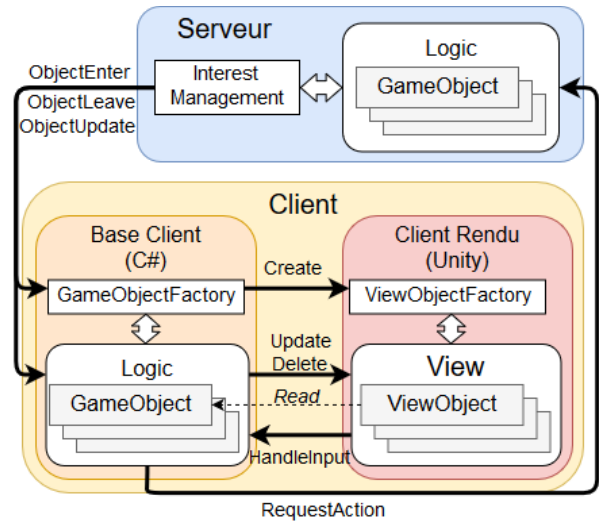

Figure 3: Séparation serveur-client-rendu

3.5 Interest Management

Un des points les plus importants dans l'optimisation des échanges réseaux concerne la gestion de la zone d'intérêt d'un joueur ou Interest Management.
Cette zone d'intérêt va impacter complètement la façon de concevoir notre projet. Comme vu précédemment, l'Interest Management repose sur la création d'une zone autour du joueur (basé sur un découpage statique ou non du monde virtuel) qui représente jusqu'où il peut percevoir le monde qui l'entoure.
Dans la solution, il a été choisi de découper les différentes cartes de façon statique en cellule de 10x10 unités (ou 1 unité correspond à 1 mètre dans le jeu). Lorsqu'un joueur se déplace dans le monde virtuel sa zone d'intérêt va donc être mise à jour en direct mais dans la limite des cellules. L'avantage de ces cellules statiques est qu'elles sont calculables sans prérequis et très rapide à déterminer en fonction de la position d'un joueur. Si on imagine une carte qui fait 40x40 unités est qu'un joueur est à la position (23, 12) nous savons qu'il est dans la troisième colonne et la seconde ligne, dans la cellule n°7 (voir l'exemple).
Cette facilité prend tout son sens dans la manière d'implémenter l' Interest Management dans le serveur. Sur le principe, il n'y a rien de sorcier, notre GameNetworkServer (qui s'occupe de la gestion de la zone d'intérêt) possède déjà une liste de clients avec lesquels il peut discuter. Il suffit donc d'ajouter une hashmap qui contient les identifiants des cellules et les joueurs dans ces dernières. L'avantage de pouvoir déterminer rapidement est visible dans la gestion de la logique du serveur. Cette dernière n'a rien à voir avec l'Interest Management à première vue, mais il lui revient de faire un premier filtrage. Lorsque le GameLogicServer traite une requête du client

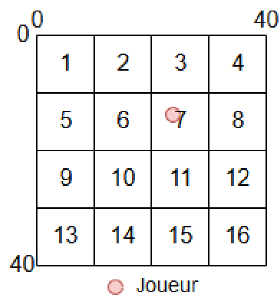

Figure 4: Exemple : détermination de la cellule à partir de la position

qui a des impacts directs, il va envoyer des paquets sous forme "ObjectEnter" "ObjectLeave" ou "ObjectUpdate" il suffit juste d'ajouter à ces paquets une nouvelle information : l' identifiant de la cellule dans laquelle l'action s'est déroulée.

Il suffit ensuite au GameNetworkServer de déterminer si la cellule dans laquelle s'est passée l'action est dans la zone d'intérêt de quelqu'un et d'envoyer les paquets aux clients qui en ont besoin.

Seulement actuellement nous n'avons que l'identifiant de la cellule dans laquelle il y a le client, et pas toute la zone d'intérêt. Pour gérer cela rapidement il va falloir indiquer aux cellules aux alentours qu'un client est intéressé par ses événements.

Pour déterminer cela nous allons avoir besoin de deux zones : une zone intérieure et une autre plus périphérique. Lorsqu'un objet entre dans la première, nous voulons que notre client soit prévenu par un paquet "ObjectEnter" afin de pouvoir préparer l'affichage de l'objet. On parle de préparation seulement puisque la première zone est censée englober toutes les cellules visibles par le frustrum du joueur. Un client commencera donc à avoir des informations sur un objet du jeu avant de le voir : cela évitera les cas où un objet apparaît de nulle part au milieu de l'écran d'un joueur à cause d'un ralentissement temporaire de sa connexion. La deuxième zone plus périphérique permet de se "désabonner" des événements d'un objet. Cette seconde zone plus large est différente de la première pour éviter d'avoir un élément à la limite d'une unique zone qui entre puis sort en boucle créant du de trafic pour rien. Cela a d'autant plus d'intérêt que même si notre système se veut relativement déterministe ((actor-based determinism)), comme ce n'est pas totalement le cas, la création d'un objet est coûteuse en trafic puisqu'il faut envoyer toutes les informations nécessaires à son bon fonctionnement (tous les composants avec les valeurs etc...). Il est donc important pour nous de réduire au maximum les causes de génération intempestives d'échanges peu utiles.

Pour gérer de façon optimale ces zones, il va donc falloir que le joueur s'abonne à chacune des cellules concernées. Cela est géré par l'Interest Manager (le GameNetworkServer) du serveur. Lors du déplacement des objets le GameLogicServer vérifie si l'objet à changé de cellule (grâce à sa position). Si c'est le cas la logique du jeu va envoyer un message de type "ObjectChangeCell" au GameNetworkClient. Dans ce message se trouvent des informations comme la cellule quittée et celle dans laquelle le joueur entre. De cette manière, La GameNetworkClient a juste à convertir ce message en "ObjectEnter" ou "ObjectLeave" pour chacun des clients concernés.

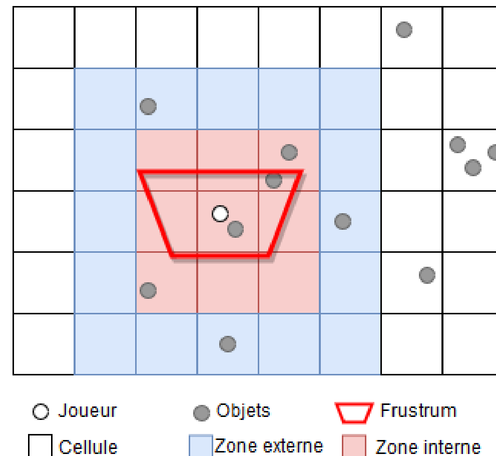

Figure 5: Gestion d'une zone d'intérêt avec des cellules

Si à ce moment, on se rend compte que l'objet qui a changé de cellule est un joueur, on va le désabonner des cellules ne faisant plus partie de sa zone d'intérêt et on va l'abonner à celle qui en fond partie. Pour que cela fonctionne, le joueur va également recevoir la liste des objets qui étaient présents dans la cellule quittée et la nouvelle pour pouvoir supprimer les premiers et créer les deuxièmes. C'est grâce à ça que nous pouvons gérer les "late join" des joueurs.

Le principe d'abonnement est assez simple, la hashmap contenant les cellules contient les identifiants des clients qui sont abonnés à la cellule. Lorsque l'on met à jour la zone d'intérêt, on se contente d'ajouter l'identifiant du client concerné à la cellule voulue. Ainsi un client apparaîtra dans 25 cellules simultanément (9 cellules de zone interne et 16 pour la zone externe). Quand une cellule doit transmettre un événement, elle itère sur tous ses clients abonnées et leur envoi le paquet associé.

Le dernier point à prendre en compte est la gestion des objets que l'on connaît déjà. Comme un joueur est abonné à l'ensemble des cellules autour de lui, si rien n'est fait il recevra un paquet pour à chaque fois qu'un objet entre dans une cellule de sa zone d'intérêt même si l'objet en question était **déjà** dedans. L'interest manager va donc garder une trace de tous les identifiants des objets étant déjà suivis. C'est aussi utile pour gérer les objets dans la zone extérieure d'intérêt : comme une cellule prévient tous les clients abonnés, elle ne sait pas si pour le client concerné elle est une cellule de la zone externe ou interne. Cela pose problème puisque si un objet vient tout juste d'entrer dans la zone externe, on ne veut pas que le client soit notifié de ses actions (ce n'est le cas que lorsqu'il entre dans la zone interne). Mais si l'objet entre dedans dans la zone alors qu'il était auparavant dans la zone interne, on continue de suivre ses actions jusqu'à ce qu'il quitte cette seconde zone. En gardant une trace des objets suivis, il suffit de savoir si un objet appartient à cette liste pour le client pour pouvoir optimiser l'envoi des messages.

Notre Interest Management est ainsi basé sur un abonnement des clients aux cellules environnantes. Ces dernières sont déterminées statiquement de façon à découper une carte en cellule de 10x10 unités. La partie logique du serveur aide un peu le GameNetworkServer, chargé de l'Interest Management, en indiquant à ce dernier

les cellules auxquelles se passent les actions. Il suffit ensuite d'itérer sur les clients abonnés aux cellules et de leur envoyer un paquet.

3.6 Synchronisation

Maintenant que nous avons vu comment l'optimisation de l'envoi de messages a été implémenté, regardons comment nous gérons la synchronisation et la compensation du lag.
Il a été choisi d'utiliser une technique d'ajout de latence en local (local-lag) pour compenser la latence générale. Et cela est fondamentalement lié à l' utilisation des *timelines* et des composants de notre architecture.
Concrètement les timelines sont des classes composantes contenant une hashmap de données avec un timestamp et des fonctions d' interpolation et d'extrapolation à un temps précis. Elles vont sauvegarder des données passées, présentes et futur pour simplifier les estimations. Dans l'implémentation actuelle nous avons un composant générique qui est ensuite étendu par des enfants. Par exemple il y a un composant PositionTimeLineComponent qui se contente de stocker un vecteur pour la position de l'objet. Grâce à ce système chronologique, il est très facile d'estimer les positions dans le temps. Quand un joueur change de direction (comme cela se fait au clic) nous connaissons la destination finale. il nous suffit d'enregistrer la position à laquelle le joueur souhaite être dans le futur et d'interpoler ensuite la position à un instant t.
Si le joueur souhaite changer de destination en cours de route, il envoie sa nouvelle destination au serveur. le serveur va ensuite estimer la position à laquelle le joueur est, enregistrer cette valeur dans sa propre TimeLine à l' instant "présent" (en comptant le délai local que l'on impose aux messages) et ajouter la destination finale dans le futur. Ainsi les clients vont recevoir ces deux informations.

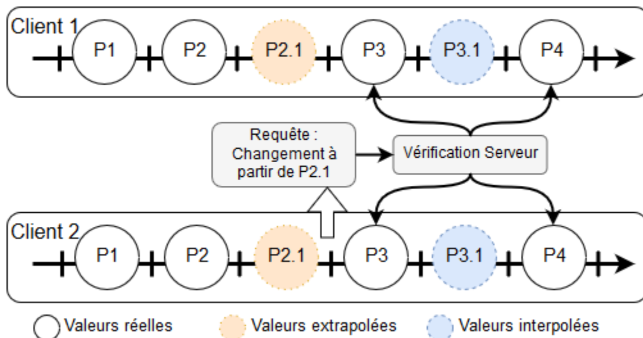

Figure 6: Fonctionnement de la timeline

Dans l'exemple précédent, la valeur P2.1 est une valeur extrapolée puisque aucune donnée n'est présente dans le futur. Après cela le Client 2 demande un changement de valeur : le serveur envoie valide une valeur "présente" (P3) et précise la valeur à atteindre dans le futur(P4). la valeur P3.1 est donc interpolée à partir de P3 et P4. Appliqué à la position d'un joueur, cela correspond à changer de direction : la valeur P3 est la position du joueur à partir de laquelle il se déplace et P4 la destination. P2.1 serait la même que P2 puisque dans notre cas si le joueur n'a plus de destination, c'est qu'il est arrêté.

Le principe est simple mais très efficace et il se marie très avec le local-lag. Comme nous avons un délai permanent pour laisser le temps aux clients de recevoir les messages, la plupart du temps les messages arrivent avant leur moment d'effet et il suffit de les ajouter à notre Timeline. Les changements sont immédiatement pris en compte si besoin par l'interpolation. Si le paquet contenant l'information arrive après son moment d'effet, il suffit d'ajouter les informations "dans le passé" et comme l'estimation sera une interpolation, la nouvelle bonne valeur sera aussitôt donnée.
Le seul problème de ce cas c'est que nous pourrions avoir une téléportation. Si un client reçoit un changement trop tard, la position sera corrigée, mais cela peut être brutal. Dans le cas de notre projet, nous souhaitons être assez précis, ainsi nous préférerons cette téléportation (statistiquement rare !) à des estimations. Mais si nous avions besoin d'avoir une transition sans téléportation dans le futur, il nous suffirait de prendre la position locale du client (chez ce dernier, et pas celle venant du serveur) d'un objet à la place de la position source donnée par le serveur. L'interpolation étant faite à partir de la position visible par le client, l'action sera invisible pour ce dernier. Cela a le désavantage d'être peu précis, et sujet à des désynchronisations. Que se passerait-il si la trajectoire cliente (différente de celle du serveur) est encombrée par un obstacle et pas celle du serveur ? Ou vice versa ? Dans ces cas nous retombons sur les problématiques du Dead Reckoning. Il faut donc faire attention et bien définir dès le début quel degré de désynchronisation nous (et surtout le joueur) pouvons accepter.
L'ajout d'un délai général avec ce système chronologique est très facile : il suffit d'ajouter les nouveaux éléments à la timeline du serveur avec le délai choisi. Si un joueur change de direction, le serveur va interpoler la position du joueur dans 'DELAI' ms (où 'DELAI' est le délai local défini) et valider cette position (s'ensuit après le déroulement expliqué auparavant). le client recevant des informations avec une date précise n'a pas besoin de connaître ce délai, tout ce qu'il sait c'est qu'il doit prendre en compte les données à partir d'un temps précis.

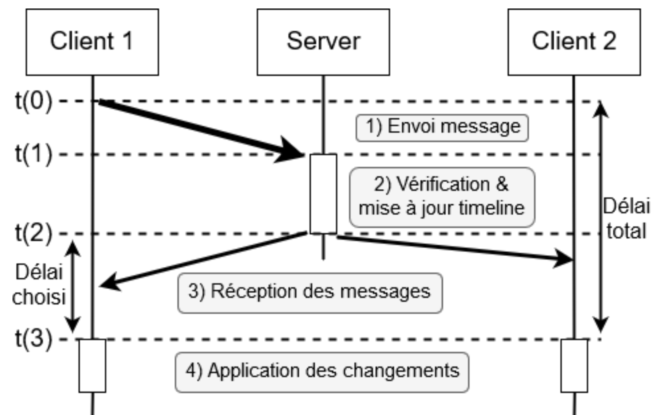

Figure 7: Fonctionnement du local lag

4 CONCLUSION

Dans ce document après avoir présenté rapidement les grandes questions du développement réseau, une architecture répondant à certaines des problématiques a été proposée. Cette dernière se concentrait principalement sur l'optimisation des échanges grâce à "l'Interest Management" et l'implémentation de la technique de compensation de latence "local-lag" couplée à l'utilisation d'un système de "Timelines".
Cette architecture est implémentée dans un projet de jeu de type RPG avec un unique client.
Il pourrait être bon de continuer à travailler l'optimisation des échanges qui est loin d'être parfait, mais surtout d'expérimenter sur le Load Balancing dans un contexte multiserveur.
Un autre point qui a été peu abordé et qui mériterait aussi un document complet pour lui seul, c'est la sécurité. Actuellement presque rien n'est fait au niveau sécurité, il faudrait travailler : la sécurisation des échanges UDP, la réduction des risques de menace de d'attaques DDOS, la complication d'attaque de type "Man in the Middle" et bien d'autres...
Le projet est accessible sur GitHub :

- Serveur : https://github.com/Suliac/ProjectDeusServer
- Client : https://github.com/Suliac/ProjectDeusClient

REMERCIEMENTS

Merci beaucoup à Vincent Desfontaines et à la société Stormancer pour leurs réponses et leurs indications.
Merci également à Vincent Stehly-Calisto, pour son aide et ses conseils.

REFERENCES

[1] Yossarian King Alexey Abramychev and Richard Harrison. 2016. Unite 2016 - Building Multiplayer Games with Unity. Retrieved August 12, 2018 from https://www.youtube.com/watch?v=-_0TtPY5LCc
[2] Marios Assiotis and Velin Tzanov. 2006. A distributed architecture for MMORPG. *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games - NetGames 06* (2006). https://doi.org/10.1145/1230040.1230067
[3] P. Bettner and M. Terrano. 2018. 1500 archers on a 28.8 : network programming in age of Empires and beyond.
[4] Carlos Eduardo Benevides Bezerra, João Luiz Dihl Comba, and Cláudio Fernando Resin Geyer. 2009. A Fine Granularity Load Balancing Technique for MMOG Servers Using a KD-Tree to Partition the Space. *2009 VIII Brazilian Symposium on Games and Digital Entertainment* (2009). https://doi.org/10.1109/sbgames.2009.11
[5] Ashwin R. Bharambe, Sanjay Rao, and Srinivasan Seshan. 2002. Mercury : A Scalable Publish-Subcribe System for Internet Games. *Proceedings of the 1st workshop on Network and system support for games - NETGAMES 02* (2002). https://doi.org/10.1145/566500.566501
[6] Don Clugston. 2005. Member Function Pointers and the Fastest Possible C++ Delegates. Retrieved August 13, 2018 from https://www.codeproject.com/Articles/7150/Member-Function-Pointers-and-the-Fastest-Possible
[7] Glenn Fiedler. [n. d.]. Gaffer On Games. https://gafferongames.com/
[8] Phil Kollar. 2014. World of Warcraft hits over 10 million subscribers as Warlords of Draenor launches. *Polygon* (2014). https://www.polygon.com/2014/11/19/7250737/world-of-warcraft-warlords-draenor-10-million-subscribers
[9] Phil Kollar. 2016. The past, present and future of League of Legends studio Riot Games. *Polygon* (2016). https://www.polygon.com/2016/9/13/12891656/the-past-present-and-future-of-league-of-legends-studio-riot-games
[10] Rynson W.h. Lau. 2010. Hybrid load balancing for online games. *Proceedings of the international conference on Multimedia - MM 10* (2010). https://doi.org/10.1145/1873951.1874194
[11] Jungyoul Lim, Jaeyong Chung, Jinryong Kim, and Kwanghyun Shim. 2006. A Dynamic Load Balancing for Massive Multiplayer Online Game Server. *Lecture Notes in Computer Science Entertainment Computing - ICEC 2006* (2006), 239–249. https://doi.org/10.1007/11872320_29
[12] Fengyun Lu, Simon Parkin, and Graham Morgan. 2006. Load Balancing for Massively Multiplayer Online Games. *NetGames'06* (2006). https://doi.org/10.1145/1230040.1230064
[13] M. Mauve, J. Vogel, and W. Effelsberg. 2004. Local-lag and timewarp : providing consistency in replicated continuous interactive media. *IEEE Transact Multimedia* (2004).
[14] Mike McShaffry. 2013. *Game coding complete.* Course Technology, Cengage Learning.
[15] Rob Pike. 2012. The byte order fallacy. Retrieved August 13, 2018 from https://commandcenter.blogspot.com/2012/04/byte-order-fallacy.html
[16] Internet Traffic Report. 2018. Internet Traffic Report. Retrieved August 13, 2018 from http://www.internettrafficreport.com/
[17] David Salz. 2016. Deterministic Simulation - What modern online games can learn from the Game Boy (GDCE 2016, Cologne). Retrieved August 12, 2018 from https://fr.slideshare.net/davidsalz54/salz-david-deterministicsimulationgdce2016
[18] David Salz. 2016. Unite Europe 2016 - Building a PvP focused MMO. Retrieved August 12, 2018 from https://www.youtube.com/watch?v=x_4Y2-B-THo
[19] Cheryl Savery and T. C. Nicholas Graham. 2012. Timelines: simplifying the programming of lag compensation for the next generation of networked games. *Multimedia Systems* 19, 3 (2012), 271–287. https://doi.org/10.1007/s00530-012-0271-3
[20] Sandeep Singh. 2011. Does bit-shift depend on endianness? Retrieved August 13, 2018 from https://stackoverflow.com/questions/7184789/does-bit-shift-depend-on-endianness
[21] Bart De Vleeschauwer, Bruno Van Den Bossche, Tom Verdickt, Filip De Turck, Bart Dhoert, and Piet Demeester. 2005. Dynamic Microcell Assignment for Massively Multiplayer Online Gaming. *NetGames'05* (2005).
[22] Shinya Yamamoto, Yoshihiro Murata, Keiichi Yasumoto, and Minoru Ito. 2005. A distributed event delivery method with load balancing for MMORPG. *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games - NetGames 05* (2005). https://doi.org/10.1145/1103599.1103610