EPSI NANTES
BP 41903
114 rue des Hauts Pavés
44019 NANTES CEDEX 1

Asyncron Games
13, Route de Bouguenais
44830 Bouaye

Echanges de données dans un jeu multi-joueurs temps réel

Samuel Bouchet

Directeurs de recherche :

Directeur technique : M. NAIM

Directeur communication : Mme CABAZ

Promotion 2010

Soutenance Septembre/2010

Remerciements

Il me parait bienvenu de commencer par remercier les personnes qui m'ont aidé dans la réalisation de ce mémoire.

Merci à Ahmed NAIM et Axelle CABAZ, directeurs de recherche, pour leurs conseils sur les éléments techniques et rédactionnels.

Merci à Françoise et Pierre BOUCHET, et à Magali PLOTEAU pour leurs relectures attentives.

Merci à Olivier CHANRY et Thierry MATTRAY, directeurs général et technique d'Asyncron Games, pour m'avoir accueilli au sein de l'entreprise et pour leurs conseils.

Merci à Vincent BANCHET, Anthony LOIZEAU, Ludovic MARC et Maxence VOLEAU pour m'avoir aidé à tester en réseau les programmes développés durant mes recherches et pour leurs conseils.

Sommaire

INTRODUCTION **2**

CHAPITRE 1 – OPTIMISATION DES ECHANGES DE DONNEES **9**

CHAPITRE 2 – GENERALISATION DES OPTIMISATIONS **32**

CHAPITRE 3 – APPLICATION DES MODELES AVEC C# ET XNA **42**

CONCLUSION **62**

SCHEMAS, FIGURES, ET TABLES **67**

GLOSSAIRE **68**

TABLE DES MATIERES **69**

BIBLIOGRAPHIE **71**

ANNEXES **73**

Introduction

Depuis quelques années le temps de travail moyen tend à diminuer en France. En effet, d'après l'INSEE, la durée de travail annuelle est passée d'un peu plus de 2200 heures par an en 1950 à moins de 1600 heures par an en 2007, soit une réduction de presque 30% ces 50 dernières années. Les raisons peuvent être multiples : automatisation des procédés (industrialisation), délocalisation due à la mondialisation, augmentation de l'espérance de vie et de la population. Dans tout les cas le constat est là : le taux de chômage est en constante augmentation et les hommes travaillent, en moyenne, moins longtemps. [15]

Globalement, on peut donc penser que l'« Homme », en France, consacre de moins en moins de temps à ses activités professionnelles, et dispose de plus en plus de temps libre qu'il peut utiliser en loisirs.

Cette tendance se confirme également dans les dépenses des ménages pour les loisirs. Toujours d'après l'INSEE dans une autre analyse, une personne dépense en moyenne 1257 € par an en 2005 contre 718 € en 1990 pour le budget « loisirs ». Il à donc presque doublé en quinze ans. Le loisir inclut beaucoup de domaines : cinéma, lecture, jardinage, sport, équipements multimédias, mais aussi jeux vidéo. [18]

Si le jeu vidéo apparaît encore parfois comme marginal auprès de générations plus anciennes, c'est en réalité une des activités actuelles de divertissement les plus florissantes. D'après un article de l'agence française pour le jeu vidéo (afjv) basé sur des chiffres de Media Control GfK International, le marché du divertissement culturel en France est en légère régression entre 2004 et 2008. Cependant, cet article montre que les achats de livres, DVD et CD audio subissent une baisse des volumes des ventes de 1 à 15% alors que les ventes de jeux vidéo sont en augmentation de 20%. Dès 2006, l'afjv publie même une étude comparant l'industrie du jeu vidéo à l'industrie du cinéma. Elle annonce que « *le business des jeux vidéo aux Etats-Unis a atteint un sommet record en 2005, avec $10.5 milliards de dollars de chiffre (…). Ce score est comparable à celui du cinéma dans ce pays dont le chiffre d'affaires 2005 lui est même légèrement inférieur : US$10.3 milliards.* » [19][20]

Plus récemment, fin 2009, c'est le jeu « Call of Duty : Modern Warfare 2 » qui décroche des nouveaux records. Jeuxvideo.com nous annonce que « *Avec 550 millions de dollars de recettes engrangés en 5 jours seulement, Modern Warfare 2 détient désormais le record du meilleur lancement d'un produit culturel, toutes industries confondues. Il bat ainsi les 394 millions de dollars de « Harry Potter et le Prince de Sang Mêlé » en littérature, les 204 millions de dollars réalisés par « The Dark Knight » au cinéma et même les 500 millions de dollars ramassés par « Grand Theft Auto IV* » *à son lancement*. » [21]

Une partie de ce succès s'explique par la façon dont les équipes d'Activision ont su reprendre dans leur jeu Modern Warfare 2 tous les concepts les plus appréciés des joueurs, et aussi par la réalisation artistique de bonne qualité. D'un côté, nous avons des images magnifiques, des musiques et une bande sonore grandioses, des personnages attachants, un scénario recherché. De l'autre côté, on profite d'un système laissant au joueur une grande part à la progression et à l'évolution de son personnage, des possibilités de personnalisation, des systèmes de jeux qui ont fait leurs preuves (comme la capture de drapeaux ou la pose/désamorçage de bombes), un mécanisme permettant de trouver facilement d'autres personnes pour jouer en ligne, et un système de « Trophée » mettant en valeur ses réussites auprès de la communauté.

D'ailleurs, seuls les jeux « en ligne » ont cette dimension communautaire : les joueurs peuvent échanger leurs scores et réussites, se confronter aux autres joueurs connectés à tout moment, développer des codes de communications communs tel que des abréviations pour désigner les lieux ou les stratégies à employer. La plupart des plus gros succès actuels sont des jeux « online » qui ont réussi grâce à un mode* multi-joueurs très développé à conquérir et garder des communautés de joueurs.

Tout cela est aujourd'hui possible grâce au fort succès d'Internet ces dernières années. Internet est désormais un réseau de communication incontournable, utilisé régulièrement et intensivement par des millions de personnes à travers le monde. Pour le jeu vidéo, il est un support permettant de toucher le monde entier.

Cependant, dans un contexte multi-joueurs, il peut exister un grand nombre d'interactions entre les joueurs. Par exemple, dans les jeux d'action, des informations entre les joueurs doivent être échangées en continu avec une fréquence idéale qui serait égale à la fréquence d'affichage du jeu : 60 échanges par secondes. Jake Simpson suggère dans « Networking for

Games 101 » que le réseau Internet ne permet pas d'atteindre cet idéal à cause des limitations du matériel qui le compose et de la quantité de données à échanger. Dans la pratique les développeurs utilisent des fréquences allant de 10 à 30 échanges par seconde, comme nous l'expose Glenn Fiedler dans *Networking for Game Programmers*. [12][7]

De plus, des problèmes de réseaux vont se produire de manière inévitable et devront faire partie intégrante de la réflexion des concepteurs de jeux. En effet, lorsque des paquets sont envoyés entre 2 nœuds du réseau informatique, ces paquets peuvent être perdus ou ne pas arriver dans le bon ordre. Les conséquences dans un jeu peuvent être catastrophiques : le personnage n'avance pas droit ou n'avance pas du tout, et l'action parfaite permettant au joueur de remporter la partie se perd dans les méandres d'Internet… Si de tels comportements pour un jeu peuvent paraitre aberrants, ils existent ou ont existé dans certains jeux. Ceux qui ont tenté de jouer sur internet à Quake, à l'époque où l'adsl n'existait pas encore peuvent en témoigner : voir son personnage se déplacer une seconde après que la commande soit enclenchée ne permet pas d'anticiper correctement ses trajectoires et gâche le plaisir de jeu.

Une grosse difficulté dans les jeux multi-joueurs en temps réel est donc de parvenir à optimiser au mieux les échanges entre les clients de façon à utiliser un minimum de bande passante. L'objectif final est de faire un maximum d'échanges de données et ainsi de rendre le jeu fluide. Même si les réseaux sont de plus en plus fiables de nos jours, les problèmes évoqués surviennent fréquemment et doivent être gérés pour assurer une qualité de jeu permettant de satisfaire les joueurs.

En outre, l'objectif n'est pas tant d'améliorer la fluidité des informations qui circulent que de donner au joueur l'impression qu'il évolue dans un monde cohérent. Il n'est donc pas nécessaire de lui retransmettre l'environnement de jeu de façon exact. Aussi, quelques astuces d'anticipation des informations en retard (latence de la connexion) ou absentes (perte de paquets) permettront de donner cette impression de cohérence. Par exemple, il existe des méthodes qui peuvent s'assimiler à de la distorsion temporelle et de la prédiction. Si ces deux notions semblent sorties de la science fiction, elles sont parfaitement applicables à un monde virtuel dont le développeur maitrise les règles. Ainsi, de la même manière qu'un artiste peut exagérer des traits pour donner cette impression de "plus vrai que nature", le développeur peut adapter l'image donnée au joueur en fonction de la réalité qu'il souhaite lui montrer. Pour une

meilleure expérience de jeu, il est impératif de parvenir à rendre cette impression. Optimiser les échanges de données dans un jeu multi-joueurs en temps réel[1] devient donc un des éléments clés de sa réussite.

L'opération n'est pas aisée : il faut pouvoir appréhender tous les algorithmes de compression, les méthodes d'optimisation, penser l'analyse du programme de jeu pour qu'il puisse fonctionner en réseau et s'adapter aux contraintes des matériels et outils.

Ces problèmes doivent cependant se découpler au maximum du gameplay*. Le gameplay* est l'essence du jeu et ne devrait pas ou peu être affecté par ces problèmes techniques. Pourtant, une grande partie des ressources du développement du jeu doivent être investies dans la résolution ou le contournement de ces problèmes technique, autant de ressources qui ne profiteront pas au gameplay*. De façon à se concentrer au maximum sur le cœur du jeu, il convient de s'affranchir des problèmes techniques récurrents et d'appréhender facilement les contraintes. Des méthodes existent.

En effet, dans le développement de logiciels, la modélisation et la génération de code à partir de modèles est l'une des pistes explorée les plus avancées pour industrialiser des développements. On parle aussi de MDA (Model Driven Architecture) ou « architecture pilotée par les modèles ». L'investissement en temps est considérable pour développer le modèle et son application, mais le temps investi est compensé par autant de réutilisations du modèle par la suite. Il s'agit de réaliser une application permettant de développer plus vite d'autres programmes.

Dans le cas particulier des échanges de données dans les jeux temps réel, où les contraintes techniques sont très fortes, les modèles doivent abstraire la partie récurrente de la complexité pour ne plus avoir à s'occuper que du strict nécessaire. A terme, on développe donc vraiment ce qui est essentiel à la valeur ajoutée du jeu. Ainsi, on peut concentrer ses efforts sur l'innovation sans perdre plus de temps à réfléchir aux problèmes déjà pensés pour lesquels une simple « utilisation » du modèle sera suffisante.

[1] « Temps réel » dans le contexte du jeu vidéo signifie « en simultané » et n'a pas le même sens qu'il pourrait avoir en développement embarqué.

Devant toute cette complexité, il est normal de s'interroger sur la possibilité de créer un modèle qui génère automatiquement les fragments de code nécessaires aux échanges pour un jeu temps réel.

En d'autres termes, est-il possible de modéliser un traitement optimal des échanges de données dans un jeu multi-joueurs temps réel ?

La tâche consiste à identifier les concepts et les solutions communes utilisées dans différents jeux temps réel en réseau pour ensuite automatiser les optimisations qui peuvent l'être.

Tout d'abord, il faut savoir que la plupart des jeux vidéo consistent en un assemblage de « moteurs » visant à fonctionner ensemble au sein de l'application. Ainsi, le moteur de rendu vidéo a pour objectif de fournir au joueur une représentation graphique des objets du jeu sur l'écran. Le moteur de son gère la représentation sonore des objets du jeu par les enceintes. Le moteur physique s'occupe de calculer les emplacements des objets en suivant des contraintes physiques. Le moteur d'entrées/sorties écoute quant à lui les entrées du joueur sur les périphériques de jeu (clavier, souris, manette) et répercute les conséquences de ses actions sur le jeu. Enfin, le moteur réseau assure les communications réseaux entre les différents joueurs.

Ces moteurs ne fonctionnent cependant pas de manière indépendante. En effet, ils sont en réalité rassemblés en un « moteur de jeu » qui consiste à les faire cohabiter tout en implémentant les fonctionnalités propres au jeu. Avec un tel fonctionnement, l'information suit le cheminement suivant : lectures des commandes du joueur, modification des objets impactés en fonction des entrées et des calculs physiques, envoi de données au moteur réseau pour répercuter les modifications chez les autres joueurs.

On remarque alors que l'optimisation des échanges ne peut pas se faire uniquement au niveau du moteur réseau. En effet, celui-ci se « contente » d'envoyer les informations et modifications qu'on lui demande d'envoyer, mais n'en effectue aucun contrôle et ne connait ni le sens ni l'importance de l'information. Si dans un jeu de stratégie, on lui demande d'envoyer les positions pour l'armée de 1000 soldats du joueur, alors les 1000 coordonnées circuleront par le réseau et s'échangeront entre les joueurs. Un tel volume de données est bien trop important pour être fluide, et le plaisir de jeu est gâché.

Une solution pourrait consister à n'envoyer que les ordres. Le joueur sélectionne les 1000 unités et les 1000 identifiants passent par le réseau (1 seule fois), puis l'ordre de déplacement est envoyé indiquant les coordonnées souhaitées (une seule fois) et à la charge du moteur de jeu de chaque joueur de calculer la trajectoire. Nous avions donc 1000 coordonnées multipliées par le nombre d'envois par seconde multipliés par la durée du déplacement, et nous optimisons en utilisant plus que 1000 identifiants et 1 coordonnée une fois.

Cet exemple a pour but de montrer que se préoccuper du moteur réseau uniquement ne suffit pas à optimiser les échanges de données, même s'il semble être le plus concerné. C'est l'ensemble du système de jeu qui doit être pensé en vue du multi-joueurs temps-réel pour être optimisé. En effet, dans l'exemple précédent, le changement de fonctionnement a réduit de façon drastique le volume de données échangées, mais oblige en contrepartie chacun des joueurs à calculer sur sa machine les trajectoires de la même façon.

Seulement, il peut a priori exister autant d'optimisations différentes que de concepts de jeux, et nous nous limiterons aux fonctionnalités essentielles pour les jeux d'action.

La catégorie des Jeux de tir à la première personne (ou FPS pour First Person Shooter ou) est la plus représentative des jeux d'action temps réel. On peut penser par exemple à Counter Strike, Call of Duty, Medal of Honor, Battlefield ou encore Half-Life qui figurent parmi les plus joués du genre. Dans un FPS, le joueur incarne un unique personnage qui dispose de capacités lui permettant de remplir les objectifs fixés par le mode* de jeu. Les modes les plus connus sont : le combat à mort où le but est d'éliminer seul ou par équipe le plus d'adversaires possible sans mourir, la capture de drapeaux où l'objectif est d'aller capturer un drapeau pour le ramener à sa base, la pose de bombe où une équipe devra poser et faire exploser une bombe pendant que l'autre tentera de l'en empêcher.

Tous les jeux intègrent également des fonctionnalités communes nécessitant des échanges par le réseau comme la communication entre les joueurs (vocale et/ou textuelle), l'affichage d'informations de la partie en cours (scores, classements, statistiques, état), la gestion et l'établissement des connexions entre les joueurs (créer une partie, rejoindre une partie, recherche de joueurs) ainsi que le scénario de la partie (ensemble d'évènements indépendants des actions des joueurs).

Nous essayerons d’optimiser les échanges de données pour les jeux d’action de type dans le chapitre un, en partant de la conception et des contraintes du jeu pour aller vers des méthodes d’optimisation.

Ces optimisations seront ensuite généralisées dans le chapitre deux en dégageant leurs interfaces*. Il s’agira donc de définir un modèle qui supportera ces interfaces.

Enfin, le chapitre trois viendra concrétiser ces recherches en appliquant les concepts des chapitres un et deux à un jeu de tir spatial en 2D. Nous proposerons des implémentations générique et spécifique en C#.

Chapitre 1 – Optimisation des échanges de données

Avant toute concrétisation d'un développement logiciel, il est important de passer d'abord par une phase de conception. Le jeu vidéo ne fait pas exception, et nécessite même des considérations spécifiques pour être jouable en réseau à plusieurs. En effet, des contraintes de cohérence, de temps de latence*, et de montée en charge doivent être respectées. Elles influenceront nos choix concernant le mode de transmission des données, l'architecture réseau à utiliser et l'architecture du jeu en lui-même.

1. Besoins d'un jeu en réseau

Dans tout cycle de développement* logiciel, le chef de projet commence par l'analyse du besoin et de la faisabilité, puis rédige les spécifications fonctionnelles et techniques. Ces dernières décrivent les fonctionnalités précises attendues du logiciel et les moyens techniques à mettre en œuvre pour y parvenir.

Le jeu vidéo ne faisant pas exception, voyons les besoins et la faisabilité d'un jeu vidéo multi-joueurs. Les besoins du jeu détermineront ensuite les contraintes que nous devront respecter et nous pourront alors orienter nos choix techniques et nos choix de conception.

1.1. Contraintes d'un jeu réseau

On attend plusieurs choses d'un jeu vidéo multi-joueurs en temps réel. Il doit permettre à plusieurs joueurs de jouer ensemble sans qu'ils soient physiquement proches, présenter un environnement de jeu cohérent à chacun des joueurs, autoriser des interactions immédiates entre les joueurs, et faire évoluer l'environnement de jeu de façon fluide.

1.1.1. Support de communication

Avant la montée d'Internet, les jeux vidéo se limitaient à gérer plusieurs joueurs sur une même machine, ou sur un petit réseau local pour les plus avancés. Jouer en ligne était considéré comme marginal pour Doom par exemple. [7] Il est maintenant standard qu'un jeu vidéo soit capable de gérer plusieurs joueurs à travers le monde.

Que les joueurs utilisent un ordinateur ou une console, le seul support disponible et fonctionnel pour faire communiquer les machines à grande échelle est le réseau Internet. À défaut d'autre support existant il sera notre moyen de communication, et notre objectif sera de répondre aux autres contraintes d'un jeu vidéo dans les limites que nous impose ce support.

On identifie trois types de limites auxquelles les applications utilisant Internet doivent faire face : la bande-passante*, le temps de latence* (ou temps de réponse), et la puissance de traitement des flux de chaque nœud. [6]

La bande passante est la capacité de transmission de données entre deux nœuds d'un réseau. Pour les réseaux internet par exemple, la bande passante maximale théorique d'un modem bas débit avec le reste du réseau est de 56 kbps (kilobits par seconde) contre 2 à 20 Mbps (mégabits par seconde) pour les offres haut-débits. Cela veut dire qu'il est théoriquement possible de transférer, sur un modem bas débit, jusqu'à 56 kilobits (soit 7 kilo-octets) par seconde. Dans la pratique, il est rare d'obtenir avec un modem bas débit plus de 5 ko/sec de façon constante. Dans la mesure où la plupart des joueurs sont aujourd'hui équipés avec des lignes haut-débit, nous considèrerons que nous disposons d'une bande passante de 1 Mbps soit 128ko/sec.

Le temps de latence* est le temps écoulé entre le moment où le message est envoyé et celui où la réponse à ce message est reçue. C'est-à-dire que plus le temps de latence* est grand, et plus l'information met de temps à arriver à son destinataire. La latence ne pourra jamais être nulle car le signal électrique lui-même n'est pas instantané. De plus, le temps de traitement des messages au niveau des nœuds du réseau augmente le temps de latence*, donc plus il y a d'intermédiaires et plus le temps de latence sera grand. L'éloignement physique, par exemple, force un message à passer par de nombreux routeurs du réseau internet, ainsi les temps de latence* de joueurs sur des continents différents seront plus importants que celui de joueurs du même pays.

Si Internet permet le support de plusieurs joueurs à distance, un jeu a aussi d'autres besoins. Malgré les difficultés liées à ce réseau, il faut par exemple assurer sa cohérence.

1.1.2. Cohérence

La cohérence d'un jeu est sa capacité à donner au joueur une image dont il ne peut réfuter la véracité. L'image doit pour cela correspondre exactement aux règles fixées par le jeu et considérer les actions que les joueurs ont effectuées.

C'est uniquement si cette cohérence est respectée que les joueurs pourront apprécier le jeu. Dans le cas contraire, ils auront le sentiment de ne pas avoir le contrôle car les règles ne sont pas respectées.

Les principaux ennemis de la cohérence dans les jeux sont les erreurs d'affichage (typiquement un objet qui n'apparait pas à son emplacement réel, empêchant toute interaction), et la triche (un joueur cherchant à outrepasser les règles en exploitant des failles techniques ou en faisant de la collusion*).

1.1.3. Réactivité

La cohérence est largement aidée par une forte réactivité. En effet, si un objet mal positionné est rapidement replacé alors l'incohérence sera minime voire imperceptible par le joueur.

Il est également important pour le joueur d'avoir un retour rapide de ses actions à l'affichage. Cet aspect est d'autant plus vrai pour les jeux d'actions comme les FPS, dans lesquels la réactivité et la rapidité du joueur sont des atouts décisifs pour la victoire.

Par conséquent, un des objectifs que nous garderons en vue dans nos choix est de minimiser le temps de latence* pour garder une forte réactivité. Pour un FPS, il devra se rapprocher de 100 ms au maximum et ne jamais excéder 200 ms. Au-delà, entre le moment ou le joueur agit (tir d'une balle sur un autre joueur) et le moment où l'action est prise en compte au niveau du serveur (calcul de la trajectoire), il s'est passé suffisamment de temps pour que le joueur visé se soit déplacé et ait évité la balle. [6]

1.1.4. Fluidité

Si le besoin de fluidité peut paraitre évident pour le joueur, la problématique ne l'est pas pour le développeur. La fluidité peut être altérée par le temps de réponse et dépend aussi de la bande passante. En effet, si la connexion ne permet pas de faire circuler des volumes de

données suffisants, il se créée un goulot d'étranglement et des messages peuvent être perdus ou arriver en retard.

Il sera donc essentiel de minimiser le volume de données à envoyer.

Ces quatre besoins (support multi-joueurs, cohérence, réactivité, fluidité) nous permettent maintenant d'orienter nos choix techniques. Commençons par choisir une architecture réseau.

1.2. Architecture réseau

L'architecture réseau de notre application est la façon dont nous allons organiser les différents acteurs pour qu'ils communiquent entre eux. Il s'agit d'une organisation logique et non physique. C'est-à-dire que nous nous intéressons à la hiérarchie à laquelle sont soumis les acteurs de l'application, sans que les contraintes physiques aient la moindre importance.

Les acteurs sont des nœuds du réseau, ils porteront un nom particulier en fonction du rôle qu'ils ont dans l'architecture. Nous distinguerons trois rôles : client, pair, et serveur.

Les schémas suivants sont tirés de « *Algorithms and Networking for Computer Games* » [6] et représentent les architectures envisageables.

1.2.1. Architectures envisageables.

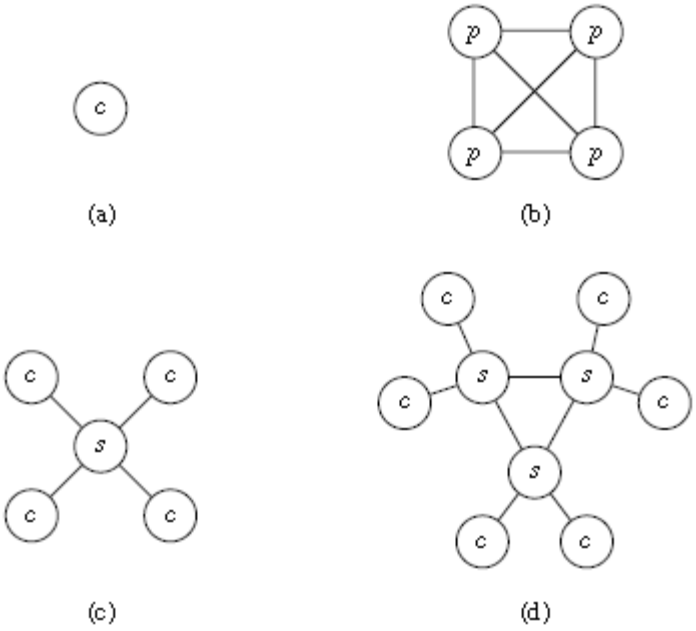

Figure 1 - Architecture d'une application multi-utilisateurs : (a) nœud seul, (b) architecture Peer-to-Peer, (c) architecture client-serveur, (d) architecture réseau de serveurs [6]

La configuration la plus simple pour un jeu multi-joueurs est la configuration (a) ou une machine seule permet à plusieurs joueurs de jouer. Nous laisserons cette architecture de côté dans la mesure où notre objectif est de jouer en réseau.

Dans une architecture Peer-to-Peer (b), tous les pairs sont reliés entre eux. L'intérêt de cette architecture est la simplicité de mise en œuvre et l'absence d'acteurs supplémentaires que les joueurs eux-mêmes. Les premiers jeux en réseaux fonctionnaient principalement avec cette architecture pour une autre raison : c'est l'implémentation multi-joueurs la plus simple pour dériver d'un jeu prévu pour un seul joueur. Son principal défaut est que tous les pairs doivent être connectés entre eux. Par exemple, il faut 6 connexions pour relier 4 pairs entre eux mais 28 connexions pour en relier 8. Un autre défaut est l'absence d'un nœud faisant autorité, c'est-à-dire d'un nœud qui puisse départager 2 autres nœuds en cas de conflit dans les calculs. Cela implique que la triche par exploitation d'une faille technique est possible et qu'un nœud externe ne peut pas rejoindre une partie en cours.

L'architecture client/serveur (c) a ensuite été adoptée pour pallier aux limitations de l'architecture Peer-to-Peer. Dans ce modèle, un nœud est promu serveur et les autres sont des clients reliés uniquement à ce serveur. Toute communication entre les clients passe par le serveur. L'intérêt de cette architecture est que le nombre de connexion est diminué (une par client) impliquant que seul le serveur a besoin d'une connexion performante pour traiter l'ensemble des clients. De plus, les flux de messages sont mieux contrôlés : le serveur peut n'envoyer les messages qu'aux clients concernés. Un autre intérêt est que le flux de données peut être vérifié par le serveur. Celui-ci peut lever les conflits éventuels entre les clients et imposer ses calculs comme référentiel, empêchant la triche. Le serveur peut également accepter des nouvelles connexions en cours de partie, dans la mesure où il est capable de renseigner le nouveau client de l'état actuel de la partie.

Enfin, l'architecture en réseau de serveur consiste à appliquer soit une architecture Peer-to-Peer, soit une architecture client/serveur aux serveurs eux-mêmes. Chaque client est connecté à un serveur local qui lui-même est connecté à un serveur de plus haut niveau ou à d'autres serveurs de même niveau. Cette architecture répartit les connexions des clients entre plusieurs serveurs, autorisant davantage de clients à se connecter. Le principal inconvénient est que les flux réseaux évoluent sur plusieurs niveaux et deviennent plus complexes à gérer.

1.2.2. Répartition les données

En plus d'une architecture définissant une hiérarchie entre les acteurs, il nous faut trouver une manière de rendre les données de jeu accessibles.

En effet, nos contraintes de cohérence et de réactivité devront toujours être respectées. Pour une grande cohérence, nous devons nous assurer que les données en local pour chaque nœud et les données distantes soient fortement liées. Pour une grande réactivité, il faut que les données soient accessibles très rapidement.

Il existe 3 façons d'envisager la répartition des données : la centralisation, la duplication et la distribution.

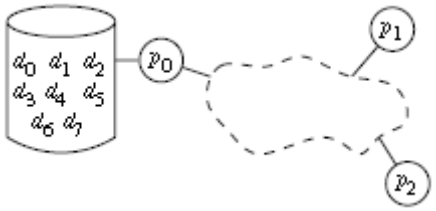

Figure 2 - Centralisation des données [6]

Dans le cas d'une architecture centralisée, un seul acteur possède toutes les données et les autres acteurs doivent passer par lui pour les obtenir. Un tel système garantit une grande cohérence, malheureusement suivant le volume de données et la fréquence des mises à jour des données, la rapidité de récupération des données par les autres acteurs peut être affectée fortement. Dans la mesure où la réactivité est essentielle dans un jeu d'action, il est impensable d'utiliser une telle architecture.

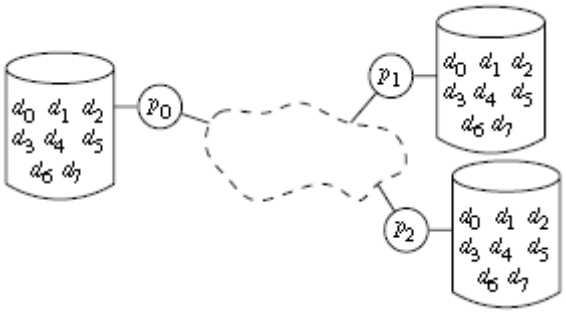

Figure 3 - Duplication des données [6]

Les données peuvent également être dupliquées. Dans ce cas les acteurs possèdent tous une copie locale de l'information. Le temps d'accès est alors très rapide et les données qui circulent sur le réseau sont les ordres de mise à jour de ces données. Pour que la duplication soit efficace, il doit y avoir le moins de mises à jour possible pour les données répliquées.

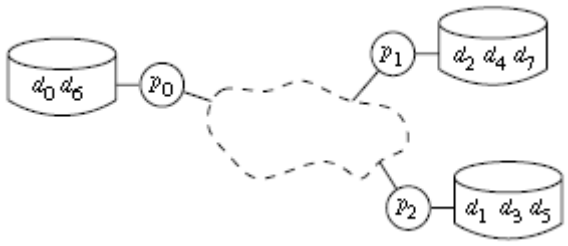

Figure 4 - Répartition des données [6]

Enfin, les données peuvent être réparties. La répartition des données entre les différents acteurs doit être faite de sorte que chacun dispose des informations dont il se sert le plus souvent. Ainsi, la réactivité est assurée aux endroits clés et la cohérence est maintenue pour les acteurs distants.

1.2.3. Choix d'une architecture

Dans un FPS, le nombre de joueurs peut être assez grand. Il n'est pas rare d'aller jusqu'à 16 joueurs par partie, le record actuel pour un FPS étant détenu par MAG qui autorise 256 joueurs à jouer ensemble[2]. Pour un grand nombre de joueurs, l'architecture Peer-to-Peer est inintéressante car pour 16 joueurs il faudrait que chacun établisse 15 connexions avec les autres joueurs, ce qui est peu performant, sans oublier que la triche serait possible. Nous envisagerons donc une solution de type client/serveur ou réseau de serveur.

Lorsque le nombre de clients est de quelques dizaines de clients maximum, l'architecture client serveur est suffisante. Une machine serveur assez performante pourra même héberger plusieurs parties simultanément. Pour un nombre très important de client, le réseau de serveurs est obligatoire. C'est le cas pour les jeux de type MMORPG* où le but est d'accueillir plusieurs centaines de joueurs. Dans les jeux d'action que nous visons à concevoir, nous nous limiterons à une architecture client/serveur avec un seul serveur. Pour approfondir le concept, rien n'empêche pas la suite de considérer un serveur comme le client d'un autre serveur, et d'appliquer deux fois le modèle client/serveur. Les serveurs

[2] Source : http://www.jeuxvideo.fr/e3-mag-256-joueurs-online-ps3-actu-151680.html

intermédiaires devront simplement implémenter les comportements d'un client et d'un serveur.

Concernant les données, nous utiliserons à la fois l'architecture distribuée et répartie. Nous avons vu qu'il est intéressant de répliquer les données lorsque peu de mises à jours sont nécessaires. Pour les calculs déterministes (que l'on peut prévoir à l'avance), il est très facile pour chaque acteur (client ou serveur) de réaliser le calcul de son côté et de mettre à jour sa base de données locale en conséquence. Ainsi, les données sont répliquées et chacun a les mêmes données sans échanger de messages par le réseau (autre que celui synchronisant le départ du calcul). Pour les données et calculs non déterministes (les ordres des joueurs par exemple ne sont pas prévisibles) il sera nécessaire de transférer les données à chaque modification. Chaque joueur est le mieux placé pour connaitre ses propres commandes et doit demander aux autres (en passant par le serveur) leurs commandes. On se situe donc, pour les calculs non déterministes, dans un mode de répartition des données. [7]

1.3. Mode de transmission des données

Dans un réseau informatique, il existe plusieurs manières de faire circuler l'information. Nous allons ici présenter les différents protocoles* et méthodes que nous pouvons utiliser et retenir les solutions qui correspondent le mieux aux contraintes d'un jeu d'action en réseau.

1.3.1. Méthodes d'envoi des données

Trois méthodes existent : l'unicast, le multicast et le broadcast.

L'unicast consiste à envoyer chaque message à un unique destinataire. Avec ce mode, si le même message doit être envoyé du serveur vers tous les clients, il y aura autant d'envois unicast que de clients connectés au serveur et le message sera dupliqué autant de fois.

Le multicast quant à lui propose d'affecter une adresse à plusieurs clients pour un seul émetteur. Ainsi, un seul message peut être envoyé une seule fois et parvenir à tous les destinataires inscrits au groupe. Dans la mesure où les messages ne sont plus dupliqués, cette méthode est un moyen efficace de transmettre une même information à un groupe de clients. [6]

Cependant, le multicast n'est pas supporté par tous les routeurs d'Internet, ni par tous les périphériques réseaux chez les joueurs. Ainsi, il est possible que certains joueurs ne puissent accéder au serveur si celui-ci fonctionne uniquement en multicast. [14]

Enfin, le broadcast permet d'envoyer un même message à tous les nœuds d'un réseau local. S'il peut être employé dans un réseau local, son utilisation sur Internet est impossible car sa diffusion est stoppée par la plupart des routeurs pour éviter la saturation du réseau.

1.3.2. Protocole de transmission

La communication entre plusieurs clients, ou en l'occurrence entre des clients et un serveur, se fait à l'aide de protocoles*. Le protocole* de plus bas niveau permettant la communication sur Internet est IP (Internet Protocole). Il est rarement employé directement par les applications en réseaux qui utilisent plutôt TCP (Transmission Control Protocol) ou UDP (User Datagram Protocol). Chacun est une surcouche du protocole* IP, c'est pourquoi on parle parfois de TCP/IP et de UDP/IP.

De par l'architecture du réseau Internet, les messages envoyés peuvent emprunter des chemins différents pour arriver à un même endroit. Ils peuvent arriver dans un ordre différent de celui dans lequel ils ont été envoyés, ou être perdus et ne jamais arriver à destination. C'est pour cette raison que le protocole* TCP à été inventé. Le protocole* TCP inclut des routines de vérification assurant que les messages arrivent toujours et dans le bon ordre. Un message n'arrivant pas doit être réémis, un message arrivant trop tôt attend l'arrivée effective des messages précédents avant d'être pris en considération. TCP utilise aussi un mode connecté. Cela signifie qu'une connexion doit être établie être les deux interlocuteurs pour que les messages puissent circuler. Une fois la connexion établie, les messages peuvent circuler librement, sans offuscation des systèmes de protection, car les interlocuteurs se connaissent. L'établissement de la connexion fait en quelque sorte foi de leur droit à communiquer entre eux. De plus, la communication est très aisée : on utilise la connexion comme un fichier où il suffit à l'un d'écrire pour que l'autre puisse lire dessus. La lecture et écriture de fichiers existe depuis les débuts de l'informatique, c'est un procédé maitrisé et simple.[12][13]

Le protocole* UDP quant à lui n'assure aucune vérification, et utilise un mode déconnecté. Un paquet peut ne jamais arriver, ou ne pas arriver dans le bon ordre. Le mode déconnecté implique de connaitre en permanence les informations du ou des interlocuteurs avec lesquels on souhaite communiquer, c'est-à-dire l'adresse IP et le port. Ce mode implique également que chaque paquet soit envoyé directement et individuellement. Il n'est ainsi plus question d'écrire et de lire un fichier, mais bien d'envoyer un message ou d'écouter la réception de celui-ci. [12][13]

Nous utiliserons donc UDP.

En effet, le protocole* TCP oblige la réémission de chaque message qui n'arrive pas, et l'attente de chaque message qui arrive en retard par rapport aux autres. S'il s'agit de transférer trente fois par seconde les coordonnées d'un objet, il nous importe peu de récupérer la vingt-septième position perdue si c'est la cent trente cinquième qui est d'actualité. Les données perdues ou retardataires n'ont plus la moindre utilité, et leur réémission ou leur attente est une perte de temps et de bande-passante qui vient diminuer la fluidité des échanges. Pour faire fonctionner ce système, TCP attend de plus une réponse d'acquittement (ACK) pour chaque message émis. C'est l'absence de cette réponse dans un certain laps de temps qui indique qu'il

faut réémettre le message. La plupart du temps, les ACK sont envoyés en réponse et la réémission n'est pas nécessaire, cependant, les messages d'acquittement viennent s'ajouter à la bande-passante consommée par le protocole* même lorsque cette vérification n'est pas voulue ou nécessaire. Il faut ajouter à cela la tendance que peuvent avoir les implémentations de TCP à stocker un certain volume de données avant d'envoyer le message pour réduire le nombre de messages. Un fois de plus, ce comportement ne correspond pas à nos besoins en réactivité pour des jeux d'action. [12][13]

Pourtant, il est parfois nécessaire que certaines informations arrivent avec fiabilité et/ou dans le bon ordre. Pour cela, il est en fait tout à fait possible d'implémenter une surcouche logicielle à UDP pour gérer, si nécessaire, l'ordonnancement et la réémission des messages perdus. Un grand nombre de librairies existent dans tous les langages de programmation supportant UDP pour gérer ces fonctionnalités. Par conséquent, nous utiliserons une de ces librairies pour nos développements. Enfin, si UDP peut laisser des pertes ou permettre le désordre des messages, la plupart des messages arriveront et seront dans le bon ordre. Le risque existe mais il n'est pas si grand.[8][13]

Nous pourrions aussi être tentés d'utiliser les deux protocoles* conjointement. UDP lorsqu'on a besoin de réactivité et de l'information la plus à jour uniquement, et TCP pour transmettre les informations vitales qui doivent à tout prix arriver. L'idée n'est pas aussi bonne qu'elle le parait, car en réalité comme TCP et UDP sont tous les deux basés sur IP, et il existe une corrélation entre les deux qui fait que l'emploi de l'un va affecter le comportement de l'autre. Pour être plus précis, dans certains cas, l'emploi de TCP tend à augmenter le taux de perte de messages envoyés par UDP. Par conséquent, utiliser UDP avec une surcouche assurant la fiabilité de l'information reste la solution répondant le plus précisément à nos besoins. [14]

2. Méthodes d'optimisation

La conception générale du programme étant maintenant définie, nous allons pouvoir y intégrer les optimisations nécessaires à la fluidité du programme.

Comme les ressources disponibles sur Internet pour transférer des données sont limitées, nous allons devoir diminuer au maximum le besoin en ressources de notre programme. Ce besoin peut être calculé à l'aide de « l'équation du principe de l'information » émise par Singhal et Zyda.[5]

$$Ressources\ Nécessaires = M \times H \times B \times T \times P$$

Équation 1 - Equation du principe de l'information

Cette équation se compose des termes :

- M le nombre de messages transmis,
- H le nombre moyen de nœuds recevant le message,
- B la quantité moyenne de bande passante nécessaire pour chaque message,
- T la rapidité de prise en charge nécessaire pour le message (Une grande valeur de T nécessite un délai de traitement très court),
- et P le nombre de cycles processeurs nécessaires pour recevoir et traiter chaque message.

Si la quantité de ressources réseau nécessaire ne dépasse jamais le palier des ressources disponibles, l'application fonctionne de façon stable. De plus, il existe plusieurs façons d'obtenir un même résultat avec des paramètres différents. Cette formule peut donc être utilisée pour privilégier certains aspects : temps de latence*, fréquence de mise à jour des messages, nombre de clients supportés par un même serveur, etc. On peut décider de privilégier le nombre de clients par serveur en faisant monter H par exemple (plus économique). Cependant cela devra se faire au détriment de la fluidité en réduisant le nombre de messages envoyés ou en autorisant des délais de traitements plus longs par exemple (diminution de M et T) pour ne pas dépasser les ressources disponibles.

2.1. Compression

Quoi qu'il arrive, les messages envoyés et reçus sur le réseau nécessitent un certain temps de traitement au niveau des nœuds. Il nous est possible de réduire le trafic en augmentant légèrement le temps de traitement et en diminuant la quantité ou la taille des messages.

2.1.1. Compression des messages

Réduire la taille de données est le rôle de la compression. Il est en effet possible de diminuer la taille d'un flux de données en le compressant avant l'envoi, puis en le décompressant à la réception. L'inconvénient est bien entendu une augmentation du temps de calcul ou de la puissance nécessaire pour traiter les messages.

Dans l'équation du principe de l'information, il s'agit donc de faire baisser B (la taille moyenne d'un message) et de faire monter P (le temps de traitement). Pour que cela fonctionne, il faudra que B soit autant diminué que P est augmenté. Si la baisse de B ne compense pas l'augmentation de P, alors la méthode de compression employée est inutile voire gênante.

Il existe deux types de compressions : avec et sans perte. La compression sans perte permet de récupérer l'information intégralement, alors que la compression avec perte est réalisée en détruisant une partie de l'information. Une compression avec perte se révèle souvent bien plus efficace, du fait de la diminution réelle de la quantité d'informations. Aussi, certaines informations numériques peuvent être simplifiées pour réduire la quantité de données envoyées. Par exemple, imaginons une variable qui représente un angle, exprimé en radians, mémorisée sous forme d'un nombre à virgule flottante à double précision (type « double » ou « long » dans un langage de programmation). Cette variable peut être « arrondie » à un nombre à virgule flottante à précision simple (type « float ») sans gêner le bon déroulement du jeu qui ne nécessite pas un affichage précis mais cohérent. Cette perte de précision n'est pas visible pour le joueur et l'opération permet de gagner un octet par envoi de cette variable. Il est important que les calculs soient faits à partir de cette valeur arrondie à la fois sur le client et sur le serveur, car la perte d'information en soit n'est pas gênante mais une différence de traitement entre client et serveur peut l'être. Ainsi chacune des deux applications devra

arrondir de la même façon et avant les calculs pour pouvoir ensuite transférer les valeurs réellement utilisées dans les calculs.

Pour chacun de ces deux types de compression, on peut aussi envisager deux niveaux. Jouni Smed les décrit comme interne et externe. [6]

Le niveau interne correspond à chaque message pris individuellement. On cherche alors à compresser le message et à éliminer ses redondances internes. Dans le cas d'une compression avec perte, il peut s'agir de filtrer les informations moins utiles ou de réduire le niveau de détail de l'information. Ce niveau de compression correspond parfaitement à l'envoi de données non fiables (avec UDP, sans vérifications) puisque l'information sort de tout contexte.

Si l'information envoyée est fiable (grâce à TCP ou à une surcouche de vérification d'UDP), le niveau externe peut également être utilisé. La compression externe réutilise des informations qui ont déjà été envoyées. Par exemple, envoyer des variations de valeurs plutôt que les valeurs absolues est plus avantageux car la taille en bits d'une variation est souvent moins importante. Un message peut aussi faire référence à un message précédemment envoyé pour reprendre tout ou partie des informations.

2.1.2. Agrégation des messages

Pour gagner de la bande passante, agréger des messages est une solution. Le principe est de regrouper plusieurs messages en un seul envoi pour diminuer la fréquence et ainsi ne pas répéter les entêtes du protocole*. Pour reprendre l'Équation 1 page 21 (du principe de l'information) : on diminue M (le nombre de messages) et T (le délai de traitement d'un message) en factorisant plusieurs messages en un seul. Ainsi B (la taille moyenne d'un message) augmente mais la bande passante utilisée est globalement moins importante, au prix d'une légère augmentation de P (le temps de traitement des messages).

Ce procédé nécessite d'instaurer des critères pour déterminer quels messages vont être agrégés et à quel moment. La première approche consiste à agréger les messages selon leur date d'expiration. On affecte à chaque message une date avant laquelle il doit être envoyé. Tous les messages non-envoyés et dont la date d'expiration n'est pas encore dépassée sont rassemblés puis envoyés sous un même message. L'inconvénient de cette méthode est que le

gain peut être nul si les messages sont créés un à un et que les envois correspondent au rythme de création. Une autre possibilité est l'approche par quorum : un nombre minimum de messages à agréger est défini, le lot de messages part lorsque ce nombre est atteint. L'inconvénient est qu'il n'existe aucune garantie quant au délai de transmission. Ainsi le gain en bande passante est très facile à prévoir, mais la latence engendrée peut gêner fortement le jeu. Il est également possible de combiner les deux approches en envoyant un lot de messages à la date d'expiration ou dès qu'un certain nombre de messages est en attente d'envoi.

2.2. Estimation des positions

Inspirées de la navigation, des techniques d'estimations des positions sont applicables au jeu vidéo. Elles permettent par exemple de calculer des positions intermédiaires dans les courbes de trajectoire pour rendre le mouvement plus fluide, mais aussi de prévoir quelle sera la position d'un objet dans un temps futur.

En outre, l'intérêt majeur du calcul de positions intermédiaires est de réduire l'utilisation de la bande passante en envoyant moins d'informations et en calculant celles qui manquent. Pour reprendre l'équation 1 : Le nombre de message (M) diminue mais le temps de calcul nécessaire (P) augmente.

Toujours dans les termes de l'équation, la prévision des positions permet de son côté de réduire le besoin en rapidité de traitement du message (T). En effet, les informations manquantes sont calculées ce qui permet un affichage fluide avec moins de données. Cela se fait aussi au détriment du temps nécessaire à traiter le message (P). De plus, le calcul des positions n'est pas le seul à mettre en place pour cette méthode. En effet, au moment où le message de mise à jour de la position arrive, il se peut que la position réelle soit différente de celle prévue. Il faut alors remplacer la position prévue par la position réelle ce qui peut provoquer des incohérences visuelles (téléportation de l'objet par exemple). On applique alors des fonctions de convergence pour rectifier discrètement l'erreur de prédiction sur le temps et se rapprocher des valeurs réelles.

2.2.1. Prédiction

La technique de prédiction des positions la plus courante consiste à utiliser des dérivées polynomiales à partir de la position de l'objet dont on veut prédire la trajectoire. En effet, les

deux premières dérivées de la position peuvent être interprétées naturellement comme étant la vitesse et l'accélération de l'objet. Il est par exemple possible d'utiliser la dérivée suivante d'ordre un, prenant en compte la position et la vitesse de l'objet : [6][8]

$$Position_{t1} = Position_{t0} + v_{t0} * (t1 - t0)$$

Équation 2 - Dérivée de polynôme d'ordre un

Cette première fonction de prédiction calcule la position au temps t1 à partir de la position au temps t0 en lui ajoutant le vecteur de vitesse à t0 multiplié par la durée entre t0 et t1.

Afin d'améliorer la précision de la prédiction, on peut utiliser une dérivée d'ordre deux qui inclut aussi l'accélération notée a. On obtient alors la fonction suivante :

$$Position_{t1} = Position_{t0} + v_{t0} * (t1 - t0) + \frac{1}{2} a_{t0} * (t1 - t0)^2$$

Équation 3 - Dérivée de polynôme d'ordre deux

Cette dérivée d'ordre deux est plus précise pour des déplacements de véhicules par exemple, mais l'Équation 2 de premier ordre convient mieux à des déplacements moins faciles à prédire tels que ceux d'un personnage humain. En effet, plus le terme est d'ordre important, et plus il sera sensibles aux imprécisions car une petite erreur peut exagérer rapidement les prédictions de mouvements. Dans le cas d'un déplacement humain, l'accélération est faible et très variable. Le mouvement est donc impossible à prédire précisément. Utiliser une dérivée de premier degré est par conséquent une approximation moins mauvaise car moins sensible aux imprécisions.

Il est aussi possible de choisir dynamiquement la fonction de prédiction à utiliser. Ainsi, le serveur et les clients utiliseront une dérivée d'ordre un si l'accélération est très variable, d'ordre deux dans le cas contraire. On cherche ainsi à réduire automatiquement l'imprécision pouvant être provoquée par le terme de l'accélération.

Les informations sur les positions sont obtenues directement du serveur. Pour obtenir les informations nécessaires à la prédiction (vitesse et accélération), deux méthodes sont possibles. La première façon de faire est de transmettre ces données de la même façon que la

position. À chaque mise à jour, le serveur envoie alors la vitesse et l'accélération en plus des positions. Cette méthode est la plus précise car on obtient l'information la plus à jour en provenance du serveur. La seconde façon de faire consiste à utiliser les 3 dernières positions de l'objet pour calculer la vitesse et l'accélération au moment de la dernière position.[5]

Une troisième approche est également possible : réaliser une prédiction spécifique à chaque objet. En effet, utiliser une dérivée ne prend pas en compte les capacités réelles de l'objet, ni sa façon de se déplacer ou les règles auxquelles il est soumis. Ainsi, il est possible de développer des algorithmes de prédiction spécifiques basés sur le comportement des objets. L'inconvénient de cette méthode est le temps nécessaire à développer et maintenir ces algorithmes.

Nous avons vu en introduction que la prédiction de positions permettait de réduire le besoin en rapidité de traitement des messages, car les objets peuvent être affichés même si les données de mise à jour ne sont pas encore reçues. La prédiction permettra également de réduire le nombre de messages envoyés. En effet, il est possible de n'envoyer les informations de mise à jour aux clients que lorsque l'erreur entre la position réelle et la position calculée est trop importante. Cela nécessite de réaliser la même prévision côté serveur (pour mesurer l'erreur) et côté client, mais économise énormément de bande passante. Le niveau d'erreur toléré peut même être adapté en fonction du contexte : on peut imaginer qu'un objet loin du joueur n'ait pas besoin d'être affiché très précisément et qu'une imprécision plus importante soit tolérable.

2.2.2. Convergence

Avec les prédictions, il arrive que la position prédite ne corresponde pas à la position réelle. Un message de mise à jour du serveur vient donc corriger l'erreur sur les clients en annonçant la position réelle. Si on positionne directement l'objet à sa position réelle, un effet de téléportation sera perceptible à l'écran. Nous ne voulons pas d'un tel effet visuel, tout à fait incohérent avec le déplacement logique de l'objet. En outre, il existe des stratégies de correction des positions consistant à faire converger la position estimée vers la position réelle. Son objectif est d'être rapide et imperceptible, de sorte à dénaturer le jeu le moins possible.

Une des meilleures techniques consiste à utiliser un point de convergence situé sur le chemin prévisionnel de l'objet, de sorte que la position affichée de l'objet coïncide avec sa position estimée à l'issue de la période de convergence. [6] Il ne reste ensuite qu'à afficher le déplacement de l'objet depuis sa position actuelle affichée vers la prochaine destination prévue. Après, l'objet suit son chemin prévisionnel jusqu'à ce qu'un message de mise à jour indique une erreur de prévision, et provoque l'exécution d'une nouvelle convergence depuis la position d'affichage actuelle vers un prochain point de prédiction prenant en compte les données à jour.

Le déplacement vers le point de convergence peut se faire de plusieurs manières. La solution la plus simple consiste à réaliser une interpolation linéaire (l'objet se déplace droit vers le point). Même si cette solution est mieux qu'une téléportation de l'objet, le comportement de l'objet reste souvent peu naturel. Aussi, il est possible d'utiliser des fonctions d'interpolation plus élaborée. Une méthode consiste à prendre 4 points : celui de la précédente position affichée de l'objet, celui de la position actuelle de l'objet, le point de convergence et le point prévu après le point de convergence. A partir de ces points, on peut réaliser une courbe cubique offrant un mouvement plus souple.

Le schéma ci-dessous montre un exemple de convergence d'une prévision de chemin vers une nouvelle. L'instant t0 est le même pour les deux courbes, ensuite t1 diffère mais l'information n'est connue qu'en t2. C'est à ce moment que le client doit recalculer une nouvelle prévision à partir de cette information, puis faire converger la position actuellement vers un point de convergence situé plus loin sur la nouvelle prédiction. Ici la durée de la convergence est de 1, elle se réalise de t2 à t3. Les ronds rouges sont les positions utilisées pour calculer la courbe cubique.

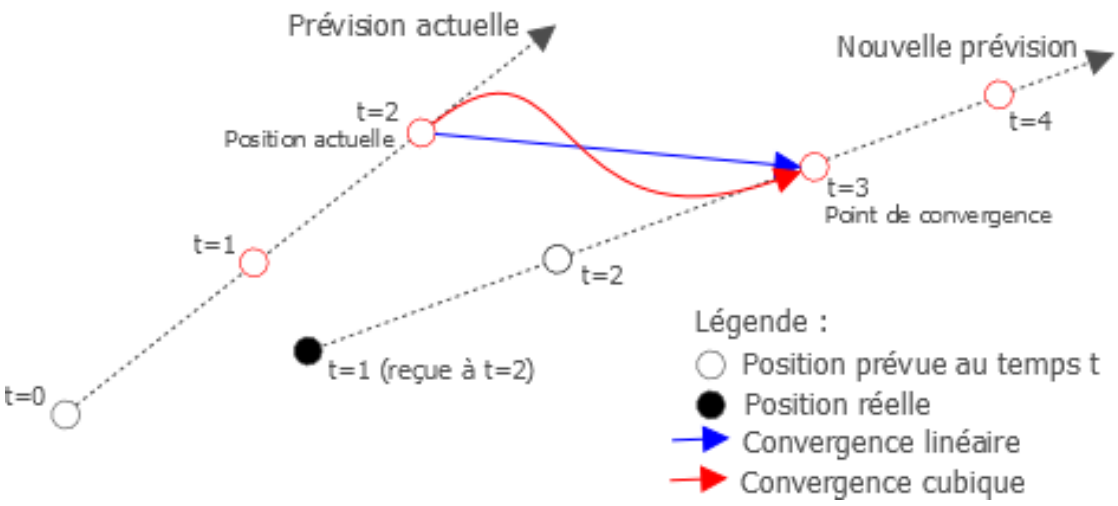

Figure 5 - convergences linéaires et cubiques [source personnelle]

L'interpolation est réalisée par le client et peut être paramétrée pour se réaliser sur un temps plus ou moins long, le plus souvent entre 0 et 500ms. Une interpolation de 0 milliseconde provoquera la téléportation immédiate de l'objet à sa nouvelle position (peu recommandée) et une interpolation de 500ms provoquera la rectification de la position de l'objet pendant cette durée. La convergence est purement visuelle (les données utilisées par les programmes pour les calculs sont les données corrigées les plus à jour). Ainsi une interpolation plus longue est moins perceptible visuellement, mais provoque un décalage entre les données réelles et les données affichées.

Dans son moteur, Valve[3] prend en considération le temps d'interpolation du client pour calculer la position affichée des objets sur l'écran du joueur. Ainsi le décalage entre les données réelles et la vue interpolée ne pose plus problème. Cependant, cela peut avantager les joueurs configurant une interpolation plus rapide, car ils pourront voir leurs adversaires apparaitre à l'écran quelques millisecondes en avance. En effet, dans le cas de Valve, l'interpolation n'est pas seulement utilisée en cas d'erreur de prédiction mais en permanence. Ainsi plusieurs positions entre la position actuelle et la position estimée sont calculées par interpolation pour rendre le mouvement plus fluide avec moins de positions connues. [16]

[3] Valve est un studio indépendant de développement de jeux vidéo. Les titres de Valve les plus connus sont Counter-Strike, Half-Life et récemment Left 4 Dead. Ils sont basés sur leur moteur « Source ».

2.3. Compensation les temps de latence

Il existe plusieurs façons d'appliquer ces outils pour compenser les temps de latence. Il n'y a pas une bonne façon de faire mais plusieurs façons qui ont leurs avantages et inconvénients. Un principe de base peut tout de même être dégagé.[8][9][11][12]

Tout d'abord, le client anticipe la position des entités contrôlées par le joueur local grâce aux commandes utilisées par ce joueur. Il s'agit d'une prédiction côté client qui s'avère très précise car le client connait les commandes du joueur. Ainsi, en appliquant les mêmes calculs de trajectoires et de collisions que le serveur, il n'y a presque aucune correction à faire sur ces prédictions, sauf pour les rares cas où le client ne ferait pas les calculs exactement de la même façon que le serveur. Une différence de calcul avec des même algorithmes peut arriver à cause de l'architecture matérielle (toutes des architectures n'ont pas forcément le même degré de précision de calcul) ou à cause de l'environnement de jeu qui peut être différent sur le serveur et sur le client au même instant (par exemple si le serveur sait qu'un autre joueur bloque le passage et empêche le déplacement, mais que le client ne le sait pas encore et prévoit d'avancer).

Après avoir reçu les commandes du client, le serveur les applique à l'environnement tel qu'il était au moment où la commande a eu lieu. Il doit donc sauvegarder des états antérieurs de l'environnement, et être capable de rejouer la scène en prenant en compte de nouveaux éléments.

De ce fait, le client attend la confirmation du serveur pour les actions importantes car elles ne sont définitives qu'au moment où tous les clients ont dépassé ce temps. En effet, tant qu'un client peut encore agir sur l'environnement dans une plage de temps pour laquelle les autres joueurs ont déjà joué, il peut encore empêcher ces autres joueurs de réussir leur action. Le serveur confirme donc avec retard certaines informations de réussite de l'action (perte de points de vie, etc.).

Sur ce concept, on peut rajouter un certains nombre de réglages optionnels. Par exemple, il est intéressant de définir une durée maximum de temps au-delà de laquelle les commandes des joueurs ne sont plus acceptées. En effet, il est coûteux en ressource de stocker un grand nombre d'états antérieurs pour pouvoir rejouer, et autoriser des modifications trop lointaines

dans le passé pourrait être une faille exploitée pour tricher. Le moteur Source créé par Valve, utilisé pour le jeu half-life ou encore left 4 dead, mémorise et rejoue jusqu'à 2 secondes d'historique par exemple.

Concernant l'historique, deux façons de faire existent. Il est possible de retenir un état global dans le passé, qui sera utilisé pour rejouer les actions des joueurs dans le passé de façon incrémentale. Chaque commande du passé modifie alors cet état qui aura déjà pu être modifié en cet instant par une autre commande. L'autre solution consiste à mémoriser la liste des états envoyés à chaque client indépendamment. Avec cette dernière solution, le serveur peut vérifier l'état spécifique dans le passé pour chaque client, sans qu'il soit modifié par les actions d'autres joueurs. Ainsi la précision est plus grande pour le joueur qui, dans un FPS par exemple, est sûr de toucher son adversaire en visant directement sur lui. En effet, le serveur va considérer que l'adversaire était à l'endroit du tir pour le joueur au moment du tir (s'il a bien visé). Dans le cas d'un état général pour tout les joueurs, cet état peut être altéré entre temps par un joueur tiers et empêcher ce tir (que le joueur ayant tiré voyait possible sur son écran). Ainsi, un état global est plus « réaliste » car, par exemple, il ne permet pas à des joueurs de s'entretuer s'ils ont tirés l'un sur l'autre durant une même période de latence : le premier joueur à avoir tiré modifie l'état, le second étant mort il ne peut plus répliquer. La même situation avec des états spécifiques pour chaque joueur aurait entrainé la mort des deux joueurs, ce qui est moins réaliste d'un point de vue général, mais plus cohérent dans la vision du joueur mort qui aurait vu son tir se déclencher sans effet.

2.4. Filtre par zone d'intérêt

Le principe du filtre par zone d'intérêt est de ne pas transmettre au client des informations qu'il n'a pas besoin d'afficher. Par exemple, si un objet se déplace dans le dos de la caméra du client ou de l'autre côté d'un mur opaque par rapport à la caméra, alors cet objet ne peut pas être vu. Ainsi, il est inutile de transmettre des informations le concernant et on peut ainsi économiser de la bande passante.

Cette optimisation nécessite de la prudence, car plusieurs éléments doivent être pris en compte. La vue est un critère, mais un objet peut aussi être entendu, et si le mur ne laisse pas passer la lumière il peut laisser passer le son. Il est alors important de transmettre les informations sur la position de l'objet pour que les sons qu'il produit puissent être joués sur le client.

Cette optimisation apporte également un avantage en termes de sécurité. En effet, il est possible pour quelqu'un maitrisant les réseaux informatique d'intercepter l'information contenue dans le message reçu sans passer par le client de jeu. S'il parvient à déchiffrer les données envoyées, alors il obtient les informations de la position d'objets. Ainsi, certains pirates parviennent à modifier le client de jeu pour y intégrer des informations supplémentaires telles que les positions de joueurs sensés être hors de vue. Cette forme de triche est cependant minimisée par les filtres par zone d'intérêt. En effet, puisque les données non accessibles ne sont pas envoyées, il est impossible de tricher en les obtenant directement au niveau de la carte réseau.

Nous venons d'appréhender plusieurs méthodes pour optimiser les échanges de données, l'objectif à présent est de parvenir à rendre ces optimisations génériques et réutilisables dans le contexte général d'un jeu multi-joueurs en temps réel.

Chapitre 2 – Généralisation des optimisations

Pour rendre ces optimisations génériques et réutilisables dans le contexte général d'un jeu multi-joueurs en temps réel, une première étape consistera à dégager l'architecture générale d'un jeu pour ensuite voir à quel niveau vont s'intégrer les différentes optimisations. Nous en déduirons les interfaces de programmation que devront implémenter les différentes classes du programme pour intégrer les optimisations.

1. Architecture générale d'un jeu

Pour répondre aux mêmes besoins d'un jeu, il peut exister plusieurs façons de concevoir l'architecture générale. De nouvelles contraintes peuvent même venir s'ajouter avec le développement récent des processeurs multi-cœurs. Ainsi, il convient de nos jours de prévoir des architectures capables de gérer les unités de calculs effectuant des traitements parallèles, en rendant les traitements parallélisables.

Le sujet étant complexe, nous utiliserons dans ce document une architecture séquentielle, plus simple à implémenter. Toute autre architecture sera également utilisable dans la mesure où nos optimisations seront découplées du contexte de jeu à l'aide d'interfaces de programmation.

1.1. Architecture de l'application client/serveur

En dehors des dépendances vers des moteurs tiers (tels que le moteur réseau, le moteur physique, le moteur de rendu sonore), l'application sera découpée en trois grandes parties : le programme client, le programme serveur, et les librairies partagées.

Certaines optimisations nécessitent l'exécution de code identique sur le serveur et sur le client. De plus, ces deux programmes seront amenés à utiliser des structures de données communes. Ils vont en effet manipuler les mêmes objets, dans le même environnement et la seule différence concerne éventuellement le niveau de connaissance des données. Ainsi, il convient de centraliser les optimisations dans une bibliothèque de classes qui sera partagée entre le client et le serveur.

Le client et le serveur seront aussi amenés à manipuler des données sur les objets qui ne concernent pas l'autre comme des informations graphiques pour le client ou l'identifiant d'un objet en base de données pour le serveur. Cependant, il n'est pas gênant que certaines informations de la structure partagée ne soient pas utilisées par l'un ou par l'autre des programmes. De plus, pour affirmer une certaine indépendance du client et du serveur, il est possible à chacun d'étendre la classe partagée de base pour la spécialiser selon ses besoins.

Le client et le serveur auront ensuite des dépendances communes vers d'autres librairies : les moteurs. En effet, il est probable que la physique soit anticipée par le client et vérifiée par le serveur. Il est alors préférable que les programmes utilisent la même librairie de calculs physiques. De même, utiliser une librairie réseau commune permet d'assurer la cohérence des communications entre le client et le serveur et facilite grandement le développement.

Ainsi, on peut déjà dégager le schéma de dépendances suivant :

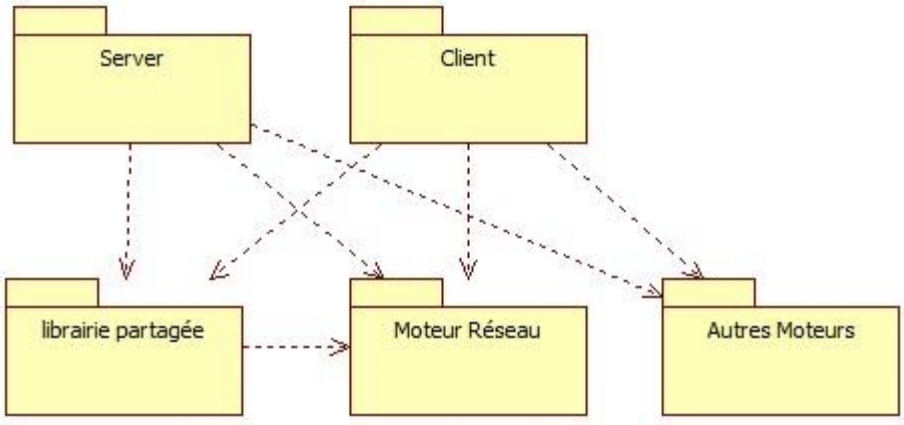

Figure 6 - Diagramme des dépendances pour un jeu en réseau [Source Personnelle]

1.2. L'environnement de jeu

La librairie partagée contiendra l'ensemble des classes et éléments dont le serveur et le client auront besoin. Cela comprend l'environnement de jeu. Dans la plupart des jeux, l'environnement est l'ensemble des éléments qui peuvent interagir, évoluer et être représentés à l'écran du joueur. L'environnement inclut les décors fixes (murs, sol, ciel), des décors modifiables (portes, caisses fragiles) et les personnages contrôlés par les joueurs. Il peut évoluer par plusieurs moyens : par la répercussion des ordres des joueurs sur les éléments contrôlés, par le scénario de jeu (des scripts exécutés à un instant donné), et par réaction à une autre évolution (typiquement les déplacements calculés par le moteur physique).

D'un point de vue modélisation, nous aurons donc besoin d'un état de l'environnement qui contient tous les éléments cités précédemment. Voici à quoi peut ressembler la classe d'état de l'environnement :

Figure 7 – Modélisation d'un état de l'environnement de jeu [Source Personnelle]

« StaticObject » est la classe d'un objet dont l'état ne change pas au cours de la partie. « MovingObject » est au contraire la classe d'un objet susceptible de se déplacer.

1.3. Mise en place des optimisations du protocole

Si la plupart des optimisations décrites dans le Chapitres 1 peuvent être mises en place au niveau de certains objets uniquement, l'utilisation d'UDP intervient à un niveau plus global.

Nous avons vu qu'UDP devait être préféré à TCP pour sa capacité à envoyer des informations sans que celles-ci ne soient systématiquement réémises en cas de perte. Envoyer des messages fiables reste tout de même nécessaire, et il convient d'utiliser une surcouche logicielle pour s'assurer de la bonne réception des données. Cette surcouche trouvera sa place dans une librairie réseau. Cette librairie ne devra avoir aucune dépendance avec le reste de l'application et être parfaitement indépendante. Bien qu'elle soit un pilier pour un jeu réseau, elle ne nécessitera presque aucun codage, à moins de besoins très spécifiques, car des librairies réseaux basées sur UDP peuvent être trouvées pour de nombreux langages, certaines ayant même des licences gratuites pour une utilisation commerciale. De plus, le développement de telles librairies est extrêmement long et difficile, et il est plus sûr de s'orienter vers des solutions éprouvées.

Nous retiendrons que l'implémentation du protocole* réseau est isolée dans une librairie réseau, et qu'elle sera utilisée par le client, le serveur, et éventuellement leur librairie partagée.

Les librairies réseaux gèrent parfois directement la compression des messages au niveau interne. C'est-à-dire qu'elles s'occupent de compresser et décompresser les messages individuellement au moment de l'envoi et de la réception. La compression de messages courts étant peu efficace, on préfèrera l'ignorer et réduire soi-même la précision des données manipulées pour qu'elles prennent moins de place. Nous privilégierons donc toujours des types encodés sur moins de bits avec une précision plus approximative tant qu'elle reste suffisante. Ce choix se fera au cas par cas lors des déclarations des objets et des valeurs à transmettre.

2. Modélisation des optimisations

Les autres optimisations peuvent être modélisées plus facilement. Voyons à présent une façon de faire pour chacune d'entre elle.

2.1. Modélisation de l'estimation des positions

L'estimation des positions nécessite de pouvoir anticiper la position de n'importe quel objet mobile. Ainsi, nous pouvons déjà en déduire que nous aurons besoin d'une interface pour piloter ce type d'objet. Il aura au minium une position (modélisable sous forme d'un vecteur), mais aussi une vitesse, une accélération et un angle qui peuvent également être représentées sous forme de vecteurs. L'angle correspond à l'orientation de l'objet et peut être différent de son sens de déplacement (indiqué par la vitesse).

Une façon logique d'implémenter l'estimation des positions est de faire hériter nos objets mobiles spécifiques d'une classe abstraite. Cette classe commune contient les différentes informations dont nous avons besoin pour gérer le déplacement des objets, ainsi qu'une implémentation générique de la méthode servant à obtenir l'état suivant par estimation. Cette méthode pourra ensuite être surchargée par chaque classe pour implémenter ses spécificités.

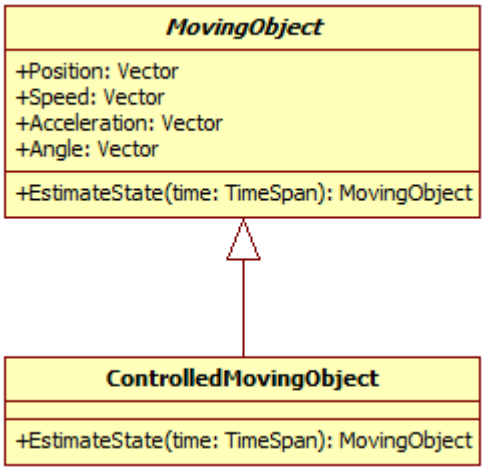

Figure 8 - Modélisation d'un objet implémentant l'estimation des positions [Source Personnelle]

Dans la Figure 8, un « MovingObject » abstrait propose une implémentation par défaut de « EstimateState ». Cependant, un objet contrôlé par le joueur a un comportement qui peut être prédit différemment en utilisant les commandes du joueur. Ainsi,

« ControlledMovingObject » surcharge la méthode « EstimateSate » pour utiliser les commandes et améliorer sa précision.

La méthode « EstimateState » prend en paramètre la durée entre l'état actuel et le prochain état à prédire, et retourne une copie de l'objet dans son état futur estimé.

2.2. Modélisation de la convergence

La convergence de l'objet dans son état actuel vers son état réel (après rectification des estimations) se fait à l'aide d'une méthode d'interpolation. La convergence est purement visuelle (les données utilisées par les programmes pour les calculs sont les données corrigées les plus à jour). L'interpolation se fait donc à partir de deux positions et la méthode calcule une position intermédiaire en fonction du temps actuel et de la durée totale de l'interpolation. Notre MovingObject se voit donc enrichi d'une nouvelle méthode « Interpolate » :

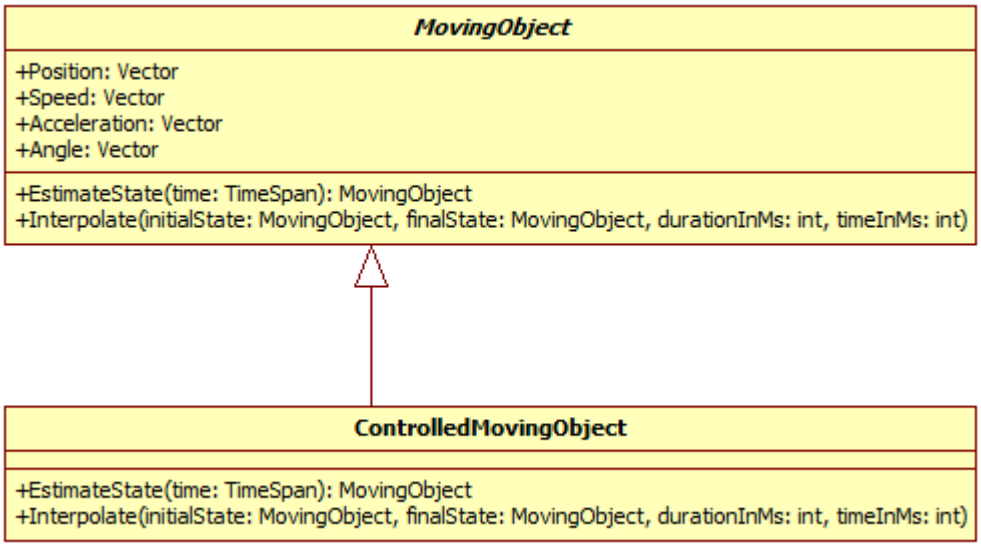

Figure 9 – Modélisation d'un objet implémentant la convergence [Source Personnelle]

La méthode « Interpolate » prend quatre paramètres. Les deux premiers sont les positions initiale et finale de l'objet. On souhaitera interpoler de l'une vers l'autre. Les deux paramètres suivant sont la durée totale et le temps écoulé depuis le début de l'interpolation. Ces deux paramètres permettront de situer l'avancement de l'objet dans le mouvement complet,

l'interpolation est finie lorsque la valeur de timeInMs (temps écoulé depuis le début de l'interpolation) rejoint celle de durationInMs (durée total).

De la même manière que précédemment, on pourra implémenter par défaut une convergence linéaire dans « MovingObject » et surcharger la méthode « Interpolate » dans une classe spécialisée pour avoir une interpolation cubique par exemple.

Si elle est prévue pour rendre l'affichage sur le client plus fluide, cette méthode doit également être connue du serveur pour savoir ce que voit le client. De cette manière, on pourra calculer l'impact d'un tir ou la position d'un corps de manière plus cohérente avec ce que voit réellement le joueur.

2.3. Modélisation de la compensation des temps de latence

La compensation des temps de latence se fait au niveau du serveur en recréant l'environnement de jeu du client au moment où sa commande a été émise. Ainsi, il est nécessaire de mettre en place les mécanismes pour garder en mémoire les historiques d'un environnement, et permettre la modification de cet environnement. Il existe plusieurs façons de le faire, mais basées sur les mêmes outils. Nous allons donc veiller à modéliser ces outils pour qu'ils puissent être utilisés dans le sens souhaité.

Pour archiver et dater différents états d'un environnement, nous avons tout d'abord besoin d'une classe qui associe une copie d'une instance à une date donnée, et qui puisse retrouver les états antérieurs d'un objet.

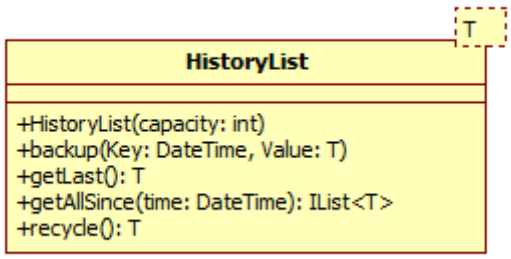

Figure 10 – Modélisation de la classe d'historisation des états [Source Personnelle]

La classe « HistoryList » présentée ci-dessus est paramétrée. Cela signifie que nous pourrons mémoriser un historique pour une classe T que nous lui indiquerons. Le constructeur prend en

paramètre le nombre maximum de valeurs que peut mémoriser l'historique. Il est préférable que cette longueur soit fixe pour optimiser l'utilisation de la mémoire grâce à la méthode « recycle ». Cette dernière permet de récupérer une instance de T qui a déjà été utilisée dans l'historique mais n'en fait plus partie. Réutiliser l'espace mémoire alloué par les instances déjà mémorisées permet de ne pas avoir à allouer à nouveau de l'espace pour les prochaines instances qui sont créées et améliore grandement les performances. En effet, l'allocation de mémoire est une opération coûteuse en temps dans un programme, et nous aurons en permanence besoin de nouvelles instances pour stocker les états successifs des objets au fur et à mesure du déroulement de la partie. Enfin, cette classe définit les méthodes « getLast » et « getAllSince » qui permettent de récupérer le où les derniers états à partir d'une date. Ces méthodes nous permettront respectivement d'envoyer aux clients les dernières informations à jours, et de rejouer tous les états de l'environnement suite à une modification dans le passé.

Aussi, nous devrons avoir la possibilité de rejouer plusieurs commandes, ce qui est la deuxième étape de la compensation des temps de latence. La commande est prise en compte au moment où elle a été émise, et entraine des répercutions sur les états suivants. Nous créerons une méthode « replay » permettant à partir d'un état de l'environnement et d'une liste de commandes de recréer le parcours des entités contrôlées par le joueur ayant émis les commandes. Cette méthode nécessite la connaissance des états de l'environnement complet pour chaque objet à rejouer.

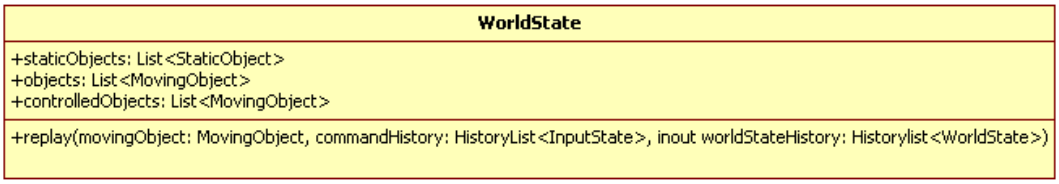

Figure 11 – Modélisation de l'environnement de jeu [Source Personnelle]

« Replay » prend trois paramètres. Le premier est l'objet dont on veut altérer la position dans l'état de l'environnement. Le second est l'historique des états de l'environnement. En effet, la méthode devra altérer les états suivants en fonction des commandes. Le troisième paramètre est l'historique des commandes à appliquer à l'environnement. La méthode peut être assez

difficile à implémenter en fonction de la complexité de l'environnement et des interactions entre les acteurs. En effet, si la commande a des conséquences sur de nombreux objets de l'environnement, les modifications à prendre en compte seront nombreuses.

2.4. Modélisation du filtre par zone d'intérêt

Filtrer les données à envoyer en fonction de leur apparition pour un joueur donné nécessite la connaissance complète de l'environnement de jeu. Pour cette raison, cette optimisation doit être calculée par le serveur. Ce dernier doit par conséquent implémenter une méthode lui permettant de déterminer si un objet est visible par le joueur ou non. Cette méthode sera très spécifique à l'environnement de jeu et devra être implémentée ou surchargée en fonction du jeu. L'environnement pourra donc indiquer si 2 objets sont capables de se voir, s'entendre, etc. à l'aide de la méthode « canInteract » :

La méthode prend deux paramètres qui sont l'objet à partir duquel on se positionne, et l'objet dont on souhaite savoir s'il est nécessaire d'envoyer les coordonnées.

Figure 12 - Modélisation du filtre par zone d'intérêt dans WorldState [Source Personnelle]

3. Extraction des interfaces à partir des modèles

Deux interfaces peuvent être extraites des différents modèles créés précédemment : une interface pour l'environnement de jeu (WorldState) et une interface pour les objets dont l'état est susceptible d'évoluer (MovingObject).

L'interface de MovingObject reprend les deux méthodes du modèle mis en avant dans la Figure 9 page 37, à savoir « EstimateState » et « Interpolate » de la façon suivante :

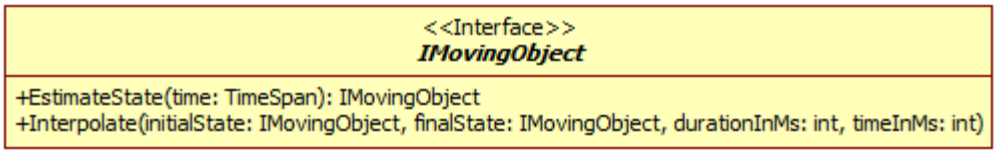

Figure 13 – Interface des optimisations d'un objet mobiles [Source Personnelle]

L'interface de WorldState, quant à elle, reprend les deux méthodes du modèle mis en avant dans la Figure 12 page 40, à savoir « replay » et « canInteract » :

Figure 14 – Interface des optimisations d'un état de l'environnement [Source Personnelle]

Ces interfaces doivent être implémentées par les classes WorldState et MovingObject. Les objets mobiles implémenteront implicitement l'interface en héritant de MovingObject, mais on peut également choisir d'implémenter directement IMovingObject dans ces classes sans hériter de MovingObject. Cette solution peut être adoptée si les implémentations par défaut de MovingObject ne conviennent pas ou si l'objet doit hériter d'une autre classe.

Maintenant qu'un mode de fonctionnement générique a été dégagé, nous allons pouvoir entrer dans le détail et proposer quelques implémentations génériques, et d'autres plus spécifiques au projet que nous souhaitons réaliser.

Chapitre 3 – Application des modèles avec C# et XNA

Les modèles et interfaces dégagées vous maintenant simplifier la mise en place des optimisations d'échanges de données. Ainsi, nous allons tenter de construire une base de jeu en C# capable d'exploiter ces interfaces. Cette base sera ensuite utilisée dans une application concrète de jeu multi-joueurs en réseau : un jeu de tir spatial en 2D.

1. Outils utilisés

1.1. Langage C# et Framework XNA

XNA est un Framework proposé par Microsoft pour réaliser des jeux pour les plateformes PC, XBOX, Zune et maintenant Windows Mobile. Nous concentrerons nos efforts pour optimiser une version PC uniquement. Les plateformes mobiles (Zune et Windows Mobile) ne proposent pas un accès au réseau ni des performances matérielles suffisantes pour le développement de jeux en réseau. Les éléments développés dans ce document peuvent en revanche bénéficier à une application XBOX qui ne se différencie que par les entrées utilisateurs (prise en charge de la manette obligatoire).

La version utilisée est XNA 3.1. Cette version contient des bibliothèques prévues pour les jeux en réseau mais nous ne les utiliserons pas. En effet, ces bibliothèques obligent l'utilisation de la plateforme « Live » développée par Microsoft, ce qui peut être intéressant dans certains cas mais rend le programme très dépendant. Nous souhaitons développer une application qui puisse fonctionner de manière indépendante, aussi préférerons nous utiliser une autre bibliothèque réseau. Une bibliothèque en particulier se distingue par sa gratuité et ses fonctionnalités : « Lidgren networking library generation 3 ».

1.2. Bibliothèque réseau

La bibliothèque réseau Lidgren est un projet libre hébergé sur Google code et qui répond aux fonctionnalités recherchées pour développer un jeu en réseau. Tout d'abord la bibliothèque est entièrement basée sur le protocole* UDP et implémente une couche de contrôle d'intégrité. Il est ainsi possible de s'assurer de l'ordonnancement des paquets et/ou la fiabilité de la transmission en spécifiant le contrôle souhaité lors de l'envoi. Cette bibliothèque gère

également la connexion entre plusieurs postes (dans notre cas la connexion de chaque client au serveur) ce qui facilite l'envoi des données vers les différents clients pour le serveur.

1.3. Environnement de développement

Pour développer cette application, nous utiliserons le logiciel « Visual C# 2010 Express » conjointement à « XNA Game Studio 3.1 » qui rend possible l'utilisation du Framework XNA. Ces deux logiciels sont disponibles gratuitement sur le site de Microsoft.[4]

L'environnement Visual Studio, développé par Microsoft, est optimisé pour l'utilisation du C# et du Framework XNA. Des alternatives libres existent tel que le projet Mono, mais nous ne nous y intéresserons pas dans cette partie.

[4] Liens disponibles à l'adresse : http://creators.xna.com/fr-FR/downloads (page visitée le 01/09/2010).

2. Détail de la structure applicative

2.1. Architecture générale

Lors de la modélisation des optimisations, nous avons mis en avant trois classes : WorldState qui représente un état de l'environnement de jeu, MovingObject qui représente un objet déplaçable de l'environnement et HistoryList qui est une collection d'éléments datés. L'objectif à présent est d'intégrer ces outils à une architecture plus générale qui va être capable de les utiliser. Notre structure applicative devra également implémenter des éléments récurrents dans les jeux vidéo, qui ne nécessitent pas forcément d'optimisation pour les échanges réseaux, mais font partie des fonctionnalités attendues de tout jeu. Nous ne détaillerons pas l'implémentation de ces éléments mais nous les inclurons à l'architecture générale.

L'application sera organisée en quatre paquets : le serveur, le client, le moteur réseau, et une librairie partagée. Leur relations de dépendances seront telles que définies dans le schéma ci-dessous :

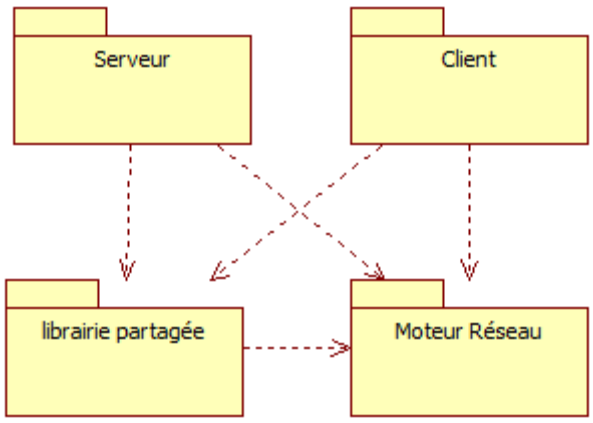

Figure 15 - Dépendances des paquets du Framework [Source Personnelle]

Notons que le serveur et le client partagent les mêmes dépendances mais ne sont pas dépendant l'un de l'autre. En effet, plusieurs programmes clients pourraient tout à fait être compatibles avec un même serveur, tout comme un même client pourrait communiquer avec des implémentations différentes du serveur. Pour autant, ils ne sont pas complètement indépendants : ils communiquent ensemble à travers le réseau et utilisent des instructions que

l'autre programme comprend. Ces instructions peuvent être consignées dans la librairie partagée par exemple.

2.1.1. Architecture du serveur

Détaillons à présent l'architecture pour le programme Serveur :

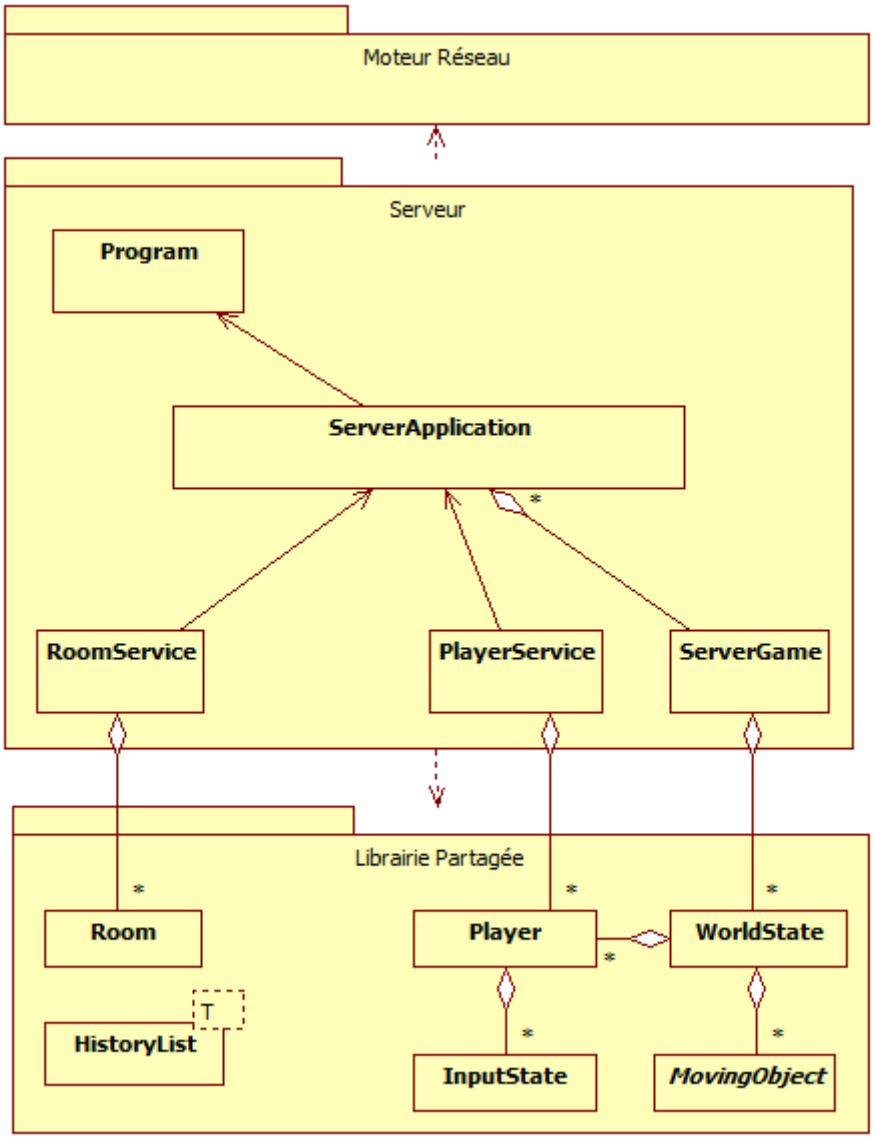

Figure 16 - Architecture du programme serveur [Source Personnelle]

Le serveur est dépendant du moteur réseau et de la librairie partagée. Le schéma détaille pour la librairie partagée quelques classes dont il dépend directement.

Le point d'entrée du serveur est "Program". Son rôle est d'initialiser le serveur et d'exécuter ServerApplication qui est le contrôleur principal. ServerApplication s'occupe ensuite d'initialiser tout le contexte nécessaire au bon fonctionnement de serveur : le gestionnaire de salles d'attentes (RoomService), le gestionnaire de communications textuelles (ChatService), le gestionnaire de connexions des joueurs (PlayerService) et enfin les parties en cours d'exécution (ServerGame). Chaque partie (ServerGame) est composée d'une collection d'états de l'environnement (WorldState) qui référencent chacun les joueurs impliqués dans la partie et les objets à déplacer dans l'environnement. Pour chaque joueur, on peut connaitre l'historique des dernières commandes reçues (InputState). Ainsi, toutes les informations nécessaires à l'utilisation des différents outils d'optimisation sont disponibles. Dans cette architecture, c'est le contrôleur ServerApplication qui écoute les messages réseaux provenant des clients et qui redirige l'information en fonction du message reçu.

2.1.2. Architecture du client

L'architecture du client est relativement similaire :

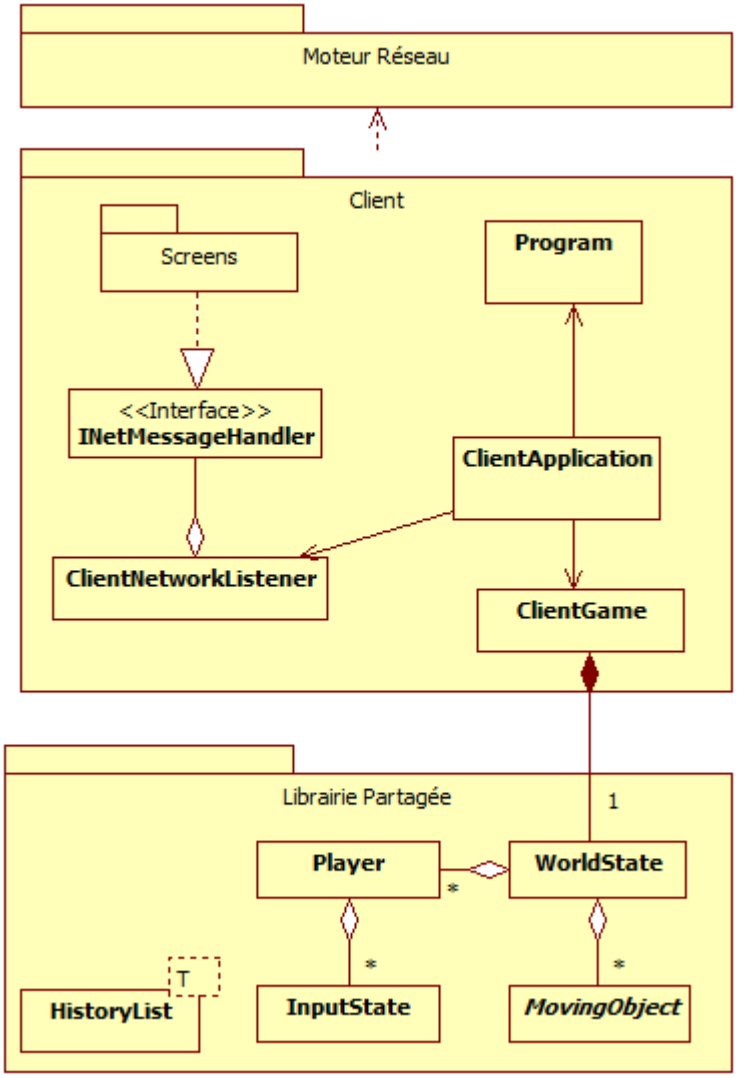

Figure 17 - Architecture du programme client [Source Personnelle]

De la même façon que le serveur, le point d'entrée du programme "Program" permet d'initialiser ClientApplication qui contrôle la logique de déroulement. En revanche, le client gère différemment la lecture des messages arrivant par le réseau. On connait sur les serveurs les différents services capables de répondre aux messages réseaux mais le serveur n'a pas un contexte global qui évolue. C'est à dire que chaque nouveau message reçu est repositionné dans son contexte à chaque fois (connexion d'un joueur, commande d'un joueur dans une

partie en particulier par exemple). Au niveau du client, il n'y a qu'un seul écran de jeu[5] visible à chaque fois par le joueur, et pour chaque écran de jeu le nombre de messages attendus est limité. Ainsi, un contrôleur principal de l'application ne redirige pas les informations en fonction des données du message, comme sur le serveur, mais envoie le message au contexte en cours (l'écran de jeu affiché) qui saura comment le traiter. Pour ce faire, un patron de conception "Observateur" peut être mis en place. Il est ici au niveau de la classe ClientNetworkListener qui reçoit les messages du réseau et les distribue à tous ses abonnés qui implémentent l'interface INetMessageHandler. Les écrans de jeu implémentent donc l'interface INetMessageHandler et s'abonnent à l'écouteur des flux réseaux pour réagir aux messages reçus.

Contrairement au serveur, la classe chargée de gérer une partie (ClientGame) n'est reliée qu'à une seule instance de l'environnement de jeu (WorldState). En effet, le client n'a besoin d'afficher que le dernier état reçu qu'il met à jour directement selon ses besoins et selon les données reçues du serveur.

[5] Un écran de jeu est un affichage unique prévu pour un état du jeu. Le menu principal est un écran de jeu, le menu Option est un écran de jeu, et la scène affichée lors d'une partie est un autre écran de jeu par exemple.

3. Mise en application dans un jeu de tir spatial

3.1. Règles du jeu

Dans tout projet de jeu, la première étape consiste à formaliser les règles du jeu et les façons de jouer dans un document appelé Game Design Document. Dans notre cas, quelques lignes suffiront à décrire les fonctionnalités du jeu.

3.1.1. Principe du jeu

Il s'agit d'un jeu de tir spatial multi-joueurs. Plusieurs joueurs pourront contrôler un vaisseau spatial dans un espace de jeu limité. La caméra est fixe, vue de dessus, et affiche la totalité de l'espace de jeu. Cinq commandes sont disponibles pour contrôler les vaisseaux : tourner à gauche, tourner à droite, accélérer, freiner, et tirer. A chaque fois qu'un joueur tire, un missile est envoyé à une vitesse fixe dans la même direction que le vaisseau émetteur du tir.

3.1.2. Conditions de victoire

Si le missile touche un autre vaisseau, le joueur contrôlant l'émetteur du missile obtient deux points, et celui contrôlant le vaisseau touché perd un point. Le premier joueur à atteindre cinquante points remporte la partie.

3.1.3. Formation des parties

Chaque joueur aura la possibilité de créer ou de rejoindre une salle d'attente. Le joueur ayant créé la salle d'attente peut lancer le début de celle-ci à tout moment, dès que 2 joueurs au moins font partie de la salle d'attente.

3.2. Fonctionnement de l'application

3.2.1. Structure des données

De ces quelques informations, on peut déduire un modèle de données qui nous permettra de mémoriser les informations dont nous avons besoin pour le déroulement du jeu :

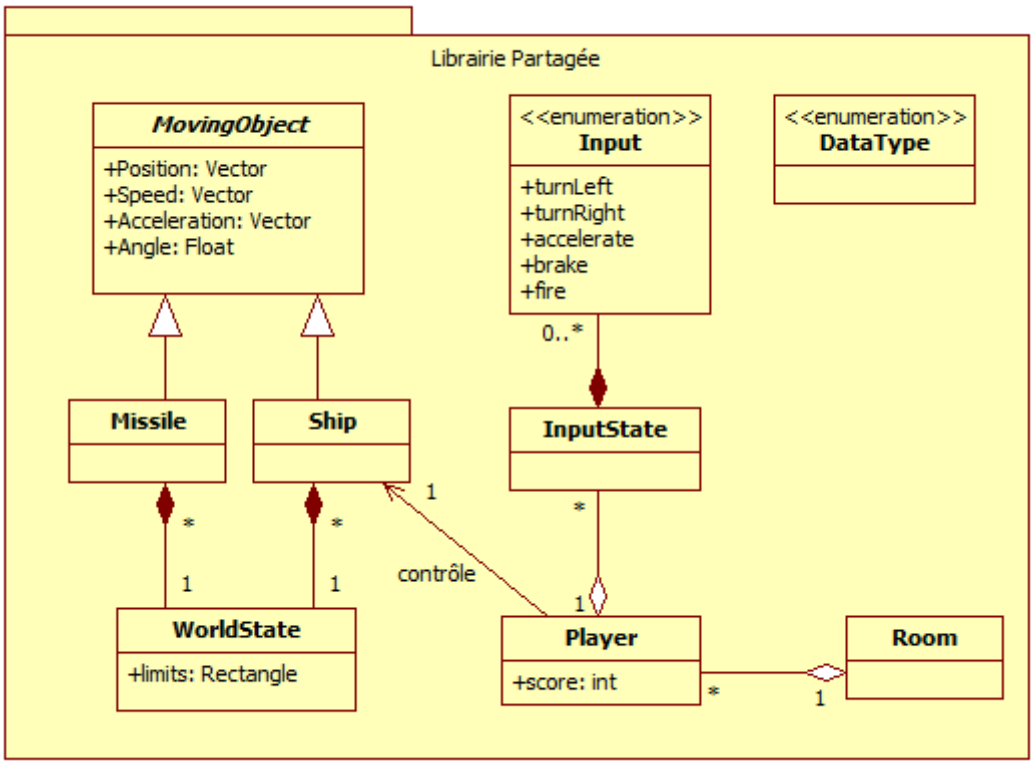

Figure 18 – Bibliothèque de classes pour un jeu de tir spatial [Source Personnelle]

D'après la description, l'environnement de jeu (WorldState) contient des vaisseaux (Ship), des missiles (Missile) et des limites. Cet environnement est altéré par les joueurs (Player) au travers de leurs commandes (InputState), il faut donc mémoriser les commandes pour les joueurs. Dans la mesure où le joueur peut appuyer simultanément sur plusieurs touches, une commande (InputState) peut être composée de plusieurs saisies (Input). Input est une énumération de toutes les saisies prises en charge par le programme. Enfin, il est prévu que des joueurs puissent rejoindre des salles d'attente (Room). Cette notion sera commune au client et au serveur et il est intéressant de la modéliser ici même si chacun l'utilise différemment.

3.2.2. Fonctionnement du Client

Le client sera découpé en quatre écrans : un écran principal, un écran de choix d'une salle d'attente, et un écran de salle d'attente et un écran de jeu. Chacun des écrans écoute les messages réseaux qui le concernent.

L'écran principal est composé de trois choix : Créer une salle, rejoindre une salle et quitter le jeu. Cet écran attend un unique message : la confirmation de la création d'une salle d'attente. Ce message survient lorsque le joueur sélectionne « Créer une salle d'attente » et que le serveur confirme la création. « Quitter le jeu » ne nécessite pas de confirmation du serveur et « rejoindre une partie » mène le joueur vers l'écran de choix d'une salle d'attente.

Une fois l'écran de choix d'une salle d'attente chargé, il demande au serveur la liste des salles disponibles et attend la liste du serveur. Le joueur peut alors sélectionner une salle, ce qui déclenche l'envoi au serveur d'une demande pour rejoindre la salle. L'écran attend alors une réponse concernant la disponibilité de la salle voulue et affiche une erreur si la salle n'est plus disponible ou passe à l'écran de salle d'attente si cette dernière a été rejointe avec succès.

L'écran de salle d'attente permet de voir les autres joueurs connectés à la salle et de personnaliser son pseudonyme. Cet écran peut recevoir quatre messages : confirmation du changement de pseudonyme, mise à jour de la liste des joueurs connectés, mise à jour de la disponibilité de la salle (si elle a été supprimée) et début de la partie. Les deux premiers messages mettent à jour les informations affichées, une indisponibilité de la salle renvoie vers le menu principal, le début de partie déclenche l'affichage de l'écran de jeu.

L'écran de jeu est l'écran laissant place aux mécanismes du jeu. Cet écran reste affiché jusqu'à la fin de la partie, qui provoque l'affichage d'un tableau des scores puis le retour au menu principal. Bien que cet écran soit le plus complexe (beaucoup d'informations sont à afficher) il n'attend que deux messages : un premier indiquant que tous les joueurs sont prêts et que la partie démarre, le second étant un paquet de données reçu du serveur. Le message de données en provenance du serveur est celui envoyé périodiquement par le serveur pour faire part au client du nouvel état de l'environnement. Le client utilise ensuite ces données pour mettre à jour son état de l'environnement. L'autre partie du travail de cet écran est d'écouter les commandes du joueur pour les envoyer, à chaque changement de commande, au serveur.

3.2.3. Fonctionnement du Serveur

Nous avions définis dans l'architecture du serveur (Figure 16 page 45) trois services : RoomService pour la gestion des salles d'attente, PlayerService pour la gestion connexions des joueurs, et ServerGame pour gérer les parties (une instance est créée par partie).

Chaque message est reçu par le contrôleur principal et redirigé vers le service correspondant. Un exemple de code correspondant à ce fonctionnement est disponible en annexe page I et intitulé : Traitement d'un message du client par le serveur du jeu.

Après une initialisation de réseau (écoute des connexions entrantes), le serveur reste sur un mode de fonctionnement unique. Il se contente d'écouter les connexions et les messages pour y répondre. Par exemple, le lancement d'une partie se déroule de cette manière :

- Joueur 1 se connecte (ajout de joueur 1 dans la liste des connexions)
- Joueur 1 créer une salle d'attente (création de la salle 1, ajout de joueur 1 comme hôte de la salle)
- Joueur 2 se connecte (ajout de joueur 2 dans la liste des connexions)
- Joueur 2 demande la liste des salles (envoi de la liste à joueur 2)
- Joueur 2 demande à rejoindre la salle 1 (la salle peut accepter un joueur supplémentaire, envoi de l'autorisation à joueur 2, envoi de la liste des joueurs présents dans la salle à joueur 2, envoi la nouvelle liste de joueurs à joueur 1)
- Joueur 1 voit joueur 2 se connecter et demande le lancement de la partie (la partie peut être démarrée, création d'un nouveau processus pour la partie, envoi de l'instruction de début de partie aux joueurs 1 et 2)
- En continu, le serveur écoute les commandes des différents joueurs et les redirigent vers le thread de la partie qui saura les interpréter.
- A intervalle réguliers, le serveur envoi aux clients les informations à jour de l'environnement de jeu.

3.3. Implémentation des optimisations

3.3.1. Compression

Indifféremment du fonctionnement du client et du serveur, les optimisations de compression s'intègrent partout dans l'application. Elles consistent à faire circuler les données les moins volumineuses possibles sur le réseau, nous allons pour cela choisir des types de données adaptés. Dans un jeu, la rapidité de traitement est généralement préférable à la précision. Certaines données doivent garder une précision suffisante (par exemple un angle utilisé pour tirer sur une grande distance doit avoir une précision au dixième voire au centième de degré) mais certains nombres peuvent être arrondis à une valeur moins précise mais moins coûteuse en mémoire.

Pour les nombres par exemple, C# nous propose les types suivants :

Type	Plage de valeurs	Nombre de bits
sbyte	-128 to 127	8
byte	0 to 255	8
short	-32,768 to 32,767	16
ushort	0 to 65,535	16
int	-2,147,483,648 to 2,147,483,647	32
uint	0 to 4,294,967,295	32
long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	64
ulong	0 to 18,446,744,073,709,551,615	64

Figure 19 – Types numériques intégraux en C# [documentation du langage]

A chaque besoin il faudra donc choisir le type qui nous est suffisant pour limiter le volume de la donnée. Ainsi, toutes nos coordonnées à l'écran ne dépasseront jamais la résolution de l'écran qui avoisine au mieux les 2000 pixels en largeur. Ainsi, nous utiliserons pour des coordonnées le type « short » plutôt que « int » ou « long ».

Shawn Hargeaves, développeur de chez Microsoft travaillant sur le Framework XNA, propose également quelques astuces pour réduire la taille des données envoyées. Les exemples suivants sont tirés du document de support de sa conférence au Gamefest en 2007 « Making Networked Games with the XNA Framework ».

Angles en radian

Pour un angle en radian, plutôt que de transmettre la donnée de type float directement, il est préférable de convertir cet angle en une valeur comprise en 0 et 255. La précision est réduite à 256 valeurs de l'angle mais la donnée utilise quatre fois moins de place. Ci-dessous le code de transformation :

```
// en radian, sur 32 bits
float rotation;

// transformation en byte (valeur entre 0 et 255)
rotation *= 256;
rotation /= (float)(Math.PI / 2);

// en byte, sur 8 bits
byte valeurTransmise = (byte)rotation ;
```

On appliquera les formules inverses (division par 256 et multiplication par PI sur deux) pour retrouver un angle en radian.

Booléens

Pour transmettre des booléens, il faut éviter de les transmettre un par un mais les regrouper dans un même byte. Nous éviterons donc :

```
// booléens à transmettre
bool isAlive, isRespawning, isFiring, hasPowerup;
// chaque booléen est écrit dans le paquet
// à ne pas faire car chaque booléen prend un byte à chaque écriture !
packetWriter.Write(isAlive);
packetWriter.Write(isRespawning);
packetWriter.Write(isFiring);
packetWriter.Write(hasPowerup);
```

Nous ferons plutôt :

```
// un byte est codé sur 8 bits
 byte bitfield = 0;
 // pour chaque bit de bitfield, on affecte un booléean
 if (isAlive) bitfield |= 1;
 if (isRespawning) bitfield |= 2;
 if (isFiring) bitfield |= 4;
 if (hasPowerup) bitfield |= 8;
 // un seul byte est envoyé et contient jusqu'à 8 booléens
 packetWriter.Write(bitfield);
```

En effet, un byte est utilisé pour chaque booléen écrit dans un buffer. Ainsi, en regroupant huit booléens dans un même byte on économise 7 bytes (soit 48 bits).

Nombres à virgule flottante

Les nombre à virgule flottant sont des nombres exprimés sous forme du type float (sur 32 bits) ou double (sur 64 bits). Le Framework XNA propose une classe capable de réduire la taille d'un nombre à virgule flottant à 16 bits. Cette classe s'appelle HalfSingle et s'utilise de la façon suivante :

```
float angle;
float speed;
HalfSingle packedAngle = new HalfSingle(angle);
HalfSingle packedSpeed = new HalfSingle(speed);
packetWriter.Write(packedAngle.PackedValue);
packetWriter.Write(packedSpeed.PackedValue);
```

De cette manière, on économise 16 bits par nombre à virgule flottante envoyé.

Nombres aléatoires

Il est parfois nécessaire de générer des nombres aléatoires. Il est en réalité possible de générer les mêmes nombres aléatoires dans plusieurs applications différentes en renseignant la même valeur de série de nombres. Par exemple, la classe Random en C# permet d'obtenir des nombres pseudo-aléatoires calculés par des algorithmes mathématiques précis mais dépendant d'une valeur initiale. C'est cette valeur que nous allons chercher à transmettre à travers le réseau plutôt que les valeurs des nombres aléatoires. Nous allons donc transmettre la valeur de la série, appelée « seed », puis chacun recevant le message sera capable de régénérer les nombres souhaités :

```
// début de série alétoire (par rapport à l'instant actuel)
int seed = (int)Stopwatch.GetTimestamp();
// on envoi le nombre de début de série
packetWriter.Write(seed);

// Puis sur chaque personne récupérant le seed envoyé
Random random = new Random(seed);
// affecte une position initiale et un angle à chaqun des vaisseaux
foreach (Ship ship in worldState.ships)
{
    ship.x      = (float)random.NextDouble()*500 ;
    ship.y      = (float)random.NextDouble()*500 ;
    ship.angle  = (float)random.NextDouble()*(Math.PI*2);

}
```

L'économie en volume par rapport à un envoi de toutes les coordonnées aléatoires calculées par le serveur est conséquent et proportionnel au nombre de vaisseaux dans cet exemple.

Les optimisations de compression seront donc appliquées aux types de données de chaque objet à échanger en réseau dans notre application.

3.3.2. Estimation des positions

Contrairement aux gains de volume liés à la compression que nous venons de voir, l'estimation des positions ne permet pas directement de réduire la quantité ou d'accélérer la transmission des informations.

Cependant, son utilisation à travers la compensation des temps de latence nous permettra de réduire le nombre d'échanges entre le client et le serveur sans perdre en fluidité au niveau de l'affichage.

Concentrons nous d'abord sur l'outil en tant que tel. Nous avons modélisé son fonctionnement à travers la méthode « EstimateState » de la classe « MovingObject ». La méthode prend un paramètre qui est la période de temps qu'on souhaite simuler. Cette méthode pourra être utilisée conjointement à la méthode « Interpolate » qui permet de calculer des positions intermédiaires entre deux positions sur une période donnée.

Dans notre jeu, deux classes sont concernées par ces optimisations : Ship (les vaisseaux) et Missile (les missiles). Ces deux classes héritent de MovingObject, aussi commencerons nous par implémenter une méthode générique d'estimation de la position et d'interpolation pour éventuellement les surcharger dans Ship et Missile. Un exemple d'implémentation de MovingObject est disponible en annexe page II. Par défaut, on applique une estimation des positions basée sur les dérivées tel que suggéré dans le Chapitre 1, et une interpolation linéaire des différentes valeurs de l'objet (position, angle, vitesse, accélération).

Pour l'implémentation d'un Missile (annexe page III) on surcharge la méthode « estimateState » pour la simplifier. En effet le comportement rectiligne constant du missile est très simple à prédire. Pour la méthode « interpolate », on simplifie également le code en le surchargeant, seule l'interpolation de la position est nécessaire, les autres valeurs restent constantes.

Enfin, l'implémentation de la classe Ship (annexe page IV) est particulière puisque le vaisseau est contrôlé par le joueur. Ainsi, il est possible de réutiliser les dernières commandes

émises par le joueur pour estimer la position actuelle. Pour cela, on s'aide des calculs utilisés par le serveur pour faire bouger le vaisseau, ce qui nous garantit une précision de prédiction parfaite si le joueur n'effectue pas d'autres actions entre temps.

3.3.3. Compensation des temps de latence

En ce qui concerne la compensation des temps de latence, la première chose à faire est d'anticiper les mouvements du vaisseau contrôlé en fonction des commandes du joueur local. Les déplacements des vaisseaux sont normalement contrôlés par les messages de mise à jour en provenance du serveur. Chaque message de mise à jour contient un état de l'environnement tel que le client doit l'afficher. A chaque message reçu, la totalité des positions des vaisseaux et missiles sont mises à jour en faisant évoluer la position actuelle du vaisseau vers la position reçue (à l'aide de la méthode d'interpolation). Par conséquent, pour que le vaisseau soit directement contrôlé par le joueur local, on commence par supprimer cette mise à jour automatique pour le vaisseau concerné et on utilise à la place les commandes du joueur pour le déplacer. Voici le code qu'on pourrait trouver au niveau du client à la réception d'un nouvel état de l'environnement :

```
// Mise à jour du worldState lors de réception d'un message serveur
void updateWorldState(WorldState newWorldState)
{
    // Le précédent état reçu est recopié.
    // nextWorldState peut ensuite être modifier
    // pour accueillir les nouvelles données.
    previousWorldState.copy(nextWorldState);

    // Pour chaque vaisseau qui n'est pas contrôlé par le joueur...
    foreach(Ship ship in newWorldState.ships)
    {
        if (ship.player != currentPlayer)
        {
            // ... on met à jour ses informations
            nextWorldState.ships[ship.id] = ship;
        }

    }
    // on met à jour les informations de tous les missiles
    nextWorldState.missiles = newWorldState.missiles;
}
```

La variable « nextWorldState » peut ensuite être utilisée comme référence pour faire évoluer l'état de l'environnement en cours d'affichage. Elle servira donc à renseigner la méthode « Interpolate » que l'on pourra appeler à chaque mise à jour de l'affichage. Avec XNA, le framework appelle la méthode « Update » automatiquement pour mettre à jour les données du

jeu. C'est cette méthode que nous allons utiliser. Dans cette méthode, nous écoutons les saisies du joueur pour les transformer en commandes, nous déplaçons nos objets à l'aide de leur méthode d'interpolation, nous envoyons les commandes du joueur au serveur si elles ont changé. Voici un exemple en C# de ce que la mise à jour des positions côté client pourrait être :

```
public override void Update(GameTime gameTime)
{
    // Appel de la méthode de la classe parente
    base.Update(gameTime);

    // si la partie est démarrée
    if (gameStarted)
    {
        // Récupère les entrées saisies par le joueur
        // et en déduis les commandes
        InputState playerInput = readPlayerInputs();

        // On compare avec les commandes précédentes.
        // En case de différence, on annonce au serveur que les commandes
        // ont changé. (sinon il réutilise implicitement les mêmes)
        if (!playerInput.Equals(lastInput))
        {
            lastInput = playerInput;
            sendNewInputToServer(playerInput);
        }

        // durée entre les deux états à interpoler
        int duration =
            (nextWorldState.time - previousWorldState.time);
        // durée écoulée depuis le dernier état
        int timeElapsedSincePreviousState =
            currentTime - previousWorldState.time;
        // Fait évoluer les éléments entre l'état précédent reçu par
        // le serveur et le dernier état reçu
        foreach (Ship ship in currentWorldState.ships)
        {
            ship.interpolate(
                previousWorldState.ships[ship.id],
                nextWorldState.ships[ship.id],
                duration,
                timeElapsedSincePreviousState
            );
        }
        foreach (Missile missile in currentWorldState.missiles)
        {
            missile.interpolate(
                previousWorldState.missiles[missile.id],
                nextWorldState.missiles[missile.id],
                duration,
                timeElapsedSincePreviousState
            );
        }
    }
}
```

On remarque cependant que le joueur déplace à présent son vaisseau de manière totalement indépendante du serveur ! Il convient donc d'effectuer un contrôle de la position du vaisseau en fonction de la dernière position envoyée par le serveur. Une façon simple de faire cette vérification est de mesurer l'écart entre la dernière position envoyée par le serveur et la position actuelle sur le client. Si la position diffère trop, alors on ramène le vaisseau au dernier emplacement envoyé par le serveur (par interpolation pour plus de fluidité dans l'affichage). On peut également envisager des techniques de contrôle plus évoluées, le client pourra par exemple mémoriser la liste des positions antérieures de l'objet contrôlé par le joueur, et vérifie la correspondance entre la position reçue du serveur et la position mémorisée au moment où le serveur a calculé la position (en prenant en compte le temps de latence). Plus difficile à mettre en place et plus coûteuse en mémoire, cette façon de faire est cependant plus précise.

Au niveau du serveur, l'implémentation de la boucle de jeu est un peu plus compliquée. Pour que le joueur puisse avoir un rendu cohérent et une visée précise, le serveur doit retrouver l'environnement de jeu tel qu'il était au moment où la commande du joueur a été jouée. Ensuite, il joue la commande à partir de l'état de jeu antérieur, puis recalcule les états suivants en conséquence. Nous avions dans nos modèles une méthode « replay » que doit implémenter l'environnement du jeu (WorldState) pour réaliser cette opération. Au moment du calcul d'un nouvel état, le serveur appelle une méhode qui pourrait être la suivante :

```
private void updateWorldState()
{
    // pour chaque vaisseau à mettre à jour
    foreach (Player player in players)
    {
        // récupère l'historique des commandes reçues
        // depuis la dernière mise à jour
        HistoryList<InputState> playerLastInputs =
            player.inputs.getAllSince(lastUpdateTime) ;
        // réactualise les états antérieurs à partir des
        // nouvelles commandes reçues qui concernent un état passé
        WorldState.replay(player, playerLastInputs, worldStateHistory);
    }
    // la dernière mise à jour vient d'être faite
    lastUpdateTime = currentTime;
}
```

Un exempe d'implémentation de la méthode « replay » de l'objet WorldState est proposé en annexe page V.

3.3.4. Filtres par zone d'intérêt

Enfin, dernière des optimisations à implémenter, le filtre par zone d'intérêt reste une optimisation importante à la fois pour économiser de la bande passante et pour limiter la triche.

Dans la mesure où dans notre jeu tous les éléments sont visibles en permanence, il n'est pas nécessaire d'implémenter une telle fonction. Pour montrer de quelle manière elle peut être faite, essayons de l'implémenter pour que les vaisseaux ne voient plus leurs adversaires situés dans leur dos. Ainsi, si la zone est visible à l'écran du joueur, un « brouillard » viendra masquer la présence éventuelle des joueurs rendant impossible la prédiction de leur trajectoire. Nous limiterons également le champ de vision des vaisseaux à 200 pixels. Voici comment on pourrait procéder du côté serveur pour ne pas envoyer les informations qu'il n'est pas sensé afficher au client :

```
// pour chaque joueur
foreach (Player player in players)
{
    // récupère l'état actuel
    WorldState worldStateToSend =
        new WorldState(currentWorldState);

    // liste les vaisseaux que peut voir le joueur
    List<Ship> shipsToSend = new List<Ship>();
    foreach (Ship otherShip in worldStateToSend.ships)
    {
        if (worldStateToSend.canInteract(player.ship, otherShip))
        {
            shipsToSend.Add(otherShip);
        }
    }

    // envoi la liste réduite à la place de la liste complète
    worldStateToSend.ships = shipsToSend;
    sendWorldStateToPlayer(worldStateToSend, player);
}
```

La méthode « canInteract » utilisée dans l'exemple de code précédent, pourrait quant à elle être implémentée de la façon suivante :

```
public bool canInteract(Ship ship, Ship otherShip)
{
    // ship ne voit pas otherShip s'il est distant de plus de 200 pixels
    if (Vector2.Distance(ship.position, otherShip.position) > 200)
        return false;

    // si la somme des vecteurs normalisés en inférieur à 1,
    // alors otherShip a plus de 90° d'écart avec le centre de l'angle
    // de vue et est dans le dos de notre vaisseau de référence.
    Vector2 shipsVector = (otherShip.position-ship.position) ;
    Vector2 viewVector = new Vector2(
        Math.Cos(ship.angle),
        Math.Sin(ship.angle)
    );
    viewVector.Normalize();
    shipsVector.Normalize();
    if ((viewVector + shipsVector).Length < 1)
        return false;

    // si aucune des conditions n'est remplie, otherShip est visible
    return true;

}
```

Nous venons de voir une manière d'implémenter les différentes optimisations dans jeu de tir spatial en 2D. Le principe du jeu est simple, les règles peu nombreuses, mais déjà mettre en place les optimisations pour obtenir un jeu fluide prend du temps.

Conclusion

Au vu de la complexité des implémentations que nous venons d'expérimenter dans un jeu aux règles simples, on peut aisément deviner la difficulté de réalisation de ces développements dans des jeux de grande ampleur où des dizaines d'objets différents interagissent entre eux. Avec l'aide des modèles dégagés cependant, la difficulté de mise en place a été amoindrie. En effet, les problématiques d'optimisation des échanges de données ont été concentrées dans quelques méthodes prévues dans les modèles, et exploitables à partir d'interfaces. Ainsi, tant que ces interfaces sont implémentées correctement, il est possible de faire fonctionner un jeu en réseau de manière fluide sur Internet.

Les jeux vidéo ont en effet besoin de fluidité. Comme nous l'avons vu ils ont aussi besoin de cohérence et de réactivité pour rendre l'expérience de jeu agréable. Un jeu prévu pour un seul joueur peut y parvenir en réduisant les besoins en calcul du jeu, mais un jeu prévu pour plusieurs joueurs en réseau nécessite des considérations supplémentaires. En effet, le réseau internet qui est largement utilisé pour les jeux en réseau impose des contraintes de débit et de temps de latence. Ainsi, pour les jeux d'action, il convient de choisir une architecture réseau adaptée, telle que l'architecture client/serveur qui est optimisée pour 2 à 32 joueurs environ. Cette architecture consiste à créer un programme serveur, qui fasse autorité, et qui arbitre et coordonne les parties des programmes clients. Au-delà de 32 joueurs, il faudra s'orienter vers une architecture en réseau de serveurs, qui consiste à appliquer l'architecture client/serveur à des serveurs entre eux. Nous veillerons également à utiliser un mode de transmission unicast vers chacun des clients plutôt que du multicast qui pose parfois des problèmes de compatibilité avec du matériel plus ancien.

Au-delà de ces choix concernant la conception générale de l'application, des optimisations plus spécifiques devront être mises en place pour contourner les limitations du réseau internet. Pour ne pas dépasser la bande passante disponible, il conviendra d'implémenter des optimisations de compression des données. Certaines méthodes de compression ou d'agrégation de messages pourront être implémentées par la librairie réseau utilisée pour faire communiquer le programme client et le programme serveur. Ainsi, elles seront utilisables sans développement supplémentaire. La compression s'applique aussi au niveau des types de données transmises par le réseau. Les types standards utilisés en C# occupent par exemple un

volume de données plus important que nécessaire pour stocker l'information. Un entier de type « int » consomme 32 bits en mémoire, alors que si la valeur stockée ne dépasse pas 65000, un type « short » sur 16 bits convient largement. Pour chaque donnée, il convient aussi de réduire au maximum la précision, dans des limites acceptables pour la cohérence du rendu, pour diminuer le volume de données nécessaire pour stocker et transmettre l'information. Par exemple, un angle qu'on limite à 256 valeurs est quatre à huit fois plus petit par rapport aux types habituellement utilisés. D'autres optimisations équivalentes existent pour transmettre des valeurs booléennes, pour réduire la taille des nombres à virgule flottante, et pour faire en sorte que chaque programme soit capable de générer des mêmes nombres aléatoires à partir d'une seule valeur plutôt que transmettre chacun de ces nombres.

La compression permet donc de réduire la taille d'un message, et donc le débit nécessaire et la durée de transmission. Une autre optimisation peut être mise en place au niveau du serveur pour cette même raison, mais aussi pour limiter la triche. Il s'agit des filtres par zone d'intérêt. Le principe est d'appliquer au niveau du serveur un filtre sur tous les objets envoyés à chaque client en fonction des données que doit afficher le client. Ainsi, on envoie au client uniquement les objets qu'il est sensé afficher. Cela réduit le volume de données envoyées et empêche le joueur de tricher en lisant directement sur sa carte réseau dans le message reçu les positions d'objets qu'il n'est pas sensé connaitre. La mise en place de cette optimisation se fait au niveau de la classe en charge de gérer l'état de l'environnement de jeu. En effet, cette classe détient les informations nécessaires pour déterminer si un objet est susceptible d'interagir avec un autre et donc si l'autre objet doit être envoyé ou non. Ainsi, il est possible d'implémenter une interface dans la classe de l'état de l'environnement, afin qu'elle contienne la méthode à utiliser pour vérifier si l'interaction est possible ou non.

Si les filtres par zone d'intérêt et la compression sont efficaces pour réduire la bande passante utilisée, ils ne permettent pas de contourner les problèmes de temps de latence. En effet, sur Internet, il faut entre 20 et 500ms pour qu'une information s'échange entre deux nœuds du réseau. Un joueur perçoit les latences entre le moment où il agit et le moment où l'image se met réellement à jour sur son écran dès 100ms. Ainsi, un système où le client attend la

réponse du serveur pour mettre à jour l'information de l'objet contrôlé par le joueur à l'écran est difficile à manier. Le joueur peut en effet être fortement gêné par le décalage entre son action et la réaction dans le jeu. Ainsi, il est possible de mettre en place des solutions de compensation des temps de latence, à l'aide d'outils tels que la prédiction des positions et l'interpolation de mouvement. La prédiction permet d'anticiper un état pour compenser certains retards dans l'arrivée de l'information. L'interpolation quant à elle rend le mouvement plus fluide entre deux positions, et permet l'envoi de mises à jour moins régulières. En effet, les positions intermédiaires entre deux mises à jour de positions pourront être calculées. Ce calcul est d'autant plus précis si chaque classe implémente sa propre méthode d'interpolation. Il en va de même pour les calculs de prédiction de mouvements. Aussi, il est avantageux de faire en sorte que les objets mobiles à optimiser implémentent eux-mêmes deux fonctions, une de prédiction et une d'interpolation. Pour ce faire, une interface spécifique à ces objets mobiles peut être mise en place.

La compensation des temps de latence ne se limite cependant pas à la prédiction et à l'interpolation. Encore faut-il utiliser ces outils convenablement. Il n'existe cependant pas de façon unique de la mettre en place. Nous avons vu les concepts généraux de la méthode et proposé une implémentation générique, mais il est préférable de considérer ces indications comme un patron de conception qui doit être appliqué spécifiquement à chaque projet.

Par conséquent, même si une partie de la complexité est abstraite à l'aide d'interfaces, il convient de connaitre ces optimisations. Il n'est donc pas possible de cacher toute la difficulté des optimisations des échanges de données dans un jeu multi-joueurs en temps réel, derrière un modèle générique. S'il est possible de modéliser certains outils pour faciliter leur mise en place, leur implémentation devra être réfléchie par rapport aux spécificités du jeu. Les solutions sont donc multiples, et devront être étudiées par le game designer[6] avec l'aide des développeurs pour trouver celle qui semble la plus appropriée à leur projet.

[6] Le game designer est la personne qui conçoit les règles et mécanismes de jeu. Terme spécifique au domaine de la création de jeux.

Nous trouvons donc ici la réponse à notre problématique : est-il possible de modéliser un traitement optimal des échanges de données dans un jeu multi-joueurs temps réel ?

Nous avons pu mettre en avant des modèles qui clarifient l'utilisation d'outils nécessaires à l'optimisation des échanges de données. Cependant, du fait des spécificités inhérentes à chaque projet de jeu, il ne semble pas possible de trouver un fonctionnement générique qui réponde à toutes les attentes de la meilleure façon. Ainsi, s'il est possible de modéliser des outils génériques qui permettent de traiter les échanges de données, ce traitement ne peut pas toujours être optimal car la généricité se fait au détriment des particularités. Aussi, plutôt que d'essayer d'encapsuler les optimisations derrière des modèles, il est préférable de faire connaitre aux développeurs les méthodes à suivre, à travers une charte de développement par exemple. Cette charte lui indiquera quel type de données utiliser pour chacune des optimisations possibles, et quelle méthode implémenter pour obtenir le résultat voulu. Impliquer le développeur permettra d'obtenir des réponses plus spécifiques, et donc mieux optimisées. De plus, le développeur est parfois amené à supposer la façon dont les optimisations fonctionnent pour implémenter d'autres fonctionnalités du programme. En développant spécifiquement les optimisations, il obtient la maitrise des mécanismes d'optimisations, ce qui évite des effets de bords dans d'autres parties de l'application.

Si les méthodes proposées peuvent être adaptées à certains besoins, il serait dommage de s'arrêter à une seule façon d'optimiser les jeux en réseau. Le gameplay* dans le jeu vidéo dépend grandement de la capacité des développeurs à innover et à trouver des meilleures solutions pour contourner les contraintes. Plus les possibilités seront grandes, et plus les jeux proposeront une expérience riche et plaisante.

Ainsi, Il est théoriquement possible de développer d'autres optimisations en parallèle. Celles présentées ici concernent essentiellement les jeux d'action, et seront peu efficaces pour des jeux de stratégie (ou RTS*), où le temps de latence importe moins, mais où le nombre d'objets à transférer est très important. Dans un RTS*, le mécanisme utilisé est la simulation synchronisée. Là où un jeu d'action se contente d'approximer les positions et de les rectifier lorsque l'écart est trop grand, le jeu de stratégie oblige chaque client à exécuter une simulation complète et strictement identique. L'intérêt est que les positions de toutes les unités présentes n'ont pas besoin d'être transmises par le réseau puisque chacun est capable

de les calculer à partir des commandes des joueurs. L'inconvénient est qu'il est hors de question de revenir en arrière, comme dans notre méthode de compensation des temps de latence, pour rejouer un état passé. En effet, il faudra annuler et rejouer en permanence un grand nombre de commandes ce qui serait très lourd à calculer. Cette fois la commande n'est donc réalisée qu'après avoir été confirmée par le serveur, et est exécutée au moment correspondant à la réception de la commande par le serveur. Le résultat apparait donc, de façon identique pour chaque client, après la durée du temps de latence.

Une amélioration possible serait de coupler les deux systèmes d'optimisation. Imaginons que nous voulions un jeu de stratégie avec des phases d'action en temps réel. Il serait tout à fait possible d'implémenter une simulation synchronisée qui garderait l'évolution de plusieurs unités, et une compensation des temps de latence* pour les quelques unités contrôlées. Les unités simulées passeraient dans le système temps-réel lorsqu'elles seraient impactées par des actions directes (collisions, explosions, etc.) puis reviendraient dans un mode simulé dès que les interactions des objets contrôlés en temps réels seraient finies.

Il existe probablement de nombreuses autres possibilités d'évolutions, et on ne peut qu'encourager les développeurs de jeux vidéo à aller dans ce sens, pour enrichir encore les possibilités offertes aux joueurs pour leur plaisir de jeu.

Schémas, figures, et tables

Figure 1 - Architecture d'une application multi-utilisateurs : (a) nœud seul, (b) architecture Peer-to-Peer, (c) architecture client-serveur, (d) architecture réseau de serveurs [6] 13

Figure 2 - Centralisation des données [6] 15

Figure 3 - Duplication des données [6] 15

Figure 4 - Répartition des données [6] 16

Figure 5 - convergences linéaires et cubiques [source personnelle] 28

Figure 6 - Diagramme des dépendances pour un jeu en réseau [Source Personnelle] 33

Figure 7 – Modélisation d'un état de l'environnement de jeu [Source Personnelle] 34

Figure 8 - Modélisation d'un objet implémentant l'estimation des positions [Source Personnelle] 36

Figure 9 – Modélisation d'un objet implémentant la convergence [Source Personnelle] 37

Figure 10 – Modélisation de la classe d'historisation des états [Source Personnelle] 38

Figure 11 – Modélisation de l'environnement de jeu [Source Personnelle] 39

Figure 12 - Modélisation du filtre par zone d'intérêt dans WorldState [Source Personnelle] . 40

Figure 13 – Interface des optimisations d'un objet mobiles [Source Personnelle] 41

Figure 14 – Interface des optimisations d'un état de l'environnement [Source Personnelle] .. 41

Figure 15 - Dépendances des paquets du Framework [Source Personnelle] 44

Figure 16 - Architecture du programme serveur [Source Personnelle] 45

Figure 17 - Architecture du programme client [Source Personnelle] 47

Figure 18 – Bibliothèque de classes pour un jeu de tir spatial [Source Personnelle] 50

Figure 19 – Types numériques intégraux en C# [documentation du langage] 53

Glossaire

Collusion : entente secrète entre 2 joueurs au préjudice d'un troisième. La collusion peut être assimilée à de la triche si elle n'est pas prévue dans les règles du jeu.

FPS (First Person Shooter) : jeu de tir à la première personne. C'est un genre du jeu vidéo où l'écran affiche ce que le personnage incarné par le joueur est supposé voir. Le personnage est souvent armé et doit combattre des créatures contrôlées par l'ordinateur ou d'autres joueurs.

Gameplay : agrément que procure d'utilisation d'un jeu vidéo. Le gameplay est influencé par les règles du jeu, par la manière dont le joueur est censé y jouer, par la fluidité de ces règles une fois appliquées à l'environnement du jeu, et également par les possibilités offertes par l'environnement.

MMORPG (Massively Multiplayer Online Role Playing Game) : jeu de rôle en ligne rassemblant un très grand nombre de joueurs en simultané. L'exemple de MMORPG le plus connu est World of Warcraft.

Mode : variante des règles d'un jeu. Elle s'exprime soit par un paramétrage particulier (mode facile ou difficile), soit par un changement des objectifs de jeu. L'intérêt d'un mode est de proposer un gameplay alternatif et d'enrichir le jeu.

Protocole : Norme de communication sur un réseau informatique.

RTS (Real-time Strategy) : Jeu de stratégie en temps réel. C'est un genre du jeu vidéo où le joueur incarne le commandant d'une armée. Il est chargé de remplir des missions contre l'ordinateur ou d'autres joueurs en donnant des ordres aux unités sous son contrôle.

Temps de latence : durée écoulée entre le moment où le message est envoyé, et celui où la réponse à ce message est reçu. En réseau informatique on parle également de temps de réponse, et de *lag* en anglais.

Table des matières

INTRODUCTION 2

CHAPITRE 1 – OPTIMISATION DES ECHANGES DE DONNEES 9

1. BESOINS D'UN JEU EN RESEAU 9
   1.1. Contraintes d'un jeu réseau 9
      1.1.1. Support de communication 9
      1.1.2. Cohérence 11
      1.1.3. Réactivité 11
      1.1.4. Fluidité 11
   1.2. Architecture réseau 13
      1.2.1. Architectures envisageables. 13
      1.2.2. Répartition les données 14
      1.2.3. Choix d'une architecture 16
   1.3. Mode de transmission des données 18
      1.3.1. Méthodes d'envoi des données 18
      1.3.2. Protocole de transmission 18
2. METHODES D'OPTIMISATION 21
   2.1. Compression 22
      2.1.1. Compression des messages 22
      2.1.2. Agrégation des messages 23
   2.2. Estimation des positions 24
      2.2.1. Prédiction 24
      2.2.2. Convergence 26
   2.3. Compensation les temps de latence 29
   2.4. Filtre par zone d'intérêt 31

CHAPITRE 2 – GENERALISATION DES OPTIMISATIONS 32

1. ARCHITECTURE GENERALE D'UN JEU 32
   1.1. Architecture de l'application client/serveur 32
   1.2. L'environnement de jeu 34
   1.3. Mise en place des optimisations du protocole 35
2. MODELISATION DES OPTIMISATIONS 36
   2.1. Modélisation de l'estimation des positions 36
   2.2. Modélisation de la convergence 37
   2.3. Modélisation de la compensation des temps de latence 38
   2.4. Modélisation du filtre par zone d'intérêt 40
3. EXTRACTION DES INTERFACES A PARTIR DES MODELES 41

CHAPITRE 3 – APPLICATION DES MODELES AVEC C# ET XNA 42

1. OUTILS UTILISES 42
   1.1. Langage C# et Framework XNA 42
   1.2. Bibliothèque réseau 42
   1.3. Environnement de développement 43
2. DETAIL DE LA STRUCTURE APPLICATIVE 44
   2.1. Architecture générale 44
      2.1.1. Architecture du serveur 45
      2.1.2. Architecture du client 47
3. MISE EN APPLICATION DANS UN JEU DE TIR SPATIAL 49
   3.1. Règles du jeu 49
      3.1.1. Principe du jeu 49
      3.1.2. Conditions de victoire 49
      3.1.3. Formation des parties 49
   3.2. Fonctionnement de l'application 50
      3.2.1. Structure des données 50
      3.2.2. Fonctionnement du Client 51
      3.2.3. Fonctionnement du Serveur 52
   3.3. Implémentation des optimisations 53
      3.3.1. Compression 53
      3.3.2. Estimation des positions 56
      3.3.3. Compensation des temps de latence 57
      3.3.4. Filtres par zone d'intérêt 60

CONCLUSION 62

SCHEMAS, FIGURES, ET TABLES 67

GLOSSAIRE 68

TABLE DES MATIERES 69

BIBLIOGRAPHIE 71

1. OUVRAGES 71
2. SITOGRAPHIE 71

ANNEXES 73

I. TRAITEMENT D'UN MESSAGE DU CLIENT PAR LE SERVEUR DU JEU I
II. EXEMPLE D'IMPLEMENTATION DE LA CLASSE MOVINGOBJECT II
III. EXEMPLE D'IMPLEMENTATION DE LA CLASSE MISSILE III
IV. EXEMPLE D'IMPLEMENTATION DE LA CLASSE SHIP IV
V. EXEMPLE D'IMPLEMENTATION DE METHODE REPLAY DE WORLDSTATE V

Bibliographie

1. Ouvrages

[1] MULHOLLAND Andrew *et al.*, *Programming Multiplayer Games*, Etats-Unis : Wordware, 2004.

[2] REED Aaron, *Learning XNA 3.0*, Etats-Unis : O'Reilly, 2009.

[3] RIEMER Grootjans, *XNA 3.0 Game Programming Recipes*, Etats-Unis : Apress, 2009.

[4] SANTOS LOBAO Alexandre *et al., Beginning XNA 3.0 Game Programming*, Etats-Unis : Apress, 2009.

[5] SINGHAL SK *et al., Networked Virtual Environments: Design and Implementation*, Etats-Unis : Addison Wesley, 1999.

[6] SMED Jouni *et al., Algorithms and Networking for Computer Games*, Wiltshire : John Wiley & Sons, 2006.

[7] VERNA Didier et al., *Urbi et Orbi : Unusual design and implementation choices for distributed virtual environments*, Etats-Unis : IOS Press, 2000.

2. Sitographie

[8] ARONSON Jesse. « Dead Reckoning: Latency Hiding for Networked Games ». [en ligne]. http://www.gamasutra.com/view/feature/3230/dead_reckoning_latency_hiding_for_.php (page consultée le 5 juillet 2010).

[9] BERNIER Yahn. « Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization ». [en ligne]. http://fabiensanglard.net/quakeSource/Lag_Compensation.htm (page consultée le 27 juillet 2010).

[10] BOOTH SIMPSON Zackary. « A Stream-based Time Synchronization Technique For Networked Computer Games ». [en ligne]. http://www.mine-control.com/zack/timesync/timesync.html (page consultée de 28 août 2010).

[11] CALDWEL Nick. « Defeating Lag With Cubic Splines ». [en ligne]. http://www.gamedev.net/reference/articles/article914.asp (page consultée le 09 juillet 2010).

[12] FIEDLER Glenn. « Gaffer on Games ». [en ligne]. http://gafferongames.com/networking-for-game-programmers/ (page consultée le 26 mars 2010).

[13] SIMPSON Jake. « Networking for Games 101 ». [en ligne]. http://www.gamedev.net/reference/articles/article1138.asp (page consultée le 26 mars 2010).

[14] SAWASHIMA Hidenari *et al.*, « Characteristics of UDP Packet Loss: Effect of TCP Traffic ». [en ligne]. http://www.isoc.org/INET97/proceedings/F3/F3_1.HTM (page consultée le 21 juin 2010).

[15] LUKIANOV Denis. « Advances WinSock Multiplayer Game Programming : Multicasting ». [en ligne]. http://www.gamedev.net/reference/programming/features/multicast/ (page consultée le 20 juin 2010).

[16] VALVE. « Source Multiplayer Networking ». [en ligne]. http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking (page consultée le 14 août 2010).

[17] INSEE. « Soixante ans de réduction du temps de travail dans le monde ». [en ligne]. http://www.insee.fr/fr/themes/document.asp?ref_id=ip1273 (page consultée le 7 mai 2010).

[18] INSEE. « Dépenses des ménages pour les loisirs ». [en ligne] http://www.insee.fr/fr/themes/document.asp?ref_id=ip1273 (page consultée le 7 mai 2010).

[19] AFJV. « Les chiffres du marché français des biens culturels ». *Agence Française pour le Jeu Vidéo* [en ligne]. http://www.afjv.com/press0902/090212_gfk_marche_entertainment.htm (page consultée le 7 mai 2010).

[20] AFJV. « Les jeux vidéo : une industrie culturelle française ». [en ligne]. http://www.afjv.com/press0606/060613_dossier_etude_jeux_video.htm (page consultée le 7 mai 2010).

[21] jeuxvideo.com. « Les 10 évènements qui ont marqué 2009 ». *www.jeuxvideo.com.* [En ligne]. http://www.jeuxvideo.com/dossiers/00011986/les-10-evenements-qui-ont-marque-l-annee-2009-3eme-sortie-de-call-of-duty-modern-warfare-2-008.htm (page consultée le 7 mai 2010).

Annexes

I. TRAITEMENT D'UN MESSAGE DU CLIENT PAR LE SERVEUR DU JEU I
II. EXEMPLE D'IMPLEMENTATION DE LA CLASSE MOVINGOBJECT II
III. EXEMPLE D'IMPLEMENTATION DE LA CLASSE MISSILE III
IV. EXEMPLE D'IMPLEMENTATION DE LA CLASSE SHIP IV
V. EXEMPLE D'IMPLEMENTATION DE METHODE REPLAY DE WORLDSTATE V

I. Traitement d'un message du client par le serveur du jeu

```
// traite un message de donnée en provenance d'un client
// prend en paramètre le message à rediriger
private void dataProcess(NetIncomingMessage msg)
{
   // récupère le type du message (premier byte), c'est un type énuméré
   DataType type = (DataType)msg.ReadByte();
   // redirige le message vers le service adapté
   switch(type)
   {
       // le client demande la liste des salles
       case DataType.serverListRooms:
           roomService.listRooms(msg);
           break;
       // le client demande la création d'une salle
       case DataType.serverCreateRoom:
           roomService.createRoom(msg);
           break;
       // le client demande à rejoindre une salle
       case DataType.serverJoinRoom:
           roomService.joinRoom(msg);
           break;
       // le client demande la liste des joueurs présents dans la salle
       case DataType.serverListPlayers:
           roomService.listPlayers(msg);
           break;
       // le client annonce qu'il quitte la salle
       case DataType.serverExitRoom:
           roomService.exitRoom(msg);
           break;
       // l'hote de la salle demande à lancer la partie
       case DataType.serverStartGame:
           roomService.startGame(msg);
           break;
       // le client demande à changer le pseudonyme du joueur
       case DataType.serverUpdatePlayer:
           playerService.updatePlayer(msg);
           break;
       // le client annonce qu'il est dans l'écran de jeu, prêt à démarrer
       case DataType.serverReadyToPlay:
           ServerGame g = ServerGame.getGame(msg);
           if (g != null)
               g.addPlayer(msg);
           break;
       // le client envoie les dernières commandes saisies par le joueur
       case DataType.serverClientDataPacket:
           ServerGame g2 = ServerGame.getGame(msg);
           if (g2 != null)
               g2.handlePlayerInput(msg);
           break;
   }
}
```

II. Exemple d'implémentation de la classe MovingObject

```
public abstract class MovingObject
{
    public Vector2 position { get; set; }
    public Vector2 acceleration { get; set; }
    public Vector2 speed { get; set; }
    public float   angle { get; set; }

    // Estime l'état de l'objet dans "timeInMs" millisecondes.
    // On suppose l'accélération contante et l'angle fixe.
    public MovingObject estimateState(int timeInMs)
    {
        // on fonctionne en unité/seconde
        float timeInSec = timeInMs / 1000;

        // recopie l'objet actuel
        MovingObject futurState = new MovingObject(this);

       // applique la formule de dérivée pour deviner la prochaine position
        futurState.position =
            futurState.position
            + futurState.speed * timeInSec
            + 0.5 * futurState.acceleration * (timeInSec * timeInSec);

        // changement de la vitesse
        futurState.speed =
            futurState.speed
            + futurState.acceleration * timeInSec;
    }

    // déplace l'objet courant vers une position intermédiaire
    // de sa trajectoire.
    public void interpolate(
        MovingObject initialState, // Etat au début de la trajectoire
        MovingObject finalState,   // Etat à la fin de la trajectoire
        int durationInMs,          // durée totale du déplacement
        int timeInMs               // état d'avancement actuel
    ) {
        // avancement (en %)
        float progress = (timeInMs / durationInMs);
        // Par défaut, on utilise une interpolation linéaire
        // chaque valeur est interpolée.
        this.position =
          (finalState.position - initialState.position) * progress;
        this.angle =
          (finalState.angle - initialState.angle) * progress;
        this.acceleration =
          (finalState.acceleration - initialState.acceleration) * progress;
        this.speed =
          (finalState.speed - initialState.speed) * progress;
    }
}
```

III. Exemple d'implémentation de la classe Missile

```
public class Missile : MovingObject
{
    // Estime l'état du Missile dans "timeInMs" millisecondes.
    // On sait l'accélération nulle et l'angle constant.
    public override MovingObject estimateState(int timeInMs)
    {
        // on fonctionne en unité/seconde
        float timeInSec = timeInMs / 1000;

        // recopie l'objet actuel
        Missile futurState = new Missile(this);

        // déplacement à vitesse fixe
        futurState.position =
            futurState.position
            + futurState.speed * timeInSec;
    }

    // déplace l'objet courant vers une position intermédiaire
    // de sa trajectoire.
    public override void interpolate(
        MovingObject initialState, // Etat au début de la trajectoire
        MovingObject finalState,   // Etat à la fin de la trajectoire
        int durationInMs,          // durée totale du déplacement
        int timeInMs               // état d'avancement actuel
    )
    {
        // avancement (en %)
        float progress = (timeInMs / durationInMs);
        // trajectoire rectiligne et constante : interpolation linéaire
        this.position =
          (finalState.position - initialState.position) * progress;
    }
}
```

IV. Exemple d'implémentation de la classe Ship

```
class Ship : MovingObject
{
    public Player player {get;set;}

    // Estime l'état de l'objet dans "timeInMs" millisecondes.
    // On pourra utiliser la dernière commande du joueur.
    abstract public MovingObject estimateState(int timeInMs)
    {
        // recopie l'objet actuel
        Ship futurState = new Ship(this);

        // avance l'objet comme le ferait le serveur si le joueur
        // maintenait ses commandes dans leur état actuel.
        futurState.move(player.lastInput, timeInMs);
    }

    //Ppas de surcharge de la méthode interpolate,
    // l'interpolation linéaire nous convient.
    public void move(InputState input, int timeInMs){
        // Ici les instructions utilisées par le serveur pour
        // réaliser le déplacement de l'objet.
        // Implémenter selon le comportement voulu.
    }
}
```

V. Exemple d'implémentation de méthode replay de WorldState

```
public static void replay(
  Player player, // joueur dont on met à jour les données
  HistoryList<InputState> commandHistory, // liste des commandes à jouer
  HistoryList<WorldState> worldStateHistory // liste des états à impacter
)
{
    // pour chaque commande on rejoue le déplacement
    foreach (InputState input in playerLastInputs)
    {
        // récupère l'état de l'environnement au moment de la commande
        // on prend les valeurs de l'ancien état qu'on fait évoluer pour
        // obtenir celles du suivant.
        WorldState worldStateInThePast =
            history[input.time] ;
        WorldState followingWorldStateInThePast =
            history.getNext(input.time);

        // tant qu'il existe un état concerné par cette commande
        // on continue de modifier les états successifs
        while (followingWorldStateInThePast.time < input.time +
input.duration)
        {
            // durée de la commande jusqu'au prochain état
            // ou jusqu'au début de la suivante.
            int duration = Math.Min(
                followingWorldStateInThePast.time - input.time,
                input.duration
            );

            // position au moment du changement de commande
            Ship shipInThePast =
                worldStateInThePast.ships[player.ship.id];
            // position après la réalisation de la commande
            Ship followingShipInThePast =
                followingWorldStateInThePast.ships[player.ship.id];
            // annulation de la position passée suivante
            followingShipInThePast.copy(shipInThePast);
            // recalcul de la position pour l'état suivant la commande
            followingShipInThePast.move(input, duration);

            // on décale les états passés, celui qui viens d'être
            // recalculé devient la référence
            worldStateInThePast =
              followingWorldStateInThePast;
            // l'état suivant est mis en modification.
            followingWorldStateInThePast =
              history.getNext(worldStateInThePast.time);
            // la durée est mise à jour
            duration =
              followingWorldStateInThePast.time - worldStateInThePast.time;
        }

    }
}
```