Besoin de performances revue technique

Pour mieux comprendre le défi que représente la réalisation d'un jeu massivement multi-joueurs, nous allons décrire dans cette partie les enjeux techniques à prendre en considération. Nous allons également passer en revue les diverses techniques actuellement utilisées pour essayer de répondre aux contraintes très fortes auxquelles sont sujettes ces applications.

Le milieu académique a tardé à s'intéresser au domaine des jeux en ligne sur Internet. Jusqu'à très récemment, les travaux exploitables dans le cadre des jeux massivement multi-joueurs étaient principalement des communications issues du monde industriel. La plupart des travaux antérieurs à l'année 2000 avaient pour objectif principal les applications de simulation militaire en réseau local, ou ne s'intéressaient aux jeux multi-joueurs qu'en tant qu'application de démonstration pour des travaux concernant l'optimisation de protocoles réseau, ou de synchronisation d'application distribuée. Un grand nombre des travaux dans ces domaines sont applicables au domaine des jeux massivement multi-joueurs : il y a une intersection commune avec toutes les problématiques rencontrées dans le développement des types d'applications pré-citées. Une bonne synthèse des travaux antérieurs à 2000 concernant les mondes virtuels peut être trouvée dans le livre écrit par Sandeep Singhal et Michael Zyda [93].

Le problème avec les jeux massivement multi-joueurs commerciaux, c'est

qu'ils se frottent à toutes ces problématiques en même temps. Beaucoup de solutions développées dans le cadre de la simulation militaire ou de la réalité virtuelle pour une problématique donnée ne sont pas satisfaisantes car elles empêchent d'en régler une autre. Publiée en 2002, une étude [59] faite par Jouni Smed, Timo Kaukoranta et Harri Hakonen tente d'établir un lien entre la recherche académique et les problèmes rencontrés par les industriels réalisant les jeux multi-joueurs. Les auteurs y définissent les points communs entre les caractéristiques des applications de simulations traitées dans l'académique et celles des jeux multi-joueurs sur Internet. Ils définissent les problématiques spécifiques à ce dernier domaine d'application, dont les innovations sont à l'époque principalement issues du monde industriel.

Dans ce chapitre, nous allons commencer par décrire toutes les difficultés et contraintes techniques que l'on rencontre dans le cadre de la réalisation d'un jeu massivement multi-joueurs.

Après avoir identifié lesquelles de ces difficultés sont en rapport avec la définition des interactions du *game-play*, nous ferons le tour des solutions permettant de jongler entre les différents impératifs de performances : la force des contraintes compliquant la réalisation d'un jeu massivement multi-joueurs dépend en grande partie du style de jeu que l'on veut réaliser. Suivant le *game-play*, certaines peuvent prendre plus d'importance que d'autres, et on peut parfois négliger la qualité d'une des caractéristiques techniques au profit d'une autre. Pour un état de l'art plus général, non centré comme ici sur la mise au point des interactions du *game-play* et abordant d'autres difficultés techniques comme l'utilisation du son et la mobilité, on peut se référer à [74].

2.1 Caractéristiques techniques d'un jeu multi-joueurs

Dans cette partie, nous allons définir ce qu'est un jeu massivement multi-joueurs en termes techniques, en donnant une liste de ce qui caractérise ces applications distribuées sur Internet.

2.1.1 Persistance

Les jeux massivement multi-joueurs sont des applications persistantes. Cela signifie que le monde virtuel est accessible vingt-quatre heure sur vingt-quatre et sept jours sur sept. Une fois que le jeu est lancé, la partie ne s'arrête pas, et les joueurs peuvent rejoindre à tout moment cet univers qui évolue en permanence selon leurs actions.

En réalité, la plupart des jeux multi-joueurs prévoient des maintenances régulières ou sporadiques, lors desquelles les serveurs sont arrêtés, afin de rajouter du contenu au jeu, ou de réparer des bugs.[1] Mais la plupart des modifications de contenu simples, comme le changement du comportement d'un personnage non-joueur peuvent souvent s'effectuer sans avoir à arrêter l'application. Par exemple, lorsque les concepteurs du jeu ont prévu cette possibilité à l'avance en utilisant des dispositifs de modification dynamique des scripts permettant de contrôler ces comportements.

Il est particulièrement important de disposer d'une bonne politique de sauvegarde de l'état du monde : lors du re-démarrage de l'application, le jeu doit se retrouver dans le même état que lorsqu'il s'est arrêté. Comme l'arrêt des serveurs peut ne pas être volontaire, mais conséquence d'un problème technique quelconque, il est également important de bien gérer la sauvegarde périodique des éléments les plus importants de l'état du monde virtuel.

Afin de garantir le meilleur service possible, il est indispensable que l'application soit facile à maintenir et à faire évoluer. Un jeu massivement multi-joueurs est exploité pendant des années et le coût de maintenance d'un jeu mal fait peut réduire grandement les bénéfices.

Le principal risque si la persistance de l'application est mal gérée est le *rollback* (voir glossaire) : en cas de corruption de l'état du jeu, pour des raisons techniques ou du fait d'une intervention extérieure malveillante exploitant une faille de l'application, il est parfois nécessaire de devoir revenir à un état antérieur, dont la sauvegarde est intacte. Le rollback est craint à la fois par les joueurs, qui risquent ainsi de perdre de nombreuses heures de jeu (et donc toute leur progression dans le monde, en termes de compétences, d'acquisition d'objets et d'argent virtuel), et par les sociétés exploitant les

[1] C'est le fameux «*Patch day don't play*» d'Everquest, dont les opérations de maintenance duraient fréquemment plus longtemps que prévu.

jeux, qui n'aiment pas fâcher leurs clients qui prennent souvent leur vie virtuelle très à cœur.

2.1.2 Synchronisation dans une application distribuée

Afin de pouvoir proposer une bonne immersion aux joueurs, l'état global du jeu doit être partagé par tous les hôtes de l'application à tout instant : dans un monde idéal, chaque joueur devrait voir en permanence sur son ordinateur les données représentant l'état actuel du jeu et partager cet état avec tous les autres joueurs.

Cette description est un paradoxe en elle-même : chaque changement du monde sur un des hôtes de la distribution doit être répercuté par le réseau sur les autres hôtes de la distribution. La propagation d'une information par un réseau prend du temps, même si elle peut être rapide à l'échelle humaine. Les informations ont également un certain volume, qui peut devenir handicapant dans certaines conditions de débit, et Internet, en tant que réseau hétérogène et au comportement souvent imprévisible, n'arrange pas les choses.

Ceci nous mène directement au problème de la consistance de l'accès aux données dans une application distribuée, c'est-à-dire de la possibilité d'avoir un état du jeu commun entre tous les hôtes de l'application énoncé dans [93] : *Il est impossible d'autoriser l'état dynamique partagé à changer fréquemment tout en garantissant que tous les hôtes d'une application accèdent simultanément à des versions identiques du monde virtuel.*

Il faut donc être capable de faire en sorte que ce problème de synchronisation ne soit pas perceptible par les joueurs, ce qui peut être particulièrement délicat, en particulier lorsqu'il s'agit de jeux où la réactivité du joueur a une grande importance : par exemple dans un FPS, lorsque le décalage dépasse le temps du réflexe humain, il devient très vite difficile de jouer dans de bonnes conditions.

2.1.3 Sécurité

Dans un jeu massivement multi-joueurs, le problème de la sécurité est crucial. Les modèles commerciaux, axés sur un abonnement pour l'accès au

jeu, font qu'on y retrouve tous les problèmes communs aux applications de services sur Internet. Sécurisation de l'authentification du joueur, afin qu'un autre ne puisse accéder à ses données et avatars personnels, paiement sécurisé, résistance des serveurs aux attaques réseau classiques, sont les caractéristiques communes aux applications de services Internet et aux jeux massivement multi-joueurs.

Aux problèmes de sécurité classique s'ajoutent les conséquences liées à la triche : si le jeu laisse la porte ouverte aux tricheurs, les joueurs honnêtes partiront très vite, frustrés, et c'est la fin du monde (virtuel).

En effet, comme un jeu en ligne se déroule dans un monde persistant, où l'économie et l'évolution des personnages selon les actions des joueurs sont soigneusement, délicatement mises au point, l'équilibre est fragile. Un joueur qui découvrirait un moyen de faire de l'argent facile alors que ça n'est pas prévu par les concepteurs peut ruiner l'économie du monde virtuel : l'argent circule, inonde le jeu, et finalement ne signifie plus rien. La moindre faille dans le jeu sera fatalement exploitée par les joueurs.

Un déséquilibre du monde virtuel a bien plus de conséquences dans le cadre d'un jeu persistant que dans un jeu multi-joueurs qui se joue par sessions. En effet, la seule manière de le rééquilibrer, une fois la faille de sécurité corrigée, sera souvent de revenir à un état antérieur non corrompu du jeu en effectuant un *rollback*.

Il est assez délicat de trancher sur les cas relevant de la triche ou non dans les jeux massivement multi-joueurs. Par exemple, dans certains jeux offrant un style de combat «réaliste» en ville, la technique qui consiste pour un joueur à se dissimuler derrière un élément du décor en attendant de pouvoir tirer dans le dos des cibles faciles sans prendre de risques grâce à une définition approximative des collisions de cet élément peut être indifféremment considérée selon les jeux comme une tactique maligne, un manque de fair-play, ou un abus d'une faille dans la construction du monde.

Jianxin Jeff Yan et Hyun-Jin Choi ont défini une taxinomie de la triche dans les jeux massivement multi-joueurs [103] que nous allons discuter dans cette partie. En effet, quelques uns des types de triche retenus par les auteurs nous semblent relever du cas par cas. D'autres nous semblent plutôt relever du *grief-play* (voir glossaire), qui peut être défini comme le fait d'adopter un style de jeu autorisé par le *game-design*, mais à en exploiter les possibilités

pour gâcher l'expérience des autres joueurs. On peut trouver une taxinomie du *grief-play* pour des jeux comme Everquest dans [42]. Nous nous appuierons également sur les exemples de fraudes constatées dans [23]. Les montants impliqués dans certaines fraudes concernant les jeux massivement multi-joueurs évoquées dans cette étude atteignent plusieurs milliers de dollars, et les auteurs évoquent une moyenne de 200 fraudes criminelles recensées par jour.

2.1.3.1 Tricher à l'aide d'un complice :

Yan et Choi donnent comme exemple le cas du jeu de bridge, où deux joueurs complices (ou un même joueur incarnant deux avatars distincts) peuvent connaître leurs mains respectives sans que les autres s'en rendent compte. Nous ajoutons à cette catégorie un exemple qui nous paraît plus représentatif des jeux massivement multi-joueurs : la technique du *power levelling* (voir glossaire) consiste, dans un jeu massivement multi-joueurs, à utiliser les capacités d'un avatar puissant pour faire progresser beaucoup plus rapidement qu'à la vitesse prévue par les concepteurs un avatar plus faible. Par exemple, l'aider à obtenir des objets qu'il n'aurait pas pu acquérir seul avant plusieurs dizaines d'heures de jeu, partager en faisant équipe avec lui l'expérience acquise en combattant un personnage non joueur qu'il n'a aucune chance de vaincre seul. Cela peut aussi être, pour les jeux incluant une notion de prestige dans l'habilité au combat entre joueurs, laisser un avatar se faire tuer un grand nombre de fois par un autre afin d'acquérir sans risque les galons de la gloire.

Le *power levelling* est possible dans beaucoup de jeux, et n'est souvent pas considéré comme de la triche, même s'il peut être ressenti comme une injustice ou un manque de fair-play par les joueurs qui n'en profitent pas.

2.1.3.2 Tricher en abusant des mécanismes du jeu :

Un exemple très commun de ce type de triche, donné par Yan et Choi, est la fuite : si quelque chose commence à mal tourner dans le jeu, que le joueur risque de perdre de l'expérience ou des possessions virtuelles car il n'a plus le dessus, il se déconnecte brutalement et disparaît du monde virtuel. L'action n'étant pas allée jusqu'à son terme, le joueur sauve son avatar par des moyens qui ne font pas partie des mécanismes du *game-play*. Nous ajoutons à cette

catégorie les phénomènes de *kill steal* (voir glossaire), consistant par exemple dans certains jeux à attendre qu'un joueur aie presque fini d'achever un personnage non-joueur pour donner le coup fatal afin d'emporter l'expérience ou les récompenses associées, et le *loot steal* (voir glossaire), consistant dans certains jeux à ramasser très vite les récompenses associées au succès d'un joueur lors d'un combat, avant que ce dernier n'ait eu le temps de le faire.

Comme la triche par complicité, la perception de ces comportements n'est souvent pas considérée comme de la triche, au pire comme un comportement déviant qui ne vaudra pas à ses adeptes une immense popularité dans le monde virtuel. Ils sont plus souvent considérés comme le problème du *player killer* (voir glossaire), ces joueurs dont la principale motivation est de ruiner le plaisir de jeu des autres joueurs : des comportements nuisibles au plaisir de la plupart des joueurs, mais légitimes. Nous pensons qu'ils rentrent plutôt dans la catégorie du *grief-play* que de la triche à proprement parler.

2.1.3.3 Triche liée au marchandage entre joueurs d'objets virtuels

Certains objets virtuels d'un jeu massivement multi-joueurs sont extrêmement rares et difficiles à obtenir. Or, à cause de l'importance que prend parfois pour les joueurs leur course au prestige, avatars et objets virtuels se vendent aux plus offrants à des prix jugés indécents par le commun des mortels sur des services d'enchère en ligne. Les abus de confiance entre les joueurs peuvent donc être très mal ressentis par les joueurs floués, lorsque les jeux ne permettent pas de contrôler les échanges.

Bien sur, il ne s'agit pas toujours d'argent réel qui est en jeu, mais parfois tout simplement des escroqueries virtuelles entre joueurs. Là encore, malgré la frustration du joueur virtuellement escroqué, la définition de ce comportement comme étant de la triche n'est pas tout à fait claire et nous semble plutôt relever du *grief-play*.

2.1.3.4 Extorsion frauduleuse des identifiants de connexion des joueurs

Là encore, la principale motivation de ce type de triche est la possession des objets du monde virtuel. Dans [23], les auteurs répertorient un certains

nombre de pratiques frauduleuses voire criminelles visant à l'obtention des paramètres de connexion d'un joueur, allant de l'agression physique et du kidnapping à la classique utilisation d'un virus troyen (ce type de programme s'installe sur un ordinateur comme les virus habituels et envoie des informations personnelles par le réseau à une personne qu'on peut alors supposer malintentionnée), en passant par les attaques «dictionnaire» (il s'agit également d'attaques classiques, basées sur le fait que les utilisateurs répugnent à utiliser des mots de passe compliqués, et qui consistent à essayer une liste de mots et de leurs variantes évidentes, inversées ou en jouant avec les lettres majuscules et minuscules).

Ici, plutôt que de la triche, les pratiques relèvent du pénal et des problèmes de sécurité réseau classique.

2.1.3.5 Escroqueries relevant de l'abus de confiance pour l'obtention des identifiants de connexion

Les habitués des services de paiement en ligne ou des sites marchands ont tous reçu, un jour ou l'autre, des e-mails les incitant à envoyer leurs identifiants pour l'accès sécurisé à ces services par retour de courrier sous un prétexte technique quelconque et fallacieux. Ce type d'abus de confiance est bien évidemment applicable aux jeux massivement multi-joueurs.

Ces méthodes externes au jeu lui-même sont clairement à classer du côté des pratiques frauduleuses et non de la triche, si on oublie ce simple fait fréquent : les joueurs ont souvent tendance à révéler eux même en toute confiance et sans incitation leurs identifiants de connexion à leurs camarades de jeux car certains avatars ont des spécialités indispensables au jeu en équipe, et il est pratique pour les autres membres de l'équipe de pouvoir disposer de cet avatar à volonté. Les abus de confiance se produisent plus fréquemment lorsqu'on se croit protégé par l'anonymat que procurent les mondes virtuels.

2.1.3.6 Triche à l'aide d'attaques *Déni de Service* contre les autres joueurs

Les attaques réseau de type *Déni de Service* sont bien connues dans le domaine de la sécurité sur Internet.

Une telle attaque peut tout simplement être réalisée après l'obtention des identifiants de connexion d'un joueur par les méthodes décrites au paragraphe précédent, l'empêchant ainsi de se connecter lui-même au jeu. Bloquer ainsi un joueur indispensable à une équipe contre laquelle on est en conflit dans le cadre du jeu peut suffire à déséquilibrer injustement le jeu.

Les techniques plus classiques d'attaques de ce type, qui consistent à saturer la connexion d'un joueur en lui envoyant des messages, pour ralentir sa connexion et ainsi avoir un avantage certain quand une bonne réactivité du joueur est nécessaire, par exemple lors de combats dans un FPS. L'adversaire aura beau avoir des réflexes, si soudainement sa connexion réseau est ralentie, il sera très handicapé. Ce type d'attaque implique néanmoins que le tricheur puisse connaître l'adresse IP de son adversaire.

2.1.3.7 Triche due au manque de confidentialité

Lorsque les commandes sont envoyées en clair par le réseau des machines des joueurs aux serveurs, une autre attaque courante consiste à analyser les messages transitant sur le réseau, et à les modifier au passage ou à en insérer d'autres. L'obtention des identifiants de connexion est bien sûr sujette à cette attaque, mais aussi tout ce qui concerne les actions des joueurs dans le monde virtuel, afin d'influer et de modifier l'état du monde pour en gagner un avantage.

2.1.3.8 Triche due à un manque d'authentification

Nous avons évoqué plus haut les problèmes liés à l'authentification des joueurs par leurs identifiants de connexion, mais il est aussi important que le serveur du jeu puisse être authentifié par les logiciels clients utilisés par ces derniers pour se connecter : en effet, pour peu que quelqu'un réussisse à orienter les joueurs vers un serveur modifié à cet effet, il peut collecter les

identifiants de connexion des participants.

2.1.3.9 Problèmes de sécurité internes aux sociétés

Toujours dans [103], les auteurs définissent une catégorie à part entière provenant des risques associés aux employés des sociétés exploitant les jeux : en effet, il est souvent prévu un mode de super-utilisateur, associé à des droits particuliers, comme la création d'objets virtuels rares et convoités, et les auteurs citent des cas de fraudes réelles constatées.

2.1.3.10 Triche par modification du logiciel utilisé par les joueurs, ou par modification des données du jeu

Ce type de triche n'est pas nouveau, puisque la modification des programmes de jeu traditionnels pour pouvoir les finir plus facilement est une chose courante. Mais alors que dans les jeux solitaires, la triche ne nuit après tout qu'au tricheur qui réduit ainsi la durée de vie de sa partie pour le simple plaisir de battre l'ordinateur, la modification du programme nuit désormais aux autres joueur : les premiers *aimbots* (voir glossaire) pour le jeu Quake permettaient à l'utilisateur de toujours viser juste.

Voici un autre exemple très simple de ce type de triche : le joueur modifie son programme client, afin que tous les avatars représentant les autres joueurs lui soient désormais affichés en costume d'oursin bardé de piques très longues dans tous les sens. Désormais, il lui sera très facile d'échapper à ses opposants, cachés en embuscade au coin d'une rue derrière un mur, puisque les piques du costume, passant à travers les autres éléments du décor, les signaleront.

Même si ce type de triche est plus complexe à réaliser et nécessite un minimum de compétence technique, à partir du moment où le logiciel modifié existe, il est à parier qu'il s'écoulera très peu de temps avant qu'il soit disponible à beaucoup de joueurs, tant il est vrai qu'il est plus valorisant encore pour les tricheurs d'expliquer la manière très intelligente avec laquelle ils ont procédé que de garder le secret de leurs performances pour eux.

2.1.3.11 Exploitation de bugs, ou de faiblesses de conception

Les auteurs de [103] font de l'exploitation des faiblesses du jeu une catégorie de triche à part entière. Il nous semble cependant qu'il y a une intersection commune avec bien des catégories précédemment citées. Par exemple, si une personne est capable de récupérer par simple espionnage des paquets envoyés au travers du réseau parce que les mots de passe ne sont pas encodés, ou pas de manière assez sûre, ça n'est pas seulement un manque de confidentialité (voir partie 2.1.3.7, page 43) mais une faiblesse évidente de conception.

De même, la plupart des triches utilisant une version modifiée du logiciel client se basent sur une de ses faiblesses, comme par exemple certaines techniques de masquage de la latence qui rendent trop facilement détectables sur la machine du joueur à l'instant t des données qu'il n'est censé exploiter qu'à l'instant $t + 1$.

Insister sur la nécessité d'avoir une application sûre, vérifiée et de qualité n'est néanmoins pas inutile dans la mesure où les conséquences peuvent être rapidement impressionnantes. Dans [23] les auteurs donnent l'exemple d'une personne s'étant faite prendre la main dans le sac en 2001, après avoir vendu un objet du monde virtuel contre monnaie sonnante et trébuchante, puis utilisé une faille du serveur pour le reprendre à son nouveau propriétaire.

2.1.4 Passage à l'échelle

Dans le domaine des applications distribuées sur Internet, le passage à l'échelle est défini comme la capacité pour l'application de pouvoir rester performante malgré une montée en charge des ressources utilisées et du nombre d'utilisateurs. Même si les performances optimales ne peuvent être maintenues, des politiques de fonctionnement en mode dégradé, mais permettant toujours des conditions de jeu acceptables, doivent être prévues pour assurer la satisfaction de l'utilisateur.

Internet est un exemple d'application distribuée possédant de bonnes propriétés de passage à l'échelle : avec le nombre exponentiellement croissant d'utilisateurs, le réseau reste utilisable et performant. Cependant, même cette application distribuée «ultime» n'est pas exempte de problèmes liés à une forte augmentation des utilisateurs : en effet, le nombre de bits identifiant

les adresses IP commence à se montrer tellement insuffisant par rapport à l'ampleur qu'a pris le réseau mondial qu'il existe des situations de pénurie dans certains pays pauvrement dotés en nombre d'adresses IP, notamment sur le continent africain, ce qui a provoqué entre autre le projet IPv6 du nouvel Internet où les adresses seront désormais codées sur 128 bits [25].

Bien sûr, le problème se présente différemment pour les jeux massivement multi-joueurs. Tout d'abord, aucun ne prétend atteindre l'échelle d'Internet, les ambitions sont moindres en terme d'utilisateurs. L'intérêt d'atteindre un nombre de joueurs simultanés relevant du million est discutable en terme de *game-play*.

Selon les jeux, l'objectif affiché se compte en milliers de connexions simultanées (pour des jeux comme Everquest), ou en centaines (pour des jeux comme Neocron). La taille du monde dans lequel le joueur évolue a également son importance : le monde de Neocron serait surpeuplé s'il devait contenir le même nombre de joueurs qu'Everquest.

De plus, quand on parle de passage à l'échelle pour des jeux massivement multi-joueurs, le passage à l'échelle en nombre de joueurs simultanément présents dans le monde virtuel n'est pas le seul aspect à considérer. En effet, les joueurs ont souvent tendance à regrouper leurs avatars en masse dans certaines régions du monde, parce qu'elles présentent des facilités pour le commerce, parce que ce sont celles où ont lieu les combats entre les joueurs, ou que ce sont des zones hostiles où il est plus facile de trouver des personnages non-joueurs à combattre par exemple. Il faut que ces régions puissent également passer à l'échelle.

Les problèmes concrets posés par le passage à l'échelle ne concernent pas seulement la conservation des performances techniques dans le cadre des jeux multi-joueurs, mais sont aussi liés au fait que les concepteurs aiment à rendre ces jeux les plus réalistes possible. Par exemple, les avatars des joueurs ont un corps qui se comporte dans l'espace virtuel comme s'il était solide dans des jeux comme Neocron, ce qui n'est pas le cas dans Everquest. Les avatars des joueurs peuvent donc se cogner les uns contre les autres. En cas de surpeuplement d'une région du monde virtuel, il devient très vite difficile de se déplacer, et on peut ressentir très vite le même sentiment d'agoraphobie. Ce sentiment peut être également lié au «bruit» ambiant, les joueurs communiquant souvent entre eux par le moyen d'interfaces textuelles qui peuvent rendre difficile le suivi des conversations lorsque tout ce petit monde parle

en même temps.

2.1.5 Ressources d'une application distribuée sur Internet

Toutes les propriétés que nous avons précédemment décrites sont difficiles à réaliser simultanément pour une raison très simple : comme toute application informatique, un jeu massivement multi-joueurs consomme des ressources physiques qui sont limitées, et qui ont un coût : les ressources d'Internet et des machines hôtes de l'application distribuée.

Première difficulté, un jeu massivement multi-joueurs se joue sur Internet, qui est par essence un système distribué asynchrone [26] : on ne peut ni borner le temps nécessaire à la transmission des messages, ni s'appuyer sur une horloge partagée par tous les hôtes. Ce sont les protocoles classiques développés pour Internet qui permettent d'établir une bonne organisation des communications et d'y maintenir un semblant d'ordre.

Dans [26], les caractéristiques de performances d'un réseau sont définies comme suit :

- la *latence* (L) est le délai entre l'envoi d'un message par un processus et le début de sa réception par un autre processus. Elle peut être mesurée en comptant le temps écoulé entre l'envoi et la réception d'un message vide ;
- le *taux de transfert* (T) des données est la vitesse à laquelle les données peuvent être transférées d'un ordinateur à l'autre du réseau, et se décompte habituellement en bits par secondes ;

Dans le cas d'Internet, la latence est souvent le facteur clé des performances, car sa valeur va dépendre du chemin emprunté par le message, de routeur en routeur, et des problèmes de congestion éventuellement rencontrés en route. Le taux de transfert point à point des données dans un réseau est par contre une caractéristique physique de ce dernier.

Le *temps de transfert* tt d'un message contenant un nombre n de bits entre deux ordinateurs est alors défini par : $tt = L + n/T$, s'il ne s'agit pas de message trop longs qui seront alors souvent fragmentés selon les technologies utilisées.

La bande passante système d'un réseau est le volume total d'information qui peut passer par ce réseau en un temps donné. Cette caractéristique diffère du taux de transfert des données dans la mesure où elle est sujette à des variations selon la congestion du réseau et l'utilisation simultanée des différents canaux de communication pour un même message. Pour simplifier, la bande passante concerne principalement les réseaux hétérogènes comme Internet alors que le taux de transfert représente une mesure point à point entre deux composants physiques du réseau. Les deux notions peuvent correspondre sur réseau local.

Enfin, il ne faut pas oublier que les jeux massivement multi-joueurs sont également des programmes complexes sur chaque hôte de l'application distribuée, et que les programmes utilisent du *temps processeur* et de la *mémoire* pour effectuer tous leurs calculs.

La consommation de ces ressources augmente en fonction du nombre de participants à l'application, et des ressources insuffisantes peuvent poser problème pour le passage à l'échelle de l'application et pour la satisfaction de ses contraintes de synchronisation. En effet, plus de machines signifie plus de communications pour garantir la cohérence du monde virtuel, et donc un risque de congestion du réseau ne disposant pas de suffisamment de bande passante. De même, plus de joueurs signifie plus de calculs et de temps processeur consommé.

Dans [93], les auteurs décrivent, en se plaçant à un niveau un peu plus abstrait, l'ensemble des ressources consommées par une application de type monde virtuel par l'équation $Ressources = I \times D \times B \times V \times P$ où :

- I est le nombre de paquets d'informations transmis au travers du réseau de l'application ;
- D est le nombre moyen de destinataires pour chaque paquet d'information ;
- B est la quantité moyenne de bande passante nécessaire pour acheminer un paquet à destination ;
- V est la vitesse (inversement proportionnelle à la latence acceptable pour l'application) requise pour la délivrance d'un paquet d'information à destination ;
- P est le nombre moyen de cycles processeur requis pour recevoir et traiter un paquet d'informations.

Cette équation définit l'utilisation des ressources d'un monde virtuel : toute

technique visant à économiser la consommation d'une des ressource augmentera la consommation d'une autre de ces ressources.

2.1.6 Qu'est-ce que la jouabilité ?

Pour conclure sur cette description des caractéristiques techniques d'un jeu massivement multi-joueurs, nous définissons la *jouabilité* comme la qualité de l'immersion du joueur dans le jeu et son plaisir à évoluer selon le *game-play* élaboré par les concepteurs. La jouabilité s'obtient par une solution à un degré adéquat des contraintes techniques décrites tout au cours de cette section.

Atteindre la jouabilité se fait différemment selon les types de jeux, et il est nécessaire de faire correspondre les propriétés techniques des jeux avec le *game-play* désiré. Ainsi, les études présentées dans [9] et [85] permettent d'évaluer l'effet de mauvaises conditions de réseau sur les performances des joueurs dans *Unreal Tournament* 2003. De même, plusieurs études ont mesuré la qualité de la synchronisation nécessaire selon des *game-play* typiques :

- Dans [80], les auteurs étudient un jeu de course de voitures. Ils concluent que si le temps total de non-synchronisation observable par les «coureurs» dépasse 100ms les joueurs commencent à déceler le décalage, sans en être encore trop gênés, tandis que 50ms de décalage leur procurent une sensation d'instantanéité.
- Dans [91], les auteurs évaluent la latence acceptable à 500ms, dans un jeu de stratégie en temps-réel comme Warcraft III.
- Enfin, dans un jeu du style First Person Shooter, qui se joue beaucoup sur la rapidité de réaction des participants, les joueurs sont excédés dès que la latence avoisine les 150ms [3].

2.1.7 Synthèse

Après avoir défini les caractéristiques d'un jeu massivement multi-joueurs, nous allons étudier dans les sections qui suivent les différentes techniques actuellement utilisées pour tenter de donner des solutions aux problèmes techniques que nous venons de décrire. Nous allons voir qu'aucune solution n'est idéale, et que tenter de diminuer l'utilisation d'une des composantes

de l'équation des ressources ou de répondre à une contrainte forte sur les caractéristiques évoquées est toujours une affaire de compromis qui aura un impact négatif sur une autre caractéristique.

Certaines solutions aux problèmes que nous venons de décrire ne rentrent pas dans le cadre de cette thèse, et nous ne les aborderons pas dans ce qui suit. Elles concernent principalement la sécurité de l'application, et les problèmes de triche.

Tout d'abord, les techniques classiques de protection contre les attaques réseau auxquelles sont sujets les jeux comme toutes les autres applications Internet revêtent un champ d'étude très vaste, et déjà largement traité. Ces techniques (cryptage, utilisation de pare-feux, protection contre les attaques classiques, détection de l'intégrité du logiciel client, utilisation de traces pour détecter les comportements aberrants) sont des domaines que nous avons choisi de ne pas traiter pour mieux mettre en valeur les problématiques propres aux jeux massivement multi-joueurs.

De plus, bon nombre de solutions à des problèmes techniques se règlent par des choix de *game-design* : les jeux dont le *game-play* favorise le rassemblement d'un grand nombre d'avatars choisissent parfois de ne pas leur donner de modèle de collision (les avatars ne peuvent pas se pousser), d'autres comme City of Heroes [24] dupliquent les zones lorsqu'elles deviennent trop peuplées. De même, on peut poser une limite du nombre de joueurs pouvant se connecter simultanément, ce qui permet d'éviter les problèmes de passage à l'échelle non prévus. Des animations graphiques sur le logiciel client occupent le joueur pour qu'il ne remarque pas la latence. Enfin, pour empêcher un joueur qui réalise qu'il est en train de perdre l'avantage de s'échapper en se déconnectant brutalement, la plupart des jeux massivement multi-joueurs implémentent une solution de *game-design* qui consiste à laisser présent dans le monde virtuel l'avatar d'un joueur quelque temps après sa déconnexion, inactif puisqu'il n'est plus contrôlé par le joueur, mais toujours vulnérable. Et tant pis pour le joueur qui perd sa connexion au cours d'un combat à cause d'un problème matériel.

Certaines solutions dépendent également des choix des sociétés exploitant le jeu, notamment en ce qui concerne le *grief-play* : un comportement peut être possible à adopter par le joueur sans pour autant être considéré comme autorisé, tout simplement parce qu'il est techniquement impossible à interdire sans nuire à l'intérêt du jeu. Des politiques de traçage des actions des joueurs

et des objets du jeu peuvent permettre de mettre en oeuvre des mesures coercitives (bannissement du joueur) afin de faire respecter les règles.

2.2 Architectures de distribution

Dans cette section, nous allons passer en revue les différents types de solutions proposées en terme d'architecture de distribution des jeux massivement multi-joueurs. Comme il est impossible d'être exhaustif en la matière, tellement ce domaine de recherche est actif et les solutions différentes selon les priorités des concepteurs, nous allons en donner une typologie, et fournir à chaque fois quelques exemples représentatifs des propositions d'architectures dans le milieu académique.

Les sociétés exploitant des jeux persistants massivement multi-joueurs ont tendance à préférer une architecture de type clients-serveur au sens large : les joueurs sont les clients du jeu, et un arbitre, contrôlé par la société exploitant le jeu, gère la persistance de l'application et garantit les changements d'états ayant lieu dans le monde virtuel.

À cause des effets que les tricheurs peuvent avoir sur le monde virtuel, il est nécessaire de ne faire confiance aux clients que dans la mesure où cela n'est pas trop risqué pour la pérennité du monde virtuel. Il est plus raisonnable de faire vérifier par un serveur contrôlé par les exploitants du jeu leurs actions les plus critiques ayant un impact sur l'état du jeu. Certains travaux académiques proposent tout de même des modèles sans arbitre central, car ce dernier peut être particulièrement coûteux à mettre en oeuvre pour les sociétés exploitant le jeu.

2.2.1 Architectures logiques clients-serveur

Les architectures clients-serveur ont actuellement la préférence des sociétés exploitant des jeux massivement multi-utilisateurs, malgré le coût additionnel éventuel en temps de transmission des messages d'un joueur à un autre : ceux-ci doivent transiter par le serveur logique avant d'être finalement acheminés vers la machine du joueur destinataire. Ces solutions ont également un coût financier non négligeable, en terme d'hébergement et de bande

passante pour l'exploitant du jeu.

En effet, ces solutions permettent de gérer plus facilement la synchronisation de l'application, et de mettre en œuvre des politiques de vérification de la légitimité des actions des joueurs par un arbitre en qui on peut faire confiance. En bref, elles permettent de rendre le jeu exploitable.

Cependant, une architecture logique clients-serveur simple, où une seule machine serveur est l'interlocuteur de tous les clients, devient vite un goulet d'étranglement et peut nuire au passage à l'échelle du jeu. Le problème consiste donc à organiser la partie serveur du jeu, sous contrôle et à qui on peut faire confiance, en un ensemble de machines physiques capables de supporter un grand nombre de clients.

On peut classer ces différentes familles d'architectures par rapport à la façon dont elles permettent d'interagir avec les clients, c'est-à-dire par rapport à la couche visible vis-à-vis des connexions des participants au jeu.

2.2.1.1 Architectures à base de services

Les jeux massivement multi-joueurs ont un grand nombre de fonctionnalités en commun, même si ils ne sont pas traités de la même manière selon les jeux. Il est possible d'extraire certaines de ces fonctionnalités pour les traiter de manière indépendante du reste du comportement de l'application : par exemple, le service qui permet aux joueurs de dialoguer entre eux, ou le service permettant de répercuter les déplacement de l'avatar d'un joueur aux joueurs possédant les avatars à proximité. Outre la possibilité de régler finement par service la qualité de service désirée, cette division permet de réutiliser facilement des services plus ou moins génériques. (figure 2.1). De plus, si les services sont suffisamment indépendants les uns des autres, un niveau minimal de synchronisation et de communications entre les serveurs logiques responsables sera nécessaire. Chaque service gère une portion de l'état du jeu, et seules les portions redondantes entre les services devront être synchronisées.

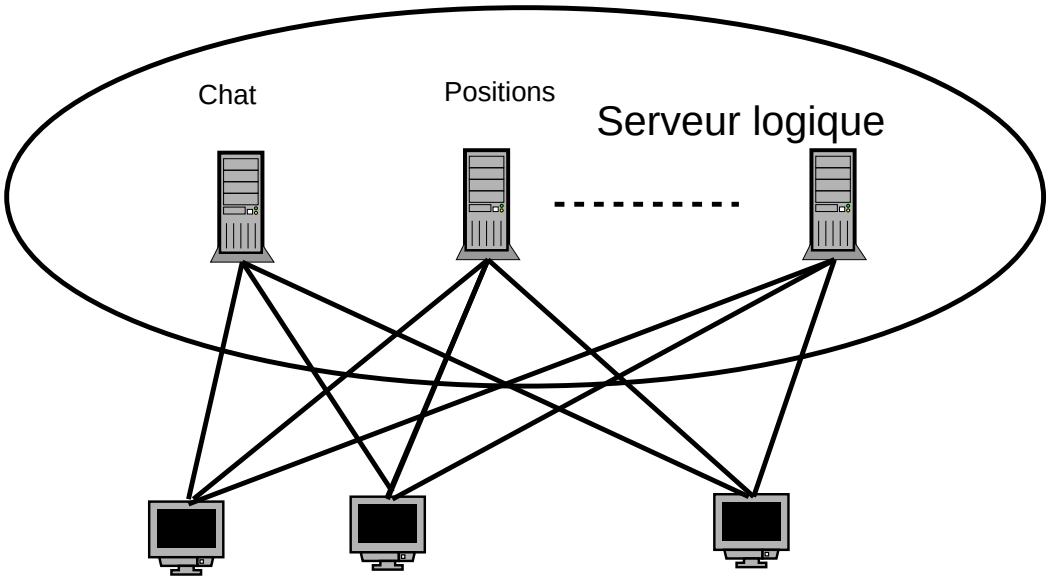

FIG. 2.1 – Serveur découpé en services

2.2.1.2 Architectures de type *proxies*

L'intérêt de cette solution est de réduire le temps de transmission des messages : elle se base sur l'attribution d'un interlocuteur pour chaque client (le *proxy*), par exemple selon la topologie physique du réseau. Son efficacité vis à vis de la rapidité de transmission des messages repose sur deux facteurs clés :

- le temps de transmission d'un message d'un client à son proxy est faible si ils sont proches physiquement ;
- le réseau entre les proxies dispose de très bonnes propriétés, en terme de latence et de bande passante. Cette propriété peut avoir un coût financier non négligeable pour le déploiement de l'application.

Les architectures de cette famille diffèrent principalement par les responsabilités attribuées à cette couche de *proxies* : les proxies peuvent être des serveurs de jeux complètement répliqués, contenant chacun la totalité de l'état du jeu, et synchronisés (figure 2.2). Cette solution peut permettre d'obtenir une meilleure rapidité de réponse à des requêtes des clients, mais peut coûter cher en bande passante à cause des communications entre les serveurs. Elle

peut également compliquer la synchronisation des différents serveurs logiques.

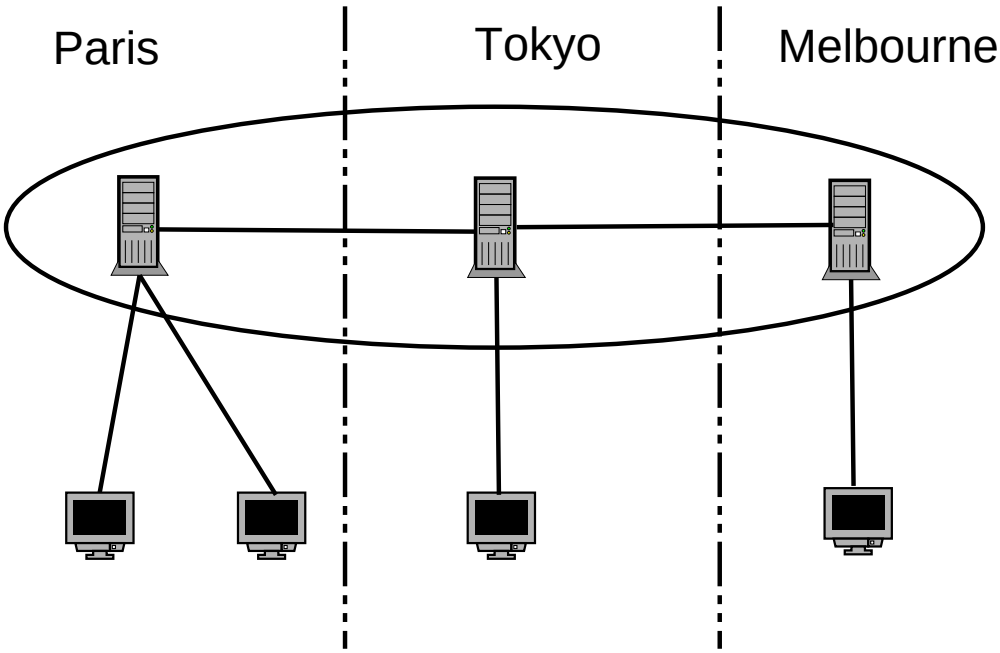

FIG. 2.2 – Serveur découpé en proxies répliqués

A l'autre extrême des possibilités, les proxies peuvent être juste une couche intermédiaire se chargeant d'acheminer les messages entre les clients à bon port, et les messages modifiant l'état du jeu vers une autre couche de serveur logique chargée de garantir la cohérence de l'application.

GISA [1] (Generic Internet Scalable Architecture) est une architecture à deux couches logiques, dont la première couche, dénommée *Concentration Tier* est en charge des connexions des clients, et des communications avec le reste de l'application. Une deuxième couche, dénommée *Server Tier* consiste en un ensemble de serveurs logiques synchronisés en charge de la maintenance de l'état du jeu. Les *Concentrators* ne font pas simplement du routage, ils permettent de filtrer certains événements venant des clients et sont munis d'un service d'adaptation dynamique à la bande passante de chacun d'entre eux.

Dans [68], les auteurs proposent une architecture de serveur logique hiérarchisée, où la gestion de certains éléments de l'état du jeu est déportée sur les proxies tandis que la totalité de l'état du jeu est maintenue sur la couche supérieure, de manière centralisée.

L'architecture en miroir proposée dans [27] se base sur une seule couche de proxies, auxquels les clients se connectent selon la topologie du réseau. Les proxies contiennent chacun une copie totale de l'état du jeu et sont synchronisés. La solution s'inspire, pour l'architecture serveur, d'une construction similaire à celle décrite dans la figure 2.2.

2.2.1.3 Architectures basées sur un découpage de zones virtuelles

Cette solution, qui a pour but de faciliter la synchronisation de l'état du jeu et d'économiser la bande passante entre les différents serveurs logiques est la plus en vogue dans l'industrie. Elle est basée sur une constatation simple : dans les mondes virtuels comme dans le monde réel, on communique et on interagit avec les gens qu'on perçoit. Elle consiste à diviser le monde virtuel en régions, chaque région étant gérée par un serveur logique différent. Chaque région dispose donc d'une portion indépendante de l'état du jeu à traiter, incluant tous les objets virtuels s'y trouvant (Figure 2.3. Chaque serveur de zone peut donc disposer d'une liste communes de services liés à la localisation dans le monde virtuel. Il peut gérer les déplacements et transactions s'y déroulant. Tout comme l'architecture basée sur un découpage en proxies, il peut être complètement découplé des autres serveurs et gérer lui même la persistance avec un support externe de base de données, ou dépendre d'un serveur central qui se charge de synchroniser certains états du jeu et ses aspects persistants. Cela implique qu'au cours de leurs déplacements dans le jeu les joueurs se déconnectent et se reconnectent de manière plus ou moins transparente à un autre serveur, lorsqu'ils quittent une région pour entrer dans une autre. Cette architecture est tellement populaire qu'elle a donné naissance à une tactique particulière de fuite, pour les joueurs dont l'avatar est en danger : le *zoning* consiste à quitter précipitamment une région du monde virtuel où son avatar est poursuivi par un *personnage non-joueur* menaçant. Ce *personnage non-joueur* n'étant en général pas conçu pour changer lui aussi de serveur, faisant partie de l'état du jeu géré par le serveur de la zone, l'avatar est sauf.

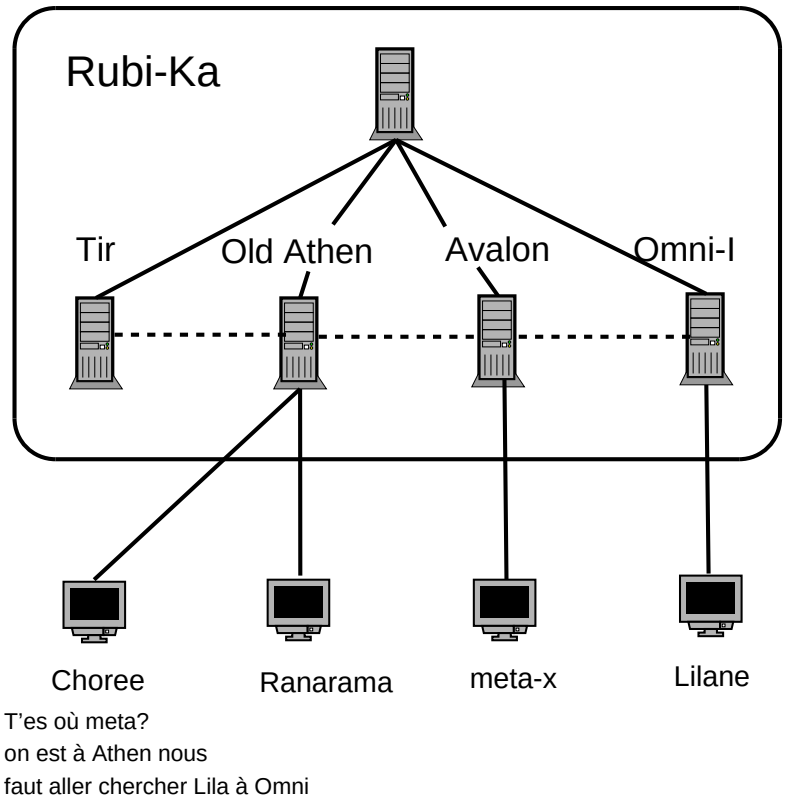

FIG. 2.3 – Architecture basée sur les zones géographiques du monde virtuel

Un autre avantage collatéral est qu'en cas de défaillance d'un serveur logique, seule une zone du monde est inaccessible aux joueurs.

Le système RING a été adapté pour une telle utilisation dans un souci de passage à l'échelle sur Internet, avec pour objectif de réduire les coûts en bande passante des communications entre les serveurs de zone auxquels les clients sont connectés [45].

2.2.2 Architectures *peer to peer*

Ce type d'architecture a pour but de réduire le coût de déploiement et d'exploitation des jeux massivement multi-joueurs pour les sociétés qui ne peuvent investir de grosses sommes dans des architectures clients-serveur.

Ces architectures sont composées uniquement des machines des joueurs, les pairs, qui représentent le monde virtuel. Tous ces pairs se communiquent entre eux les modifications de l'état du monde. (Figure 2.4). La plupart des

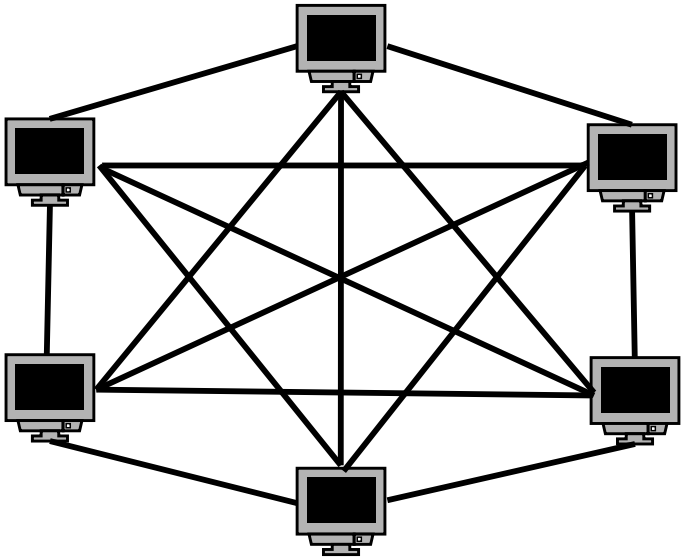

FIG. 2.4 – Architecture *peer to peer*

travaux en ce sens proviennent de la recherche induite par les mondes virtuels non commerciaux, parfois sur réseau propriétaire où les problèmes relatifs à la nature d'Internet n'interviennent pas, et des recherches dans le domaine du calcul distribué.

Une architecture purement *peer to peer* peut difficilement passer à l'échelle : si tous les clients doivent communiquer entre eux et diffuser leurs changements d'états, la consommation de bande passante de l'ensemble du réseau grandit de manière spectaculaire dès qu'un joueur se connecte à un jeu déjà peuplé. De même, si aucune précaution n'est prise à cet effet, il est particulièrement délicat de s'assurer que des incohérences ne vont pas s'introduire dans l'état du jeu, puisque rien ne garantit a priori que tous les pairs vont recevoir les mises à jours des différents clients dans le même ordre, et à partir du même état de référence.

MiMaze, développé à l'INRIA, est un exemple de jeu multi-joueurs sur

Internet qui repose sur une architecture de cette famille [31], mais qui utilise des techniques permettant de résoudre les problèmes de passage à l'échelle. Nous reviendrons plus tard sur cette application qui possède des propriétés intéressantes.

La plupart des propositions d'architectures totalement *peer to peer* pour des jeux massivement multi-joueurs ne donnent aucune solution aux problèmes de sécurité et de triche qu'elles posent par nature, laissant cet aspect pourtant crucial à des projets de recherche ultérieurs.

Dans [52], les auteurs proposent une architecture purement *peer to peer* dans l'objectif de construire un jeu massivement multi-joueurs passant à l'échelle de manière importante tout en gardant un bon niveau de réactivité, en utilisant pour résoudre ce problème une construction mathématique élaborée. Les résultats annoncés concernant le défi relevé sont très satisfaisants. Malheureusement, les auteurs préviennent dès le début de l'article : ils font confiance aux clients et les échanges de messages sont suffisants pour assurer la maintenance de l'état de l'application. En d'autres termes, les aspects de sécurité et de persistance ne sont absolument pas pris en compte.

Dans [55], l'architecture *peer to peer* proposée a également pour but de résoudre le problème du passage à l'échelle de l'application. A cet effet, chaque participant au monde a la responsabilité de la gestion de l'état du jeu concernant une zone géographique du monde virtuel. L'inconvénient majeur de cette solution est que si l'un des hôtes de l'application est sujet à une défaillance technique, c'est toute une zone du jeu qui disparaît. De plus cette solution, là encore, est plus adaptée au cas de joueurs se faisant confiance car chacun pourrait si l'envie lui prenait tricher sur la gestion de l'état du jeu dont il est responsable. Les auteurs pensent améliorer cet aspect dans de futurs travaux en utilisant des mécanismes de réputation mais il nous semble probable que même totalement aboutie, l'idée n'est pas assez sûre pour convenir à un jeu massivement multi-joueurs persistant car il resterait sensible à des attaques d'un grand nombre de joueurs coordonnés. Toutefois, la solution nous semble intéressante pour les jeux par session, par exemple, un jeu de stratégie temps réel massif en nombre de joueur mais se jouant par parties.

De plus, beaucoup de ces architectures inspirées par le domaine de la simulation ont été inventées pour être déployées en réseau local. Dans ce cas, on peut s'appuyer sur des protocoles de communications de groupe plus robustes que ceux proposés sur Internet et que nous étudierons plus loin, ce

qui n'est malheureusement pas le cas des jeux massivement multi-joueurs.

Les problèmes de cohérence de l'état du jeu ayant un impact moindre pour les jeux multi-joueurs non persistants, ces solutions peuvent être plus intéressantes dans le cas des jeux éphémères, se jouant par session sur Internet.

2.2.3 Synthèse

Dans la pratique et selon les performances recherchées, il est souvent adéquat de mélanger allègrement les différentes architectures que nous avons présentées, pour organiser l'application de manière à mieux répondre aux ressources jugées critiques : les architectures centralisées souffrent de problèmes de congestion, les solutions non centralisées posent des problèmes de consistance, de cohérence de l'état du jeu, et les solutions basées sur des *proxies* peuvent améliorer la latence mais sont plus coûteuses.

Par exemple, un premier découpage en services génériques (service de *chat* ou de communication audio) peut être suivi d'un découpage en *proxies* chargés du routage de l'information entre des serveurs de zone se partageant l'état du jeu.

Il est également possible de superposer deux architectures différentes, gérant chacune de manière différente l'état du jeu. Par exemple, on trouve dans [82] une architecture hybride, *peer to peer* avec serveur. Le rôle du serveur y est limité à un arbitrage du jeu. Chaque interaction d'un joueur avec le monde virtuel est envoyée à tous les autres clients, mais également au serveur qui joue un rôle d'arbitre : il calcule et simule l'état du jeu, et s'il détecte un problème de synchronisation, il intervient auprès des clients. Ce modèle garde les propriétés de rapidité de communication des architectures *peer to peer*, conserve les garanties de synchronisation des architectures clients-serveur, mais garde des problèmes de passage à l'échelle, même quand tout se passe bien et que le serveur n'a pas à intervenir pour re-synchroniser l'application.

En conclusion, il n'y a pas d'architecture idéale, toutes ont leurs avantages et inconvénients, améliorant une caractéristique jugée critique pour dégrader le traitement d'une autre, augmentant le coût financier du jeu en améliorant le passage à l'échelle, améliorant le temps de transmission des messages entre

joueurs en rendant la synchronisation plus problématique, et ainsi de suite.

Il ne peut donc pas exister d'architecture générique pour les jeux massivement multi-joueurs

2.3 Modèles de communication

2.3.1 Politiques globales de mise à jour de l'état du jeu

Différentes politiques de communication entre les hôtes de l'architecture distribuée d'un jeu massivement multi-joueurs peuvent être utilisées, selon les propriétés jugées les plus critiques. De ces politiques de mises à jour dépend la synchronisation de l'application, et le degré de cohérence global de l'application est fortement lié à ce choix. Voici une classification non exhaustive de ces différentes politiques de mise à jour.

2.3.1.1 Régénération fréquente de l'état du jeu

Le principe général de ce modèle bien décrit dans [93] est simple : chaque hôte de l'architecture de distribution responsable de la gestion d'une partie de l'état du jeu envoie régulièrement à tous les autres hôtes les valeurs à jour de ces états, de la manière la plus rapide (et donc non fiable) possible sans se poser la question de savoir si la valeur d'un état particulier a été modifié depuis la dernière communication, ou si un des destinataire a reçu la valeur précédemment communiquée (les communications ne sont pas forcément ordonnées, et il en est donc de même pour les mises à jour).

Ce choix sacrifie la synchronisation de l'application au profit de la rapidité de mise à jour, surtout dans des mauvaises conditions de réseau : la cohérence des états du jeu vus de chaque hôte de la distribution souffre avec la perte et le non ordonnancement des messages.

Ce modèle a beaucoup été utilisé à l'époque des premiers FPS , qui étaient principalement destinés à être joués sur réseau local en mode multi-joueurs : il est en effet assez simple de rajouter ce modèle à un jeu mono-joueur pour le transformer en un jeu multi-joueurs.

La consommation de bande passante que ce modèle implique a vite donné lieu à l'application des techniques de communications de groupe, que nous étudierons plus loin. Par exemple, l'adaptation de RING [45] qui utilise originellement ce modèle de communication avec des techniques d'optimisation adéquates le rend viable pour une application plus massive sur Internet.

2.3.1.2 Temporisation

Les messages transmis entre les hôtes d'un jeu massivement multi-joueurs peuvent parfois être très nombreux et de petite taille. Lorsque le besoin se fait sentir d'économiser la bande passante et que la rapidité de mise à jour n'est pas critique, il est alors possible de travailler sur la couche réseau de l'application pour rassembler les informations en un seul message [93]. Cette technique permet d'économiser sur la taille des en-têtes des messages qui peuvent représenter la plus grande partie du message transmis.

- On peut décider d'une taille optimale de quantité d'information à envoyer au destinataire, et attendre que ce quorum soit atteint avant de construire le message et de l'envoyer. Cependant, si le rythme des mises à jour est irrégulier, on risque d'introduire un délai de mise à jour gênant.
- On peut également poser un intervalle fixe auxquels les messages seront assemblés et envoyés. On maîtrise ainsi bien mieux le sacrifice de latence qui sera fait, mais si peu de messages sont produits pendant l'intervalle il y a peu de chance de faire des économies.

Il peut donc être intéressant, pour les applications se mettant à jour de manière très irrégulière, d'utiliser les deux bornes simultanément, et de concaténer les messages selon la première borne (temporelle ou quorum d'informations) atteinte. Ainsi, en fixant ces bornes de manière adéquate, on peut équilibrer selon les besoins spécifiques de l'application le compromis à trouver entre la bande passante et la rapidité des mises à jour.

2.3.2 Politiques de mise à jour état par état

Les deux premiers modèles que nous allons aborder rentrent dans la catégorie des modèles à objets distribués. Dans ces modèles, l'application est vue comme un ensemble d'objets indépendants qui peuvent être invoqués sur

chaque hôte par d'autres objets distants. C'est l'extension du paradigme de programmation objet aux applications distribuées.

2.3.2.1 Invocation de méthode distante

Dans cette approche, les objets peuvent invoquer des méthodes d'objets distants tout comme ils peuvent invoquer des méthodes d'objets locaux dans le paradigme de programmation objet classique. Les communications sont synchrones, c'est-à-dire que l'objet qui invoque attend la réponse de l'objet distant. Ce modèle a donné lieu à de nombreuses implémentations dans le domaine des applications distribuées généralistes. Par exemple, Java RMI [88] en donne une implémentation : les invocations se font selon la même syntaxe pour les objets locaux et distants. Le programmeur doit utiliser une interface particulière pour représenter les objets distants, ce qui lui permet de gérer les problèmes particuliers à ce type d'invocation au travers d'exceptions spéciales. Nous étudierons plus en détail plus tard un *framework* utilisant un modèle apparenté (voir 3.3, page 90).

2.3.2.2 Modèle diffusion - souscription

Ce modèle est guidé par les événements se produisant pour les objets. Un objet diffuseur permet à des objets distants souscripteurs de s'abonner, pour recevoir certains événements. Lorsque l'événement pour lequel un souscripteur s'est abonné se produit, il est notifié par le diffuseur.

Les notifications envoyées par les objets diffuseurs, producteurs d'événements, sont envoyées de manière asynchrone aux souscripteurs.

Ce modèle de communication comporte plusieurs implémentations classiques, non dédiées aux jeux massivement multi-joueurs. Par exemple, *Jini* [4] permet à un objet d'une machine virtuelle Java de souscrire à un objet résidant sur une autre machine virtuelle Java et d'en recevoir les événements.

En ce qui concerne les jeux massivement multi-joueurs, le système Mercury [12], basé sur ce modèle, a été construit pour répondre à des impératifs de passage à l'échelle. De même, les auteurs de [40] proposent d'utiliser ce modèle conjointement avec une division en zones du monde virtuel, et proposent une catégorisation statique des objets diffuseurs selon que les événe-

ments qu'ils produisent modifient le monde virtuel ou non.

2.3.2.3 Réplication d'états

Le principal inconvénient du modèle diffuseur-souscripteur est que rien n'empêche un client de jeu modifié de souscrire aux événements d'un objet, et qu'ils peuvent donc constituer un risque de triche en permettant à un joueur d'obtenir des informations lui donnant un avantage injuste.

Pour répondre à ce problème de sécurité, l'auteur de [48] propose un *framework* basé sur un modèle plus fin de réplication d'états, incluant également une notion de priorité, construit sur une architecture clients-serveur utilisant des proxies. Le modèle de communication est la encore guidé par les événements, et l'assignation des objets devant recevoir des notifications est faite côté serveur, de la manière décidée par le développeur.

2.3.3 Techniques de communications de groupe

Ces techniques ont pour but d'optimiser la consommation de bande passante, en réduisant le nombre de destinataires à qui les mises à jour d'états doivent être envoyées.

2.3.3.1 Le protocole multicast sur Internet

La première chose qui vient à l'esprit, quand on pense à réduire la bande passante utilisée pour envoyer le même message à un grand nombre de personnes est le protocole multicast, dont il existe une implémentation IP. Les récepteurs doivent être abonnés à un *groupe de multicast*, identifié par une adresse IP. L'expéditeur du message envoie la mise à jour à cette adresse IP, et c'est le protocole qui se charge du routage de l'information aux abonnés. L'utilisation de ce protocole a donné lieu à de nombreux travaux dans le domaine des mondes virtuels, surtout ceux étant construits pour fonctionner sur des réseaux propriétaires. Par exemple, [65] propose tout simplement de partitionner le monde en zones géographiques, et d'attribuer à chaque zone une adresse de multicast différente pour le routage des mises à jour de la portion de l'état du jeu concernant cette zone.

Malheureusement, l'implémentation d'IP-multicast pose quelques problèmes pour son utilisation dans le cadre d'un jeu massivement multi-joueurs. Tout d'abord, l'envoi d'un message à une adresse multicast n'est pas sécurisé, un intrus peut donc perturber le déroulement de l'application en envoyant des messages à cette adresse. De même, un joueur disposant d'un client modifié peut joindre un groupe de multicast et obtenir des informations qui peuvent lui permettre d'obtenir un avantage injuste. Les adresses multicast sont également peu nombreuses sur l'Internet actuel (IPv4), et les sessions doivent être annoncées (sur ce point précis, IPv6 fournira des améliorations [25]).

De plus IP-multicast repose sur des transmissions non fiables et non ordonnées. En d'autres termes, il n'y a aucune garantie que le message soit bien acheminé à tous les destinataires abonnés, ce qui peut être nécessaire pour certains messages critiques.

Enfin, les performances actuelles de l'implémentation sont insuffisantes : joindre un groupe de multicast sur Internet est une opération encore trop longue pour permettre de baser des modèles de communications nécessitant des changements fréquents d'abonnement à une adresse multicast. Les auteurs de [64] fournissent une étude très pertinente des limitations actuelles de l'implémentation du multicast sur Internet ainsi que des pistes pour son amélioration.

Toutefois, certains jeux ont relevé le défi. Mimaze [30], dont nous avons déjà parlé en évoquant les architectures *peer to peer* est un jeu entièrement basé sur le multicast, afin d'améliorer le passage à l'échelle du jeu. Les auteurs utilisent des techniques d'extrapolation pour remplacer les messages éventuellement perdus et une horloge partagée pour synchroniser l'application et améliorer la consistance. Cette solution fournit des qualités de synchronisation suffisantes pour le jeu Mimaze, mais pas pour un jeu de style FPS où les interactions entre les joueurs sont plus nombreuses [27].

2.3.3.2 Le multicast au niveau applicatif

Il existe de nombreuses propositions de multicast au niveau applicatif pour les applications distribuées, proposant des propriétés qui manquent à IP-multicast. Dans ces techniques, le routage des informations est calculé sur les hôtes de l'application, et des propriétés d'ordonnancement ou de réachemi-

nement des paquets perdus, qui manquent au protocole IP-Multicast, peuvent alors être utilisés. Les chercheurs du domaine font la distinction entre les solutions purement applicatives, où le routage est effectué sur chaque hôte de l'application, et les solutions reposant sur une architecture à base de *proxies* chargés de l'implémentation du protocole de distribution. On peut trouver dans [62] un comparatif de quelques unes de ces techniques. Certaines de ces solutions ont été utilisées dans le domaine des mondes virtuels. Par exemple, DIVE [44], dispose d'un module permettant d'utiliser un protocole multicast fiable, au niveau applicatif [2]. Yoann Fabre propose également dans [36] d'utiliser la bibliothèque *Extended Virtual Synchrony* [71] pour la conception d'une solution générique pour créer des applications de type «monde virtuel». Cependant, la plupart des solutions proposées dans le cadre de ces applications spécifiques sont généralement très spécialisées pour le domaine d'application. Ce sont les techniques de gestion des intérêts dont nous allons parler dans ce qui suit.

2.3.3.3 Les techniques de gestion des intérêts au niveau applicatif

Ces techniques consistent à regrouper les différents hôtes de l'application selon leur intérêt pour certaines mises à jour ou événements produits sur d'autres hôtes de l'application [70, 59, 37].

Le concept repose sur les notions de *focus* et de *nimbus* des objets du monde virtuel. Nous illustrerons ces concepts en considérant comme objet l'avatar d'un joueur :

Le focus d'une entité du monde virtuel représente tous les objets sur lesquelles l'avatar doit recevoir des informations. Transposé dans le cadre de la géographie d'un monde virtuel, ce sont les objets de son «champ de vision». Le nimbus représente toutes les objets à qui il doit envoyer les informations sur son changement d'état. Pour la même analogie spatiale, c'est la zone d'où il est «visible» par un autre objet. Dans l'idéal, on traite chaque information séparément, et on l'envoie uniquement aux entités dont le nimbus a une intersection avec le focus de l'émetteur. L'inconvénient de ce modèle est que des calculs potentiellement importants selon le nombre de destinataires sont faits à chaque fois que l'état du jeu est modifié. Sa complexité dépend de sa

[2] Dive : manuel utilisateur : `http ://www.sics.se/dive/manual/sid2.html`

précision : si on découpe le monde en un maillage de carrés, il sera moins complexe mais moins précis que si on utilise des figures polygonales se rapprochant plus de l'idéal du cercle. C'est acceptable seulement quand il n'est pas indispensable de supporter un passage à l'échelle en nombre de clients, et quand le monde n'est pas trop riche et dynamique.

Dans la pratique, et pour permettre un bon passage à l'échelle de ces techniques coûteuses en temps de calcul, ces services sont souvent installés côté serveur sur la couche de l'architecture en communication avec les clients. Par exemple, dans GISA [1] un filtrage des intérêts des clients, est utilisé sur les *proxies* d'une architecture serveur centralisée.

Ces services peuvent également être adaptés dynamiquement aux capacités de traitement de l'application et permettre un équilibrage dynamique de la charge. Par exemple, si un client dispose d'une connexion de moindre qualité, il est possible de réduire ses centres d'intérêts en définissant un mode dégradé basé sur des priorités qui lui permet de recevoir les communications les plus importantes vis-à-vis du *game-play*. Cette philosophie s'inspire des techniques utilisées pour le rendu graphique des jeux, qui doivent pouvoir s'adapter aux capacités des différents matériels utilisés par les joueurs, en diminuant la qualité du traitement de l'image pour gagner en rapidité. Cette possibilité est évoquée dans [48]. De même, dans le cadre d'une architecture de serveur logique basée sur les zones géographiques du monde virtuel, si le serveur souffre d'un soudain surpeuplement, une réadaptation dynamique du filtrage des intérêts des clients basée sur une telle définition des priorités peu permettre à l'architecture de continuer à fonctionner de manière satisfaisante en sacrifiant un peu de temps de calcul à une meilleure utilisation de la bande passante disponible.

2.4 Techniques de masquage de la latence

Les techniques que nous allons aborder dans cette section ont deux principaux contextes d'application.

Les jeux doivent présenter un état du jeu le plus cohérent possible à chaque joueur pour être *jouables*. Or, les temps de transmission des messages sur Internet peuvent souvent dépasser la limite acceptable pour le joueur.

La désynchronisation temporaire de sa vision de jeu par rapport à l'état courant de l'application peut l'empêcher d'adapter ses actions à celles de ses partenaires qu'il reçoit trop tardivement. Cette désynchronisation peut également le frustrer quand ses propres actions sont faussées car ce qu'il perçoit ne correspond pas à l'état jugé valide par l'arbitre de l'état global du jeu. Il devient donc nécessaire d'adopter des stratégies de masquage de ce problème de cohérence, dû à l'inévitable latence réseau.

De plus, les protocoles de communications rapides qu'on utilise dans le cadre de ce type de jeu peuvent perdre des informations sur Internet. Ces inconvénients sont d'importance imprévisible, et inhérents à la nature même d'Internet, on ne peut que faire avec. Les techniques de masquage de la latence ont pour but non pas de résoudre ces problèmes, mais de les masquer à l'utilisateur en essayant de lui fournir l'illusion que toutes les informations sont bien arrivées.

Dans [10], un article décrivant les solutions mises en oeuvre pour la réalisation du FPS à succès *Half-Life* [51], l'auteur décrit les trois principales techniques utilisées pour masquer la latence : l'introduction d'un délai artificiel, la compensation côté serveur, et le *Dead-reckoning* (voir glossaire). Nous allons passer en revue ces différentes méthodes et discuter d'autres travaux à leur sujet.

2.4.1 Introduction de délai

Ces techniques consistent à synchroniser «de force» l'application sur chaque client, en introduisant un délai suffisant pour que tous les messages nécessaires à produire l'état du jeu arrivent à destination avant de fournir une représentation de cet état au client. Le jeu tel qu'il est perçu du point de vue du joueur a donc toujours un temps de retard par rapport aux dernières modifications de l'état du jeu, qu'elles soient déjà disponibles sur la partie cliente de l'application ou non. Son principal avantage est que les joueurs ont plus de chances d'avoir la même vue sur l'application, pourvu que le délai introduit soit supérieur ou égal à la latence du système. Son principal inconvénient est qu'un client malintentionné peut modifier son programme afin d'exploiter les informations présentes mais qu'il n'est pas censé percevoir à son avantage dans le jeu. De plus, le choix de la durée du délai est crucial : plus il est grand, plus on s'approchera d'une vision idéale de l'appli-

cation, où tous les joueurs percevront le même monde, mais plus le décalage entre ses actions sur le monde et leurs conséquences sur l'état partagé par tous sera perceptible, et potentiellement gênant pour un *game-play* rapide et réactif. L'équilibrage de ce compromis est intrinsèquement lié aux choix de *game-design*, et donc propre à chaque jeu. Les auteurs de [80], déjà cité, fournissent une étude des délais produisant des *game-play* jouables pour des types de jeux classiques (courses de voiture, FPS).

2.4.2 Compensation côté serveur

Bien nommées, ces techniques visent à corriger les conséquences de la latence pour les joueurs dans le cadre d'une architecture logique clients-serveur.

La technique classique décrite dans [10] consiste à évaluer l'effet des messages transmis par chaque client dans le contexte de l'état du jeu dans lequel il s'est produit, en retrouvant cet état passé à l'aide d'une estimation de la latence de ce client. Coûteuse en calcul côté serveur, cette solution peut néanmoins être profitable et réduire le sentiment de frustration des joueurs.

Dans [49], les auteurs proposent un service d'échange de message équitable qui peut être placé sur la couche visible des clients dans le cadre d'une architecture logique clients-serveur. Le but est d'ordonner les messages provenant des différents clients afin que ce ne soit pas toujours celui qui a la plus faible latence qui emporte la mise, sans pour autant avoir besoin d'utiliser une horloge partagée. À cet effet, lorsqu'une mise à jour d'état est envoyée par le serveur, la réponse des clients est prise en compte avec un temps de réaction qui est utilisé pour coordonner les réponses des différents clients.

2.4.3 Dead-reckoning

Cette technique, particulièrement bien décrite dans [59] et [93], peut être utilisée dans le cadre d'applications construites sur une architecture *peer to peer* [31] comme clients-serveur. Elle a été très largement utilisée, dans le cadre des jeux de style FPS industriels comme dans l'académique, pour permettre un rendu fluide des déplacements des avatars des joueurs distants. Elle consiste à extrapoler la nouvelle valeur d'un état du jeu, dont la mise à jour tarde à arriver ou s'est perdue dans le réseau, à partir de ses valeurs

précédentes. Bien sûr, la valeur extrapolée peut se révéler finalement incorrecte auquel cas il faudra la corriger en évitant de faire sauter brutalement l'objet mal déplacé de sa position extrapolée à la nouvelle. Il faut donc mettre au point deux algorithmes complémentaires, l'un pour la prédiction, l'autre pour la correction éventuelle de cette prédiction.

Par exemple, dans [81], les auteurs comparent les performances de deux algorithmes de prédiction dans différents styles de jeu (FPS, courses de voitures). Le premier, très classique, est basé sur la dérivation de polynômes représentant les caractéristiques des déplacements des avatars des joueurs pour obtenir leurs vitesse, accélération, et courbe instantanées afin d'en déduire leur position suivante. Le deuxième peut paraître plus surprenant, et se base sur les états passés des périphériques de contrôle des joueurs (souris, clavier...) pour prédire ses prochaines actions (et en déduire par exemple également la prochaine position de son avatar, mais cela peut aussi être d'autres actions concernant les objets qu'il contrôle).

La méthode classique des polynômes dérivés étant basée sur des données instantanées et se révélant donc souvent insatisfaisante, l'auteur de [94] propose une méthode hybride, permettant un choix dynamique de l'ordre du polynôme, utilisant un polynôme du 1er ordre quand la vitesse est petite ou nulle, ou que l'accélération change fréquemment. Sinon, on s'autorise un polynôme d'ordre supérieur qui donnera des résultats plus précis (au prix de plus de consommation de temps de calcul). Ce protocole permet également de baser l'information non sur des données instantanées, mais sur des moyennes sur une période de temps, ce qui contribue à garantir des résultats plus stables.

Plus récemment,les auteurs de [2] proposent d'améliorer la pertinence des résultats obtenus par l'utilisation d'une horloge partagée.

Les algorithmes de correction ont eux à résoudre un compromis fluidité et naturel contre rapidité de la correction. À un extrême de l'étendue des possibilités, si la re-synchronisation de l'application doit être faite le plus vite possible, il est toujours possible de la corriger brutalement. À l'autre, calculer une courbe permettant de déplacer moins rapidement mais plus naturellement l'objet à sa nouvelle position permet de maintenir le joueur dans l'illusion d'une bonne synchronisation au prix justement de la synchronisation et d'un peu de temps de calcul.

Les principaux inconvénients du Dead-reckoning sont décrits dans [6]. Tout d'abord, par nature, ces techniques introduisent des inconsistances dans le jeu tel que perçu par les joueurs par rapport à l'état de référence de l'application, ce qui peut mener à une certaine frustration et à des décisions considérées comme injustes du *game-play* (le joueur croyait viser juste, mais en réalité il avait tiré à côté de la cible dont la position était mal extrapolée). De plus, un joueur malintentionné peut modifier son client afin d'exploiter l'algorithme à son avantage : il peut ralentir temporairement la diffusion de ses déplacements, simulant ainsi un problème réseau. Cela provoquera chez les autres joueurs l'utilisation de l'algorithme d'extrapolation, tandis que le tricheur prendra lui bien garde à ne pas faire correspondre ses déplacements à la prédiction résultant de l'algorithme. Ainsi, les autres joueurs auront une mauvaise représentation de sa position et bien des difficultés à le viser.

2.5 Conclusion

Grâce à ce catalogue des techniques utilisées pour la mise au point des jeux massivement multi-joueurs, qui met en balance les bénéfices apportés par chaque solution par rapport à ses inconvénients, nous pensons avoir démontré dans ce chapitre le pivot de notre réflexion : les jeux massivement multi-joueurs sont des applications complexes à mettre au point, et chaque choix technique visant à l'amélioration d'une caractéristique jugée critique pour la jouabilité se paye par une moins bonne résolution d'une autre caractéristique.

Réaliser un jeu massivement multi-joueurs est une affaire de compromis, dont la solution est intrinsèquement liée au type de jeu que l'on veut réaliser.