Synchronisation par le réseau d'un monde virtuel sur Unity

Travail de Bachelor

Anthony Boscardin

Juillet 2018

Supervisé par :

Prof. Dr. Jacques Pasquier-Rocha
Software Engineering Group

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Groupe Génie Logiciel
Département d'Informatique
Université de Fribourg (Suisse)

Software Engineering Group

Remerciements

Merçi à mes parents pour tout leur soutien durant mes études. Un grand merci à ma soeur Emilie pour son aide précieuse et ses conseils.

Ensuite, j'aimerais remercier mes amis proches qui m'ont soutenu durant les bons et les moins bons moments de ma formation.

J'aimerais également remercier tous les professeurs et camarades de classe pour ces trois dernières années.

Sommaire

Comment synchroniser par le réseau et en temps réel un monde virtuel composé d'une centaine d'objets indépendants ?
Au travers de cette étude, il s'agira de préciser quelles sont les techniques de synchronisation qui existent et de proposer une implémentation concrète de deux versions ; l'interpolation de snapshots et la synchronisation des états. Durant ce travail, on mettra en valeur les enjeux et contraintes apportés par chacune d'entre elles mais également, on analysera leurs résultats, leurs performances et leurs améliorations possibles.

Mots-clés : Unity, Réseau, Physique, Virtuel

Table des matières

1. Introduction **2**

1.1. L'illustration de la thématique et les motivations 2
1.2. Organisation 2
1.3. Notations et Conventions 3

2. État de l'art **4**

2.1. Présentation du problème 4
2.2. Le monde virtuel 5
2.3. Intéractions 6
2.4. Contraintes 7

3. Design et Implémentations **9**

3.1. Aperçu 9
3.2. Choix du protocole 9
3.3. Synchronisation des contrôles 10
3.3.1. Encodage des contrôles 10
3.3.2. Décodage et API 11
3.4. Interpolation de snapshot 12
3.4.1. Tampon d'interpolation 12
3.4.2. Interpolation linéaire 13
3.4.3. Analyse de la performance 14
3.4.4. Améliorations possibles 15
3.5. Synchronisation des états 16
3.5.1. Accumulateur de priorité 17
3.5.2. Tampon contre la gigue 17
3.5.3. Analyse des résultats 19
3.5.4. Améliorations possibles 21
3.6. Autres techniques de synchronisation 22
3.6.1. Simulation déterministe 22
3.6.2. Comparaison des résultats 22

4. Conclusion **24**
4.1. Prise de recul 24
4.2. Derniers mots 24

A. Acronymes **25**

B. License of the Documentation **26**

Liste des figures

2.1. Le serveur et trois clients connectés 6
2.2. Force d'attraction émise par le cube 7
2.3. Force de répulsion émise par le cube 7

3.1. Détails d'un packet UDP 10
3.2. Remplissage du tampon d'interpolation 13
3.3. Interpolation linéaire 14
3.4. Tampon au temps t + 0ms 18
3.5. Tampon au temps t + 50ms 19
3.6. Tampon au temps t + 100ms 19
3.7. Interface de Clumsy 21

Liste des tableaux

3.1. Apperçu des coûts (Synchronisation des états) 20

Liste des codes source

3.1. NetworkInput.cs : keys enumeration 11
3.2. NetworkInput.cs : MoveForward input pressed 11
3.3. NetworkInput.cs : Decode keymap 11
3.4. NetworkInput.cs : Server API 12
3.5. SnapshotInterpolationManager.cs : Lerp calculation 14
3.6. IStateSynchronize.cs 17
3.7. StateSyncSmallCube.cs :CalculatePriority() methods 17

1 Introduction

1.1. L'illustration de la thématique et les motivations 2
1.2. Organisation 2
1.3. Notations et Conventions 3

1.1. L'illustration de la thématique et les motivations

Le but de ce travail est simple, en utilisant le moteur de jeu Unity[1], il va falloir synchroniser à l'aide du réseau un petit monde virtuel. Ce dernier devra être accessible par plusieurs clients différents et chacun d'entre eux devra être capable d'intéragir avec lui et de le modifier. Toutes les interactions et les modifications apportées à ce monde devront être gérées et repliquées de la manière la plus proche de la réalité possible sur les autres clients et en temps réel. Il faudra implémenter plusieurs techniques de synchronisation différentes afin de pourvoir comparer quels sont les avantages et désavantages de chacune d'entre elles et également analyser ce qui pourrait être amélioré.

Ce travail a été inspiré par les articles "Networked Physics" de Glenn Fiedler [7] et utilise la librairie réseau "SmartNet" de Kyle Douglas Olsen [9].

1.2. Organisation

Introduction

L'introduction contient les buts et la motivation de ce travail, ainsi qu'une petite explication de chaque chapitre.

Chapitre 1 : Présentation du problème

Ce chapitre présente les éléments qui composent le petit monde virtuel puis indroduit les enjeux et contraintes nécessaires pour avoir une synchronisation en temps réel. Il met également en évidence l'approche par laquelle les contraintes seront respectées.

[1] Moteur de jeu Unity : `https://unity3d.com/`

Chapitre 2 : Design et Implémentations

Ce chapitre explique en détail les processus executés chez le serveur et chez le client afin de pouvoir implémenter les deux stratégies de synchronisation choisies ; l'interpolation de snapshot et la synchronisation des états. Il décrit également les améliorations et les autres stratégies qui auraient pu être utilisées. Finalement, un petit comparatif des solutions sera présenté.

Chapitre 3 : Conclusion

Décrit l'idée originale et si le but a été atteint.

Appendix

Contient la liste des abréviations et des références utilisées tout au long de ce travail.

1.3. Notations et Conventions

— Conventions de formatage :

— Le code est formaté en suivant les "C# Coding Conventions"[2] :

```
1 public double division(int x, int y) {
2     double result;
3     result = x / y;
4     return result;
5 }
```

— Le travail est divisé en quatre chapitres qui sont structurés en section et sous-sections. Chaque section ou sous-section est organisée en paragraphes qui signalent les pauses logiques.

— Figure s, Tableau s and Listing s sont numérotés par chapitre. Par exemple, une référence à la Figure j du Chapitre i sera notée *Figure i.j*.

— En ce qui concerne le genre, je sélectionne systématiquement la forme masculine par raison de simplicité.

[2]Lien des conventions : `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/inside-a-program/coding-conventions`

2
État de l'art

2.1. Présentation du problème **4**
2.2. Le monde virtuel **5**
2.3. Intéractions **6**
2.4. Contraintes **7**

2.1. Présentation du problème

La première difficulté lorsque l'on souhaite mettre en réseau et en temps réel un jeu va être de cacher le délai de communication entre le serveur et le client. En effet, afin de proposer une simulation qui semble fluide et précise à l'utilisateur, ces actions doivent être executées immédiatement sur son écran et celles des autres joueurs doivent se refléter chez lui de manière réaliste.

Dans la majorité des cas, un utilisateur dispose uniquement du contrôle d'un nombre limité d'objets de la simulation, le reste des objets sont soit contrôlés par d'autres joueurs, soit uniquement par le serveur et la simulation. À cause du temps de latence avec le serveur, chaque joueur ne reçoit donc que les informations concernant les objets dont il n'a pas le contrôle dans le passé contrairement à la version correcte et au présent du serveur. Cacher cette différence chronologique entre nos actions et celles des autres dans la même simulation s'avère être un des principal défi à résoudre afin d'obtenir une simulation réaliste.

Il est également important de noter que dans la plupart des jeux, le serveur dispose de la responsabilité de trancher sur la validation des actions de chaque client. Par exemple si un joueur essaie de ramasser un objet et ce, au même moment qu'un autre joueur, seulement un seul des deux sera capable de le prendre. Il faut donc permettre à notre implémentation réseau de résoudre certains cas plus complexes comme celui-ci. Dans ce cas, il ne faut pas forcément immédiatement intéragir côté client mais attendre la réponse du serveur afin de savoir quel joueur a ramassé l'objet. Une solution afin de cacher le délai serait par exemple d'ajouter une animation afin de ramasser l'objet. L'animation prendra un peu de temps côté client et laissera du temps au serveur pour nous répondre qui a réelement ramassé l'objet.

La deuxième grosse difficulté d'une mise en réseau d'un jeu est la limitation des ressources réseau disponibles. En effet, il faut prendre en compte qu'un paquet réseau dispose d'une taille maximale et que multiplier le nombre de paquets veut également dire augmenter le nombre de paquets potentiellement perdus. Il s'agit de trouver une solution fiable qui minimise la taille et le nombre de paquets nécessaire à faire passer par le réseau.

La dernière difficulté est de gérer la triche. Dès qu'une application est connectée, il faut pouvoir gérer une personne qui ne suit pas les règles du jeu, ou les tricheurs. Certaines applications en ligne peuvent avoir des revenus économiques potentiels pour les tricheurs, mais aussi pour certains joueurs, l'excitation de la victoire est si forte qu'elle les poussent à des pratiques non fair-play. Il est important de mettre en place le maximum de barrières à ce genre de pratiques.

De nos jours, il existe une grande quantité d'applications et de jeux en réseau qui implémentent ces concepts. Mais étant donné qu'il existe une multitude de familles de jeux et que la plupart d'entre elles ne disposent pas des mêmes fonctionnalités, chaque implémentation de ces concepts réseaux restent différents d'un jeu à un autre. Il n'existe donc pas de librairie ou solution miracle lorsque l'on souhaite mettre en réseau une application.

Dans la littérature, il existe une grande quantité de ressources qui expliquent les divers problèmes liés à la mise en réseau, mais peu de ressources qui expliquent un processus d'implémentation concret et en détail basé sur certaines contraintes fixées.

2.2. Le monde virtuel

Le monde virtuel est composé de cent petits cubes noirs, initialement disposés en forme de carré sur le sol. Puis, à chaque nouveau client connecté, un cube blanc apparait à l'origine (centre du carré initial). Tous les cubes sont régies par la gravité et ils peuvent également interagir physiquement entre eux.

Sur la Figure 2.1, on peut voir le serveur (fenêtre en haut à gauche) et trois clients connectés, ainsi que leurs cubes blancs respectifs.

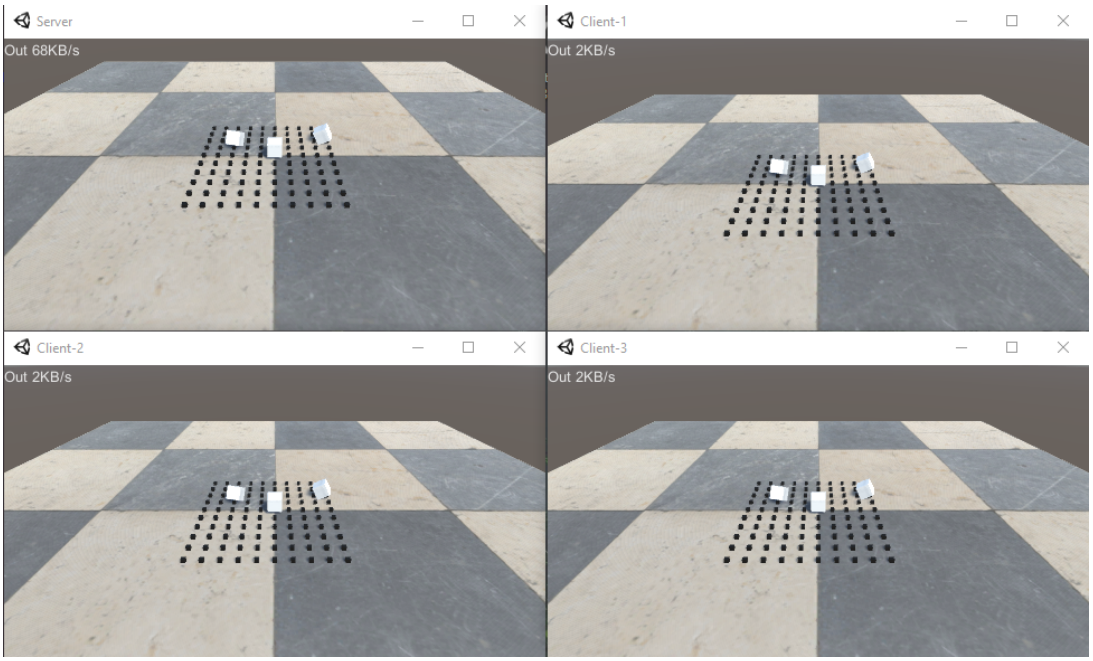

Figure 2.1. – Le serveur et trois clients connectés

2.3. Intéractions

Chaque client peut uniquement contrôler son cube blanc, il n'a aucun contrôle direct sur le reste du monde. Par contre, son cube blanc peut bel et bien intéragir avec tous les autres cubes et donc indirectement changer leurs positions, rotations et vitesses.

Le client peut contrôler son cube de trois manières différentes. La première avec les touches "WASD", cela applique une force linéaire au centre de gravité du cube, cette force est parallèle au sol, ce qui permet au cube de se déplacer en glissant et roulant dans chaque direction.

Deuxièmement, en pressant la touche "G", le cube se transforme en attracteur qui attire tous les cubes présents dans une petite sphère centrée sur le cube. Voir Figure 2.2

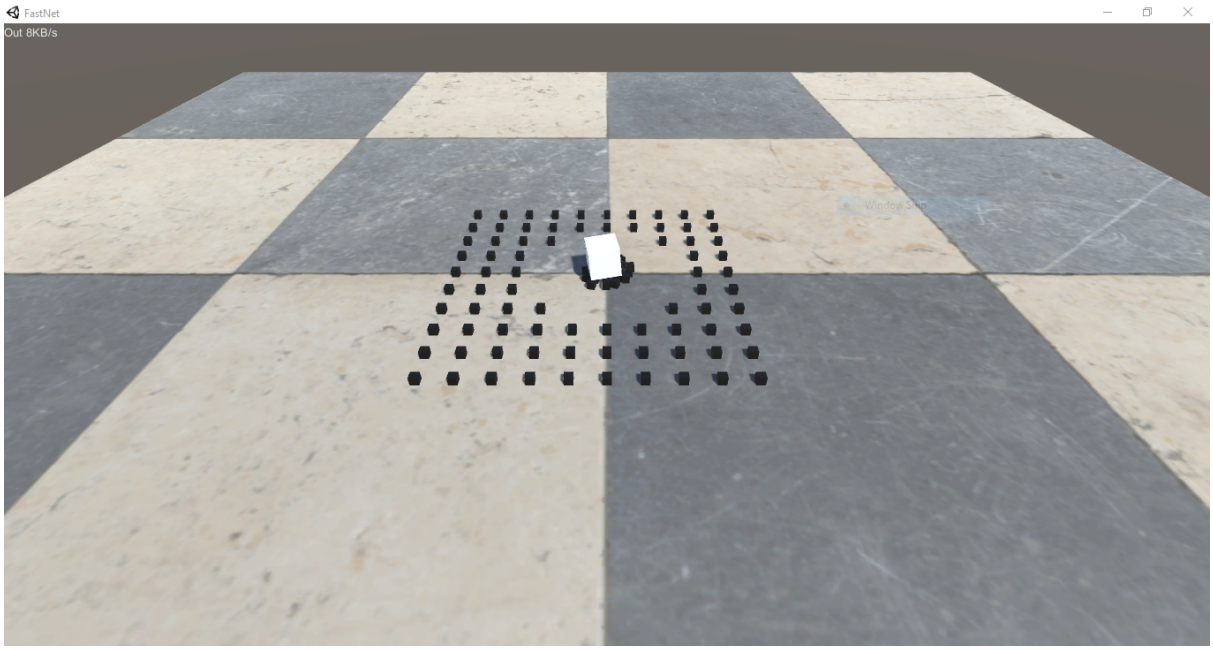

Figure 2.2. – Force d'attraction émise par le cube

Finalement, à la même manière que l'attraction, le client peut émettre une force de répulsion émanant de son cube en pressant la touche "H" Voir Figure 2.3

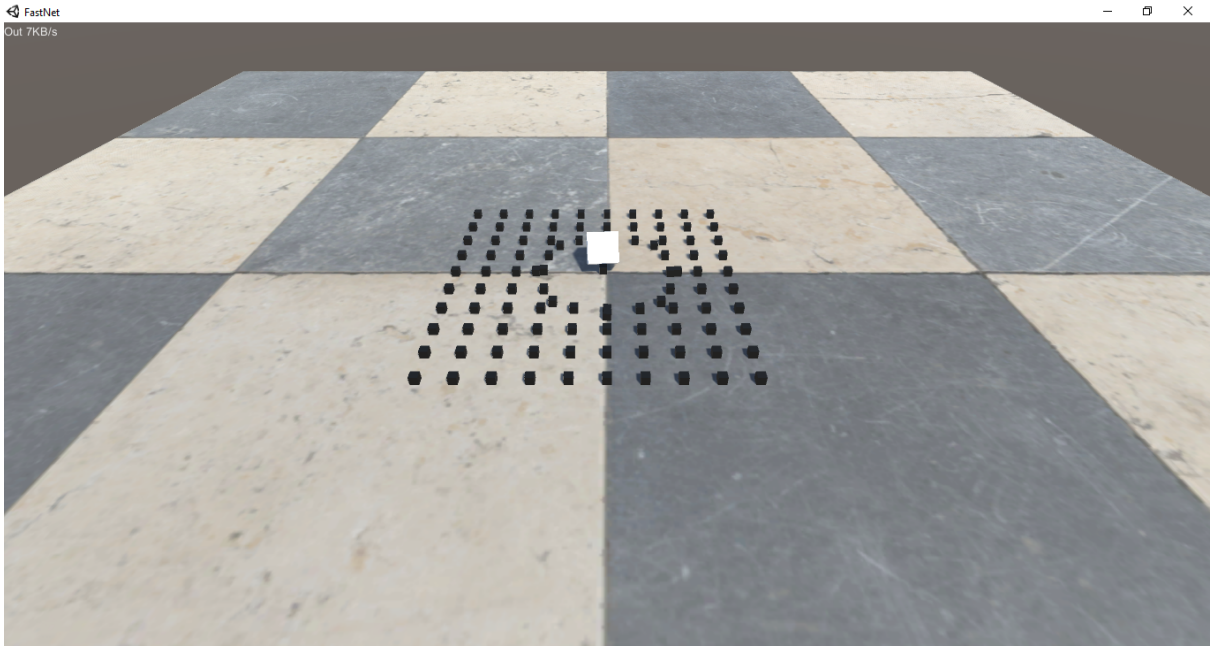

Figure 2.3. – Force de répulsion émise par le cube

2.4. Contraintes

Afin d'avoir une simulation correcte et d'éviter toute tentative de triche venant d'un client modifié, le serveur sera autoritaire. Cela signifie qu'il sera le seul responsable de

la simulation du monde et qu'il ne fera jamais confiance aux données transmises par un client. Grâce à cette contrainte, le seul et unique état correct de la simulation est celui du serveur, cela est donc ce dernier qui devra être repliqué de la facon la plus proche de la réalité sur chaque client.

Le client ne transmettra donc pas des positions et des rotations de son cube et des autres, mais uniquement les contrôles (les touches du clavier) qu'il utilise. Le serveur aura la responsabilité d'exécuter les contrôles de chaque client sur la simulation.

3
Design et Implémentations

3.1. Aperçu **9**
3.2. Choix du protocole **9**
3.3. Synchronisation des contrôles **10**
3.3.1. Encodage des contrôles 10
3.3.2. Décodage et API 11
3.4. Interpolation de snapshot **12**
3.4.1. Tampon d'interpolation 12
3.4.2. Interpolation linéaire 13
3.4.3. Analyse de la performance 14
3.4.4. Améliorations possibles 15
3.5. Synchronisation des états **16**
3.5.1. Accumulateur de priorité 17
3.5.2. Tampon contre la gigue 17
3.5.3. Analyse des résultats 19
3.5.4. Améliorations possibles 21
3.6. Autres techniques de synchronisation **22**
3.6.1. Simulation déterministe 22
3.6.2. Comparaison des résultats 22

3.1. Aperçu

Ce chapitre est présenté de manière chronologique identique par rapport au processus de réflexion et d'implémentation du code source. Cette présentation chronologique devrait aider à comprendre les décisions logiques prises à chaque moment durant la conception du code source et de la solution.

3.2. Choix du protocole

Avant même de pouvoir commencer à travailler sur une solution, il a fallu déterminer quel est le protocole réseau qui va être utilisé dans notre cas. Il existe deux protocoles ré-

seaux : le Transmission Control Protocol (TCP) et le User Datagram Protocol (UDP). La différence entre ces protocoles est que la connection TCP se fait à l'aide d'une connection bidirectionnelle et que pour chaque paquet envoyé, nous avons la garantie qu'ils seront reçus et également qu'ils le seront tous dans le bon ordre, au contraire du protocole UDP qui ne garantit ni la livraison des paquets, ni leur ordre. Par contre un de ses gros avantages et qu'il est bien plus rapide.

On a donc choisi d'utiliser le protocole UDP afin de profiter de sa rapidité. On a néanmoins dû s'occuper manuellement du cas des paquets reçus dans le mauvais ordre et des paquets perdus dans les solutions.

Une fois le protocole choisi, il reste un autre élément important à garder en tête. Il s'agit du Maximum Transmission Unit (MTU) qui est la taille maximale d'un paquet pouvant être transmis en une seule fois et ce, sans fragmentation par le réseau. D'après la littérature [10], pour les paquets UDP, la limite est de 576 bytes. Il ne faut pas oublier de prendre en compte que le header IPv4 prend 20 bytes, et que celui d'UDP prend 8 bytes. Ce qui nous laisse avec 548 bytes disponibles par paquet pour nos données. Il est important de noter que cette valeur est le strict minimum qu'un routeur doit être capable de transmettre sans fragmentation mais dans la réalité, de nombreux routeurs disposent d'un minimum plus élévé. Ce chiffre est très important à garder en tête et nous reviendrons donc dessus plus tard.

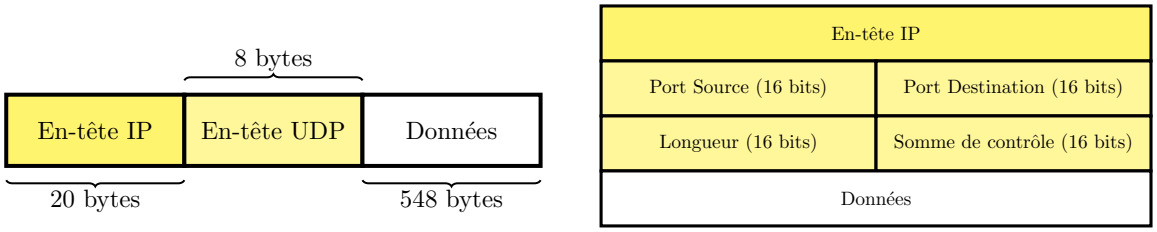

Figure 3.1. – Détails d'un packet UDP

3.3. Synchronisation des contrôles

Comme mentionné par les contraintes, nos clients doivent être capables de transmettrent leurs contrôles au serveur afin que ces derniers puissent être executés sur la simulation. La solution choisie doit non seulement être rapide et avoir un impact minimal sur la bande passante utilisée mais doit également pouvoir proposer une interface de programmation (API) simple d'utilisation côté serveur.

3.3.1. Encodage des contrôles

Afin de minimiser la taille du message transmis, la solution choisie est la suivante : on encode chaque contrôle à une position particulière sur un entier non-signé (uint). En effet, avec cette techique, il est possible d'encoder jusqu'à 32 contrôles différents sur seulement 4 bytes. Si le bit à la position x est 1, le contrôle a été pressé, sinon il est égal à 0. Il est

important de remarquer que le choix des 32 contrôles est totalement arbitraire. Ce chiffre est basé sur le fait que dans la plupart des jeux, il existe en général plus de 16 contrôles mais moins que 32.

Et ainsi, avec l'aide d'une simple énumération, on peut ainsi coder quel contrôle est assigné à quelle position.

```
public enum Keys : uint //Can have a maximum of 32 different inputs
{
    None = 0u, // 1
    MoveForward = 1u, // 2
    MoveBackward = 2u, // 3
    MoveLeft = 4u, // 4
    MoveRight = 8u, // 5
    MoveJump = 16u, // 6
    Use = 32u, // 7
    Push = 64u, // 8
    Pull = 128u // 9
}
```

Listing 3.1 – NetworkInput.cs: keys enumeration

Ensuite, si le contrôle a été activé, il suffit de faire un OU logique (OR) ce que va modifier notre uint en changeant le bit à la position souhaité par 1. Appelons désormais cet uint la "keymap".

```
public uint Keymap = 0;
if (Input.GetKey(KeyCode.W))
{
    Keymap |= (uint)Keys.MoveForward;
}
// Avant: 00000000 00000000 00000000 00000000
// Apres: 00000000 00000000 00000000 00000010
```

Listing 3.2 – NetworkInput.cs: MoveForward input pressed

A chaque nouvelle image, le client ajoute les contrôles qui ont été activés à sa keymap. Puis, toutes les 0.05 secondes (20 fois par seconde), il construit un paquet contenant le temps actuel et sa keymap et l'envoie au serveur. Il réinitialise ensuite sa keymap par zéro.

3.3.2. Décodage et API

A chaque réception d'un paquet contenant des contrôles, le serveur vérifie que le temps du paquet est supérieur au temps du dernier reçu, si c'est le cas, il sauve la nouvelle keymap comme étant la dernière reçue. Il garde également l'avant dernière keymap en mémoire. Le serveur dispose donc des deux dernières keymap de chaque client. Le décodage est assez similaire à l'encodage, il suffit de faire un ET logique (AND) entre la dernière keymap reçue et le contrôle qui nous intéresse. Si le résultat est 1, il est actif sinon il ne l'est pas.

```
private bool GetKeyValue(Keys key)
{
    return (_lastKeymap & (uint)key) != 0;
```

```
4 }
```

Listing 3.3 – NetworkInput.cs: Decode keymap

Le fait de garder les deux dernières keymap permet en plus de savoir quels contrôles sont actuellement pressés, de savoir si la touche vient d'être activée et si on vient de la relacher.

L'API côté serveur se compose d'une série de trois méthodes statiques, qui prennent en paramètres : la connection (qui definie de quel client il s'agit) et du contrôle que l'on souhaite vérifier.

```
1 public static bool GetKey(Connection connection, Keys key)
2 public static bool GetKeyDown(Connection connection, Keys key)
3 public static bool GetKeyUp(Connection connection, Keys key)
```

Listing 3.4 – NetworkInput.cs: Server API

3.4. Interpolation de snapshot

Tout d'abord, afin de pouvoir expliquer en quoi consiste cette interpolation de snapshot, il faut expliquer ce que l'on entend quand on parle de snapshot. Il s'agit d'une représentation exacte de la simulation à un temps précis. Elle contient toutes les données nécessaires afin de pouvoir reconstruire à l'identique la simulation à un temps donné. Dans notre cas, elle contient donc la position et la rotation de chaque petit cube et cube de client.

L'interpolation de snapshot peut être divisée en deux parties distinctes.

Dans la première partie, nous avons le serveur qui est responsable de la simulation et qui est également en charge de s'occuper de créer les snapshots et de les envoyer aux clients connectés. Le débit de capture et d'envoi est fixé à 10 snapshots par seconde.

Puis, dans la deuxième partie, nous avons les clients qui ont la responsabilité de reconstruire la simulation de la manière la plus proche de la réalité possible. Les clients eux ne connaissent strictement rien de la simulation et des règles de physiques établies. Afin de parvenir à cette tâche, ces derniers utilisent les snapshots reçus et une technique appelée l'interpolation linéaire.

3.4.1. Tampon d'interpolation

Chaque client dispose d'un tampon d'interpolation, ce dernier est composé d'un tableau qui permet de stocker les derniers snapshots reçus par le serveur et d'un pointeur qui indique la position du snapshot le plus vieux du tampon, voir Figure 3.2.

En effet, lorsque l'on parcourt le tampon, on commence toujours par l'élément à la position du pointeur et l'on continue jusqu'à avoir parcouru tous les éléments. Si on arrive au bout du tableau et que l'on a toujours pas parcouru entièrement le tableau, on continue la boucle en sautant au début du tableau. Lorsque l'on souhaite ajouter un nouvel élément au tampon, il suffit de l'ajouter à la place du pointeur et de déplacer le pointeur d'une position.

Cela permet à notre tampon de garder une structure très intéressente. Le tampon contient donc toujours une liste de snapshots par ordre croissant de leur date de création ; et donc le tampon dans son ensemble, répresente toujours des états de la simulation ordonnés suivant leur ordre chronologique.

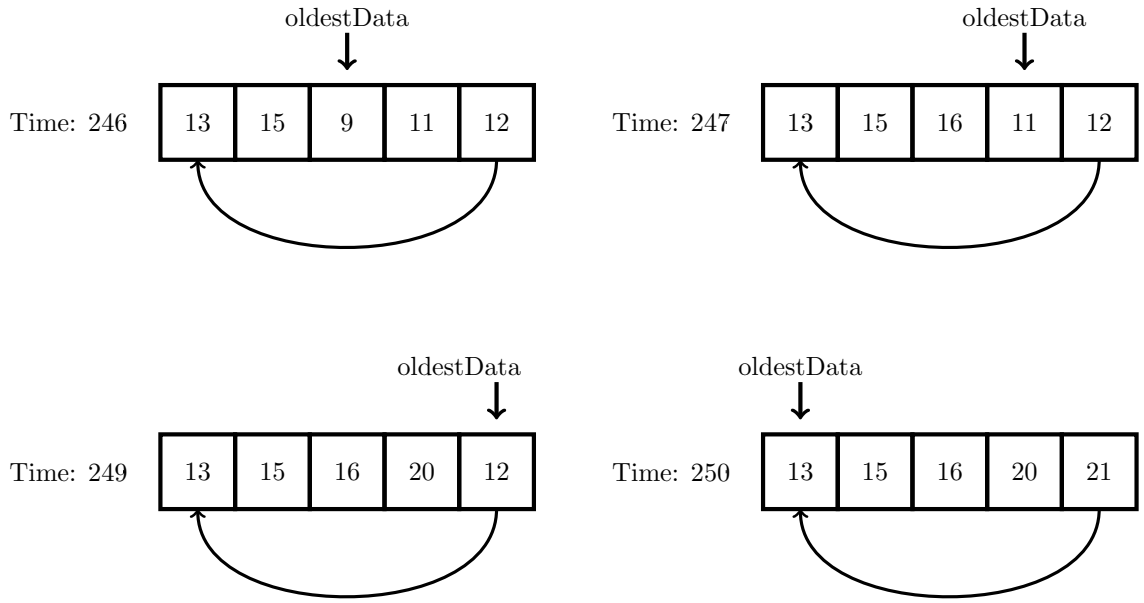

Figure 3.2. – Remplissage du tampon d'interpolation

Il est important de noter, que l'on ajoutera en aucun cas un snapshot dans le tampon si un snapshot plus récent en fait déjà parti. Il est donc possible lorsque l'on reçoit des paquets dans un mauvais ordre, d'avoir des petits sauts dans la liste des snapshots, ce n'est pas grave car l'interpolation linéaire permet de prendre en compte ce cas particulier.

3.4.2. Interpolation linéaire

Nous disposons maintenant d'un tampon rempli et ordoné de snapshots, mais on ne peut pas encore recréer la simulation. En effet, si l'on se contente simplement d'afficher chaque snapshot les uns après les autres, le résultat n'est pas fluide. On observe que tous les objets sautent de position en position. Cela est dû au fait que nous ne recevons, dans le meilleur des cas, seulement 10 snapshots par seconde et que nous ne simulons rien côté client. C'est ici qu'entre en jeu, l'interpolation linéaire. Cela nous permet d'approximer la position des objets entre chacun des snapshots reçus de la simulation.

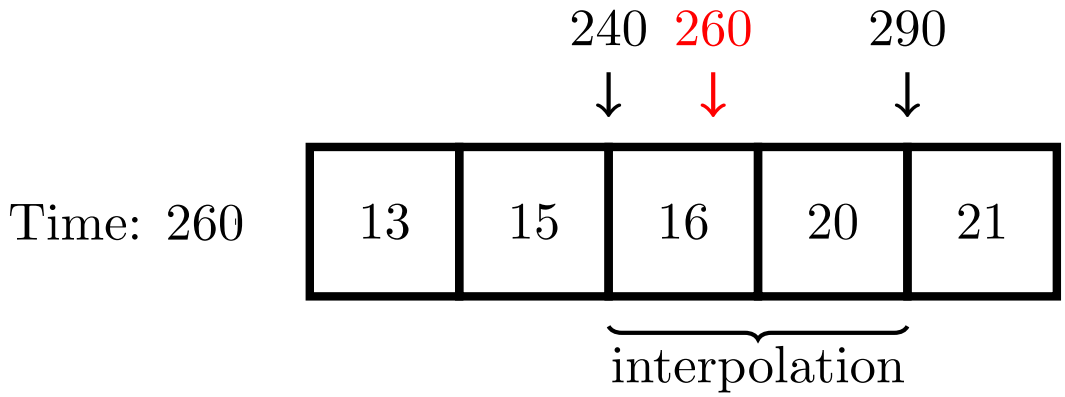

Figure 3.3. – Interpolation linéaire

Sur la Figure 3.3, on observe le cas suivant : le temps de la simulation est 260. Il nous faut donc trouver dans notre tampon quelles sont les deux snapshots qui entourent ce temps. Nous avons le snapshot n°16 qui référence la simulation au temps 240 et le snapshot n°20 qui représente la simulation au temps 290. Nous n'avons pas reçu d'informations concernant les snapshots {17,18,19}. Il faut donc pouvoir calculer à quel pourcentage entre les deux simulations nous nous trouvons.

```
1 float lerp = (renderTime - oldSnapshot.Time) / (newSnapsnot.Time - oldSnapshot.Time);
2 float lerp = (260 - 240) / (290 - 240) = 0.4;
```

Listing 3.5 – SnapshotInterpolationManager.cs: Lerp calculation

Dorénavant, nous disposons d'assez d'information pour recréer une simulation approximative, mais proche de la réalité. Il suffit de positionner chaque objet à 40% entre la position en n°16 et n°20. Il faut aussi changer sa rotation de la même manière.

Il est cependant également important d'ajouter, que le temps en rouge n'est pas le temps actuel mais le temps 100 millisecondes dans le passé. Ce petit décalage permet de s'assurer une grande probabilité que nous avons bien deux snapshots disponibles afin de faire l'interpolation.

3.4.3. Analyse de la performance

Analysons maintenant les statistiques de cette solution.

Pour chaque objet présent dans la simulation, il faut synchroniser sa position et sa rotation.

— La position de chaque objet est définie par un vecteur 3 qui est composé de trois nombres à virgule flottante (float). Étant donné que la représentation d'un float en C# nécessite 4 bytes, il nous faut donc envoyer 12 bytes par objet pour synchroniser la position.

— La rotation de chaque objet est, quand à elle, définie par un quarternion qui se compose de quatre float. Il nous faut donc transmettre 16 bytes par objet afin de synchroniser la rotation.

Pour résumer, chaque objet a donc besoin d'envoyer un total de 28 bytes *(12+18)* afin de pouvoir être repliqué correctement par le client. Comme défini arbitrairement au début, le nombre total de petits cubes est fixé à cent. On émet également l'hypothèse que l'on supporte au maximum quatre clients connectés. Il nous faut donc synchroniser 104 objets qui nécessitent chacun 28 bytes ; il nous faut donc transmettre 2'912 bytes par snapshot. À cela, il ne faut pas oublier d'ajouter l'en-tête nécessaire afin que l'interpolation linéaire marche. Il s'agit bien sur du numéro de séquence qui permet de remplir le tampon d'interpolation. Ce numéro est défini par un uint qui utilise 4 bytes. Finalement, nous avons donc besoin d'un total de 2'916 bytes. Ce chiffre est plus grand que les 548 bytes du MTU, cela signifie que le paquet n'est plus garanti d'être transmis non-fragmenté. Dans le pire des cas, il pourrait même être fragmenté en 6 paquets. La fragmentation n'est pas forcément dramatique mais elle amplifie le problème de la perte des paquets ; si un paquet est fragmenté en cinq morceaux et qu'un seul de ces morceaux est perdu, alors tout le paquet sera perdu.

Sur des tests, en réseau local, la taille assez importante des paquets ne semble pas poser de problème. Il est par contre bien plus dur de tester cela en conditions réelles, mais nous allons quand même essayer d'améliorer la situation.

3.4.4. Améliorations possibles

Il existe malheuresement peu d'améliorations possibles avec cette technique.

En effet, comme le client ne connait rien de la simulation physique et s'occupe uniquement d'afficher passivement la simulation, il est impossible de mettre en place un système d'extrapolation, comme nous le verrons avec l'autre solution. Il est également impossible d'augmenter le nombre d'objets simulés sans augmenter directement la bande passante utilisée. Chaque objet ajouté à la simulation devra être inclus dans le snapshot.

Il est par contre possible en compressant les vecteurs 3 et les quaternions de réduire la taille des paquets envoyés. Mais cette compression est forcément accompagnée d'une perte de précision de ces données. Cette compression, si elle est appliquée avec soin à notre simulation sera imperceptible à l'utilisateur final.

Compression des quaternions

Les quaternions suivent la propiété mathématique suivante :

$$x^2 + y^2 + z^2 + w^2 = 1$$

Il est donc possible d'en reconstruire un avec seulement trois de ces quatre composants. Si on envoie x, y et z, on peut retrouver par calcul w avec :

$$w = \sqrt{1 - x^2 - y^2 - z^2}$$

Avec les quaternions, si on change le signe de chacun des éléments tel que (x,y,z,w) devient (-x,-y,-z,-w), alors les deux quaternions représenteront exactement la même rotation. Cela permet de ne pas avoir à transmettre un bit de signe car il suffit simplement de se débrouiller à toujours transmettre le w positif, en inversant le quaternion si nécessaire. Afin de garder un maximum de précision, le choix de l'élément à reconstruire n'est pas forcément toujours w. Il faut choisir le plus grand élément (en valeur absolue) et encoder son index à l'aide de deux bits [0,3] (0=x, 1=y, 2=z, 3=w). On transmet aussi les trois éléments les plus petits avec et de l'autre côté, on peut reconstruire notre rotation à l'aide de l'index. La dernière étape consiste à réduire la précision des float des trois plus petits éléments en utilisant des short à la place des uint. En C#, cela permet de réduire de 4 bytes à 2 bytes la taille utilisée par chaque élément. Notre rotation une fois compressée utilise donc 6.25 bytes *(2*3 bytes et 2 bits)* à la place des 16 bytes, une amélioration de 60.94%.

Compression de la position

Afin de compresser les vecteurs 3 qui représentent la position des objets, il faut se rappeler que notre monde virtuel n'est pas infini. En effet, le sol ne fait que 32 mètres sur 32 ([0-32] en x et y) et nos objets ne peuvent pas traverser le sol et ne peuvent pas dépasser la hauteur maximale de 16 mètres ([0-16] en z). Nous pouvons donc utiliser ses bornes afin de créer un nombre à virgule flottante qui nécessite bien moins de bits que le float. Il nous reste encore à définir quel précision nous souhaitons avoir. Après expérimentation, nous avons trouvé qu'une précision de 512 valeurs par mètre (moins de deux milimètres) était largement suffisante. Cela nous donne donc [0,16383] valeurs possibles en x et y, et pour z [0-8192]. Cela requiert donc 13 bits pour z et 14 bits pour x et y. Ce qui nous donne un total de 27 bits à la place des 96 bits (12 bytes) sans compression, une amélioration de 71.88%.

À l'aide de ces deux compressions, nous avons réussi à réduire de 65.53% le poids du snapshot afin d'arriver à 1005 bytes ! Cela permet de nous rapprocher du MTU et d'éviter dans la majorité des cas la fragmentation, ou du moins a la réduire à deux paquets.

3.5. Synchronisation des états

Cette solution consiste à envoyer les données de position et de rotations des objets au client, mais également la vitesse linéaire et angulaire de chaque objet. Avec cette solution, le client est donc lui aussi capable de simuler la physique du monde sur les objets. Cela permet ainsi à nos clients d'extrapoler entre le dernier état reçu et le prochain. Dans la plupart des cas, la différence sera minime et impercevable. A cela s'ajoute que nous pouvons désormais choisir intelligemment quels objets seront envoyés au client, en effet, il ne sert à rien d'envoyer 20 messages par secondes pour un objet qui ne bouge pas du tout. Si on choisit les bons objets, on peut significativement augmenter le nombre d'objets synchronisés et ça sans impacter la bande passante utilisée.

3.5.1. Accumulateur de priorité

Dans ce cas, afin de choisir judicieusement quel est la priorité de chaque objet au temps t, nous avons implémenté un accumulateur de priorité. Chaque objet synchronisé implemente l'interface IStateSyncronize Listing 3.6

```
public interface IStateSynchronize
{
    SyncObject Prefab { get; }
    ushort UniqueId { get; set; }
    Rigidbody Rigidbody { get; }
    float Priority { get; set; }

    void CalculatePriority();
}
```

Listing 3.6 – IStateSynchronize.cs

Cette interface oblige tout objet qui l'utilise à avoir une variable de priorité et une méthode qui permet de calculer la priorité de l'objet. Voici par exemple comment l'object "petit cube" recalcule sa priorité.

```
    public override void CalculatePriority()
    {
        if (Rigidbody.IsSleeping())
        {
            Priority += 1;
        }
        else
        {
            Priority = (Rigidbody.velocity == Vector3.zero && Rigidbody.angularVelocity
                == Vector3.zero) ? Priority + 10 : Priority + 100;
        }
    }
```

Listing 3.7 – StateSyncSmallCube.cs:CalculatePriority() methods

On peut voir, qu'en rapport direct avec son activité, sa priorité augmentera de 1, 10 ou 100. La raison pour laquelle on augmente également sa priorité même si l'objet est totalement inactif et que l'on souhaite quand même de temps en temps mettre à jour tous les objets. C'est par exemple très utile lors de la connection d'un nouveau client, car il faut également qu'il reçoive les données des objets inactifs. Chaque vingtième de secondes, le serveur oblige chaque objet à recalculer leur priorité et envoit les 20 objets les plus prioritaires. Après avoir inclu les objets prioritaires dans le paquet, il réinitialise leur priorité à zéro.

3.5.2. Tampon contre la gigue

Ici, puisque notre client simule lui aussi la simulation physique, le moindre décalage entre le temps de réception des paquets peut avoir un gros impact négatif sur la fluidité de la simulation. En effet, le serveur envoie des mises à jour exactement toutes les 0.05 secondes (20 fois par seconde) mais côté client, il est extrêmement rare de les recevoir exactement

espacés de ce même délai. Soit on en reçoit plusieurs à la fois, soit certains peuvent être perdus, soit dans un temps plus au moins proche de ces 0.05 secondes, on appelle cela la gigue (ou jitter en anglais).

Afin de rémedier à cela, il a fallu mettre en place une nouvelle sorte de tampon contre la gigue. Le tampon a la responsabilité de mettre les paquets dans une file d'attente et de libérer chaque paquet précisément 0.05 secondes après le précédent. La tampon prend aussi en compte la perte des paquets. Par exemple, si nous nous trouvons dans un cas ou un paquet est perdu, alors le tampon libérera le prochain paquet exactement 0.1 seconde après le dernier.

On va maintenant expliquer son fonctionnement en détail et s'aidant des Figures 3.4, 3.5 et 3.6.

— Les cases jaunes reprénsentent les paquets en attente d'être libérés ; il y a toujours quatre paquets dans la gigue et donc 200ms de décalage volontaire.

— La case verte indique que le paquet est libéré.

— La bordure rouge indique un paquet qui n'a pas encore ou jamais été reçu.

— Les chiffres en dessus du tableau indiquent le temps restant en millisecondes avant la libération de ce paquet.

D'après la Figure 3.4, on peut voir que trois paquets attendent d'être libérés. Dans un monde parfait, il devrait y en avoir quatre, mais le paquet n°23 a été perdu.

Ensuite nous avons la Figure 3.5, le paquet n°21 peut bel et bien être libéré. Le prochain paquet, le n°22 sera libéré dans 50 millisecondes.

Finalement sur la Figure 3.6, nous voyons que le prochain paquet a être libéré sera le paquet n°24 et il ne devra pas être libéré dans 50ms mais dans 100ms. Nous avons également le paquet n°30 qui est arrivé et a été placé dans le tampon.

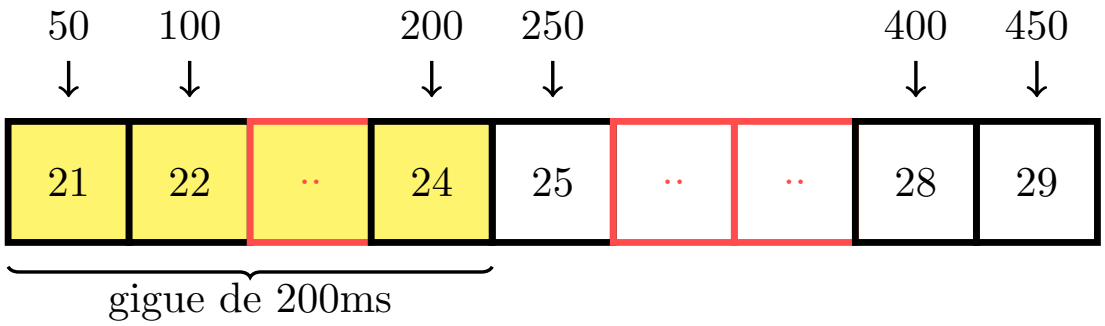

Figure 3.4. – Tampon au temps t + 0ms

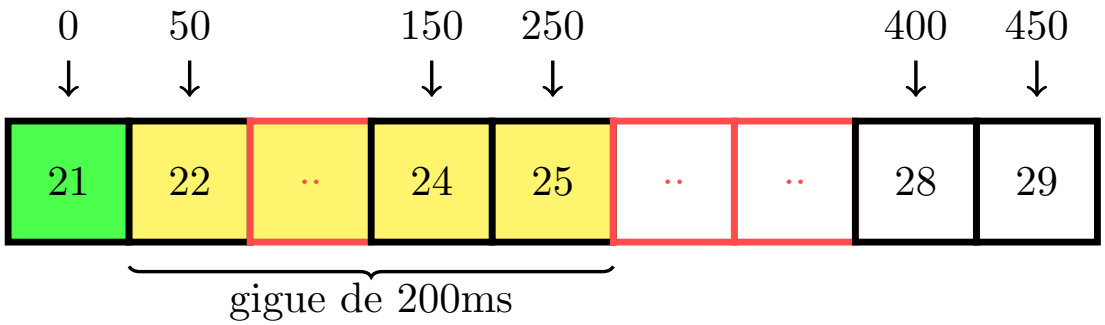

Figure 3.5. – Tampon au temps t + 50ms

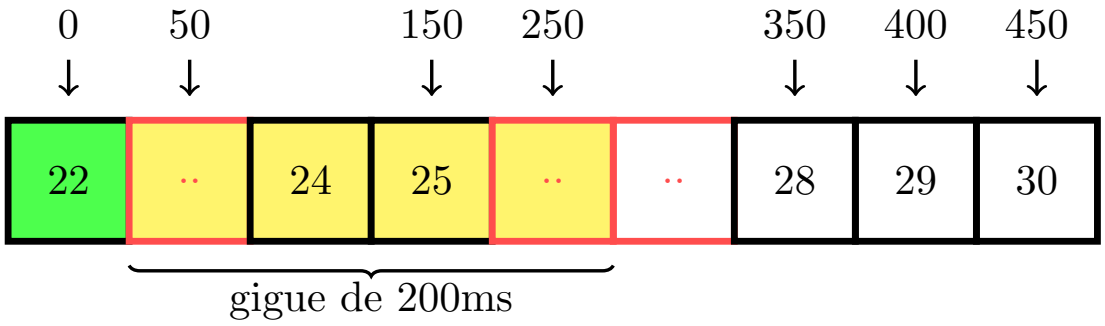

Figure 3.6. – Tampon au temps t + 100ms

3.5.3. Analyse des résultats

Avec cette solution, en plus de transmettre les données de position et de rotation, il faut également envoyer la vitesse linéaire et angulaire de chaque objet.

- La position de chaque objet est définie par un vecteur 3 qui est composé de trois float. Étant donné que la représentation d'un float en C# nécessite 4 bytes, il nous faut donc envoyer 12 bytes par objet pour synchroniser la position.
- La vitesse linéaire est définie par un vecteur 3 et nécessite donc 12 bytes par objet.
- La vitesse angulaire est elle aussi définie par un vecteur 3 et nécessite également 12 bytes.
- La rotation de chaque objet est, quand à elle, définie par un quarternion qui se compose de quatre float. Il nous faut donc transmettre 16 bytes par objet afin de synchroniser la rotation.
- Chaque objet dispose d'un ID unique, ce dernier est encodé par un ushort (choisi arbitrairement) et qui utilise donc 2 bytes.

Pour résumé, nous avons donc besoin d'un total de 54 bytes par objet *(12+12+12+16)*. Cela signifie que si l'on souhaite ne pas dépasser le MTU de 548 bytes, nous pouvons transmettre jusqu'à dix objets par paquet.

L'analyse des résultats de cette solution est assez compliquée à faire. En effet, comme cette solution dépend entièrement du nombre d'objets actifs, il faut procéder à plusieurs tests et apprendre par essais et erreurs. Nous pouvons néanmoins nous faire un ordre d'idée à l'aide du Tableau 3.1. À partir d'un nombre d'objets total et d'un nombre de snapshot par secondes, nous pouvons déduire le temps qu'il faut à un objet qui vient d'être transmis avant qu'il soit à nouveau transmis. Ici, on émet l'hypothèse que tous les objets disposent d'une augmentation de priorité identique les uns des autres. Dans le tableau, nous appelons ce temps le "Temps de boucle".

Total d'objets	**Snapshots par secondes**	**Temps de boucle**
100	20	500ms
100	60	166ms
200	20	1000ms
200	60	333ms

Table 3.1. – Apperçu des coûts (Synchronisation des états)

À l'aide du logiciel Clumsy [1], il est possible de simuler des problèmes de réseau tels que :

— Du lag, en retenant les paquets pour une petite période de temps.
— La perte aléatoire de certains paquets.
— Bloquer l'envoi d'un petit nombre de paquets afin de les recevoir tous en même temps.
— Dupliquer certains paquets.
— Mélanger l'ordre des paquets reçus.

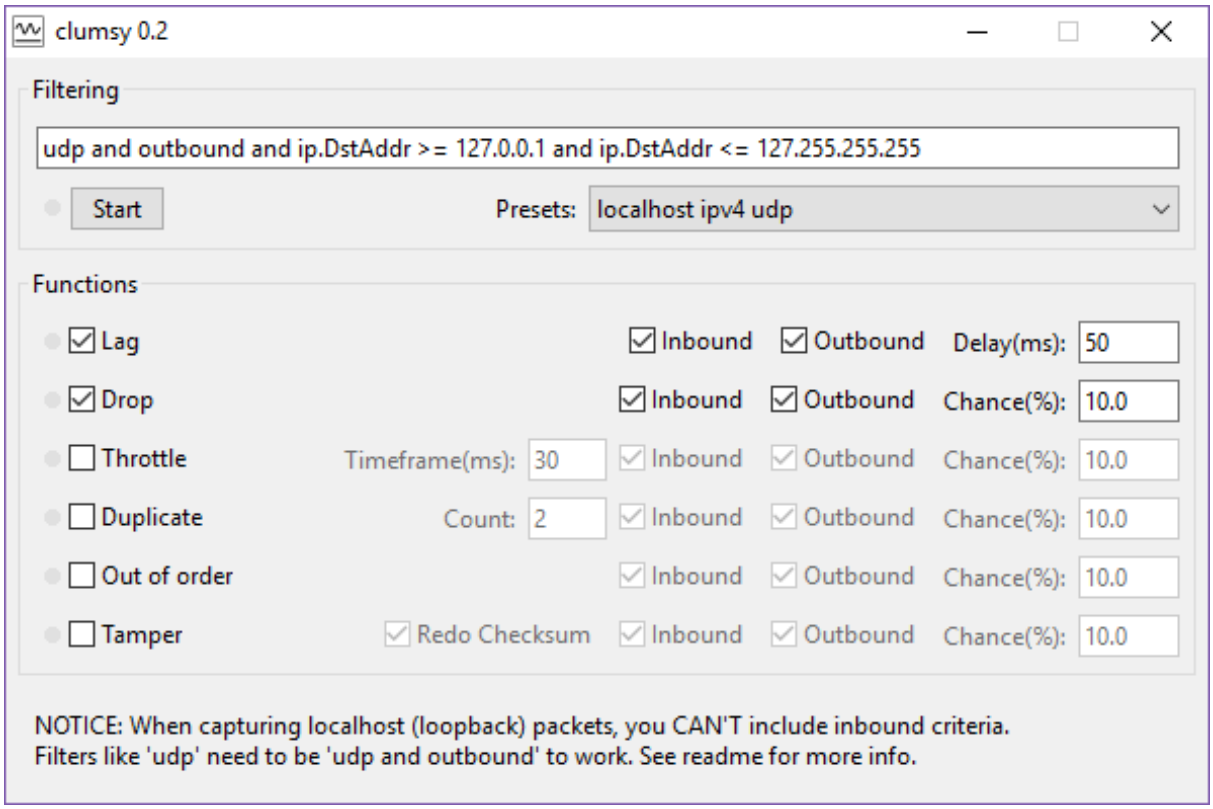

Figure 3.7. – Interface de Clumsy

Avec notre simulation qui contient cent objets et quatre joueurs et par expérimentation, nous allons essayer de trouver à partir de quel moment la perte de paquets devient visible car elle devient trop importante pour notre système.

On remarque des problèmes dès que l'on dépasse 13% de perte de paquets lorsque l'on envoit 20 snapshot par secondes. Et avec 60 snapshot par secondes, nous pouvons monter jusqu'à une perte de 37%.

3.5.4. Améliorations possibles

Contrairement à la première technique, il existe un certain nombre d'améliorations possible avec cette technique.

En effet, étant donné que le client connait les règles physiques de la simulation, il est possible simuler directement les mouvements de notre cube sans avoir à attendre la réponse du serveur. Il faudra par contre mettre en place un système de réconciliation car le serveur reste authoritaire sur la simulation et cette dernière n'est pas déterministe. Il est donc possible que des divergences se produisent avec le temps.

Il est également possible d'augmenter dynamiquement le nombre d'objets présents par paquets en implémentant un système qui n'induit la vitesse linéaire et angulaire des objets dans le paquet, seulement si l'objet en question est en mouvement. Cela permettrait de réduire le coût par objet statique de 52 bytes à 28 bytes. Par exemple, si seulement cinq

objets sont en mouvement, alors nous pouvons envoyer 15 objets *(52*5 + 28*10)* dans le paquet sans dépasser le MTU.

La compression des données, comme utilisé lors de l'autre solution est également possible, mais elle a tendance à émettre quelques artefacts visuels car le moteur physique n'apprécie pas la diminution de la précision. En effet, certains cubes alors pénètrent légèrement dans leurs voisins et le moteur physique réagit trop violemment en projetant les cubes. Il est possible que sur un autre moteur que sur Unity, ces artefacts soient moins importants. Mais dans notre cas, il est préferable de ne rien compresser.

3.6. Autres techniques de synchronisation

Il existe bien sur d'autres techniques de synchronisation et d'autres variantes des techniques utilisés pour ce travail. Mais elles n'ont pas été integrées car elles sortaient du cadre de ce travail.

3.6.1. Simulation déterministe

La simulation déterministe est une solution assez élégante, mais malheuresement impossible à implémenter avec le moteur de jeu Unity. En effet, cette technique se base sur le principle d'une gestion de la physique deterministe. Ce qui induit que si l'on part d'une base identique et que l'on indroduit les mêmes contrôles en même temps, alors le résultat des simulations seront identiques. L'impact d'une telle physique deterministe est révolutionnaire dans notre cas, car nous aurions uniquement besoin de synchroniser les contrôles et tout le reste de la simulation, et ce peut importe la compléxité et le nombre d'objets, serait parfaitement synchronisé. Il est malheuresement très difficile d'avoir un moteur physique qui permet une simulation déterministe sur deux machines différentes.

3.6.2. Comparaison des résultats

Comme on a pu le voir, la technique de l'interpolation de snapshot reste la solution la plus simple à mettre en place et à tester si la simulation dispose de moins d'une centaine d'objets. Elle n'est par contre pas capable de synchroniser un nombre plus élevé d'une centaine d'objets. Cette technique permet également de calculer par avance et de manière théorique le nombre maximal d'objets synchronisables. Cette solution est fortement conseillée car plus simple à prévoir et ne nécessite peu de tests pour confirmer qu'elle fonctionne correctement.

La solution de synchronisation des états est quant à elle capable de simuler un plus grand nombre d'objets, mais nécessite un très gros travail par essais et erreurs afin d'obtenir une simulation correcte. Elle est également beaucoup plus dure à maintenir sur le long terme si notre simulation change, car il faudra à nouveau retester quels sont les paramètres optimaux. Si nous disposons d'un grand nombre d'objets différents avec des priorités différentes, il sera également long et difficile pour trouver les valeurs optimales des priorités. Cette technique est néanmoins capable de simuler des cas beaucoup plus complexes

et variés, et permet de résoudre certains problèmes qui seraient impossible avec l'autre solution.

4
Conclusion

4.1. Prise de recul

Ce travail a commencé par l'idée simple et naïve d'implémenter deux techniques de synchronisation des articles de Glenn Fiedler. Il s'est ensuite averé être un travail bien plus complexe et touchant des aspects bien plus vastes qu'initialement prévu.

J'ai pu constater qu'il n'existe pas d'implémentation miracle pour résoudre ce problème, mais que chaque technique apporte du bon et du mauvais. J'ai également pu constater que le contexte dans lequel l'on souhaite intégrer cette solution détient une très grande importance dans les choix des solutions.

Je tiens encore a mettre en avant que ce travail n'est qu'une introduction au problème et qu'il serait intéressant d'élargir le cadre de recherche à d'autres implémentations et à d'autres simulations plus variées ; cela permettrait d'identifier si dans certains cas, une solution en particulier sort du lot.

4.2. Derniers mots

L'enjeu personnel de ce travail se trouve dans le fait qu'il n'existe pas une seule et unique solution parfaite à ce problème et qu'il a fallu avant tout expérimenter et implémenter plusieurs techniques afin de développer une vision globale des enjeux et du problème.

A

Acronymes

AND	ET logique
API	interface de programmation
float	nombres à virgule flottante
MTU	Maximum Transmission Unit
OR	OU logique
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
uint	entier non-signé

B

License of the Documentation

Bibliographie

[1] Clumsy : an utility to simulate broken network. https://jagt.github.io/clumsy/ (dernière consultation le Juillet 15, 2018). 20

[2] James W. Cooper. *C# Design Patterns : A Tutorial*. 2002.

[3] Fast-Paced Multiplayer. http://www.gabrielgambetta.com/client-server-game-architecture.html (dernière consultation le Juillet 16, 2017).

[4] GDC talk : I Shot You First. http://www.gdcvault.com/play/1014345/I-Shot-You-First-Networking (dernière consultation le Juillet 17, 2017).

[5] GDC talk : Physics for game programmers networking. http://www.gdcvault.com/play/1022195/Physics-for-Game-Programmers-Networking (dernière consultation le Juillet 14, 2017).

[6] Josh Glazer and Sanjay Madhav. *Multiplayer Game Programming : Architecting Networked Games*. 2015.

[7] Site web de Glenn Fiedler, articles "Networked Physics". https://gafferongames.com/categories/networked-physics/ (dernière consultation le Janvier 08, 2018). 2

[8] Using abstractions and interfaces with Unity3D. https://www.gamasutra.com/blogs/VictorBarcelo/20131217/207204/Using_abstractions_and_interfaces_with_Unity3D.php (dernière consultation le Juillet 16, 2017).

[9] Source code de SmartNet, par Kyle Douglas Olson. https://gitlab.com/kyledouglasolsen/SmartNet_Source (dernière consultation le Janvier 19, 2018). 2

[10] W. Richard Stevens. *UNIX Network Programming, Volume 2, Second Edition : Interprocess Communications*. 1990. 10