Game-design et conception technique

4.1 Introduction

Ce chapitre a pour but de faire une synthèse des problèmes que nous cherchons à résoudre, en regard de l'état de l'art présenté lors des précédentes parties de ce manuscrit. Nous y décrivons la réflexion qui nous a amené à élaborer les solutions proposées dans le cadre de cette thèse. Nous expliquons pourquoi les outils et *frameworks* actuels ne suffisent pas, et comment nous proposons de réaliser un environnement de développement dédié à la réalisation de jeux massivement multi-joueurs.

Nous défendons l'idée de la nécessité de fournir aux développeurs un *framework* muni d'une sémantique simple et bien définie, permettant ainsi l'adjonction ultérieure d'outils de mise au point, et nous expliquons pourquoi les méthodologies basées sur le raffinement de prototypes successifs sont les seules solutions viables pour mettre au point une application si complexe.

Nous décrivons ensuite les grandes idées sur lesquelles repose le modèle défini au cours de cette thèse, le comparons avec des approches voisines, et nous illustrons son utilisation par des exemples simples.

4.2 Le problème des interactions dans un monde virtuel

4.2.1 L'importance de la qualité des interactions dans le *game-design*

Le cœur d'un jeu en ligne ou d'un monde virtuel, c'est l'interaction. On joue pour interagir avec d'autres joueurs, dialoguer, collaborer ou se battre, commercer et échanger des objets du monde virtuel. Le fait de déplacer son avatar dans un monde virtuel est une interaction du joueur qui modifie l'état du monde virtuel en modifiant sa position. Le fait d'acheter un objet virtuel dans un magasin du monde virtuel en est une autre. Lorsque des joueurs se battent, ils interagissent les uns avec les autres tout en modifiant également l'état du monde...

Ce sont les interactions que le joueur a avec les autres joueurs et avec le monde persistant lui-même qui rend l'expérience si différente des jeux traditionnels.

Cependant, pour que le sentiment d'immersion dans le monde virtuel soit réussi, ces interactions doivent sembler les plus naturelles possible. Pour une bonne expérience de jeu, leurs propriétés techniques doivent être adaptées au *game-play*. Par exemple, nous avons comparé au chapitre 1 les manières dont se déroulent les combats entre joueurs, dans un FPS comme Quake[84] ou Half-Life[51], et dans un jeu massivement multi-joueurs comme Everquest : un FPS se joue beaucoup sur la qualité des réflexes des différents participants et requiert une propagation très rapide du moindre mouvement d'un joueur aux ordinateurs des autres protagonistes, tandis que dans Everquest, c'est le nombre d'informations à propager aux participants qui devient très vite important et la précision et la rapidité de transmission des positions ne sont plus aussi cruciales.

Pour le développeur, définir une interaction dans un jeu massivement multi-joueurs, c'est définir de quelle manière un joueur, ou un événement contrôlé par le serveur, modifie l'état global du monde virtuel. Nous avons vu dans le chapitre 2 un échantillon représentatif des techniques actuellement utilisées pour développer un jeu en ligne compte tenu des impératifs de jouabilité et d'immersion du joueur dans la monde virtuel. Nous allons main-

tenant expliquer pourquoi, malgré le fait que ces techniques soient connues, il est très délicat de faire les choix qui permettront de réaliser un jeu satisfaisant les qualités des interactions requises par le *game-play*.

4.2.2 La délicate mise au point d'un jeu en ligne

Nous allons dans cette partie faire une synthèse des difficultés rencontrées lors de la réalisation d'un jeu massivement multi-joueurs, étudiées lors des chapitres précédents. Ce retour sur les points clés du développement d'une telle application a pour but d'étayer notre argumentation.

4.2.2.1 Les limites physiques des interactions

Un jeu massivement multi-joueurs est une application distribuée sur Internet qui doit faire partager une même vision du monde à tous ses participants. Comme une consistance absolue de l'état sur tous les participants au monde virtuel n'est pas possible compte tenu de la nature de l'Internet, le problème consiste à déterminer pour chaque état local à quel point la synchronisation avec sa valeur dans l'état global de l'application est nécessaire. La gestion de l'état global est le principal problème de mise au point pour les jeux massivement multi-joueurs.

Par exemple, un calcul assez simple montre que dans un réseau parfait, où les paquets d'information circulent à la vitesse de la lumière, un aller-retour entre Paris et Melbourne dépasse le temps considéré comme étant celui du réflexe humain (100ms). Il y a donc une limite physique à la rapidité de la synchronisation. Nous avons étudié des techniques permettant de pallier ce problème. Mais elles sont difficiles à mettre en œuvre car chacune présente des inconvénients par rapport à une autre caractéristique de l'application, et sont donc à utiliser au cas par cas, selon le *game-play* désiré.

4.2.2.2 La difficulté de tester les interactions en conditions réelles :

De plus, la solution au compromis précédemment cité est difficile à mettre au point car la qualité de l'interaction réalisée ne sera vraiment testée en conditions réelles que dans les dernières étapes du développement. Dans le

contexte d'un jeu en ligne, certaines faiblesses de conception n'apparaissent que lorsque le jeu est déployé dans des conditions réelles d'exécution, qui doivent tenir compte de la réalité d'Internet avec sa latence imprévisible, et de la participation d'un grand nombre de joueurs connectés afin de vérifier le passage à l'échelle. Ces conditions s'obtiennent généralement seulement lors du *beta-test*, qui est la toute dernière étape avant le lancement du jeu. Anarchy Online, un des premiers jeux massivement multi-joueurs a succès a bien failli ne pas se remettre de problèmes découverts seulement après la mise en ligne de l'application [47]. Toléré par les joueurs à l'époque ou les jeux en ligne étaient encore un type d'application novateur et la communauté confidentielle, ce type de lancement n'est plus aujourd'hui considéré comme acceptable, le grand public étant désormais plus exigeant. Un jeu dont le *beta-test* est désastreux se fera non seulement très vite une mauvaise réputation, mais devra provoquer un remaniement de toute l'application, et donc probablement des délais de lancement retardés et des coûts supplémentaires potentiellement élevés.

4.2.2.3 Le besoin de spécialiser finement chaque interaction :

Pour complexifier encore le problème, nous avons vu que la qualité des interactions des joueurs avec le monde virtuel est fortement liée à un style de *game-play* : selon le type de jeu, chaque modification de l'état du jeu doit donc être répercutée de manière différente, en faisant les compromis adéquats compte tenu des impératifs de fiabilité, de sécurité ou de rapidité de chaque propagation d'événement.

De plus, dans un même jeu, les différents états composant l'état global du jeu peuvent requérir différentes qualités d'interactions et donc différents compromis. La description des interactions doit donc également s'effectuer au cas par cas au sein d'un même jeu.

Mettre au point les interactions d'un *game-play* se résume donc à résoudre un compromis entre les qualités requises pour chaque caractéristique de l'application.

4.2.3 Les jeux massivement multi-joueurs actuels

Aujourd'hui, les types d'interactions et de *game-play* réalisables sont bien connus des concepteurs de jeu, et le lancement d'un jeu massivement multi-joueurs est moins dangereux, surtout si on se contente d'utiliser les recettes déjà éprouvées. Cependant, cela mène à une ressemblance frappante entre les jeux à succès, ennuyeuse pour les joueurs. Cette ressemblance est particulièrement flagrante du point de vue des interactions possibles entre les joueurs et le monde virtuel. Les jeux à succès les plus récents, comme *Starwars Galaxies* [95] ou *World of Warcraft* [53], ne proposent aucune innovation dans la manière dont les joueurs interagissent par rapport à *Everquest*, le pionnier du genre.

Neocron [76] est un exemple de jeu massivement multi-joueurs ayant toutefois essayé d'innover en terme d'interactions. Son lancement a d'ailleurs plus ressemblé à celui des premiers jeux massivement multi-joueurs, avec de nombreux tâtonnements dans les premiers temps, et son arrivée à maturation s'est faite seulement après des mois de jeu, certains problèmes n'ayant même jamais été réglés. Ce jeu a désormais été interrompu, les développeurs repartant de presque zéro pour en développer une seconde version sur des bases saines.

On peut donc penser que les concepteurs de jeu ne manquent pas d'envie d'innover et de nous proposer de nouveaux types d'interactions dans les jeux en ligne massivement multi-joueurs. Cependant, l'importance des budgets nécessaires et la difficulté de mise au point de nouveaux styles d'interactions freinent actuellement l'innovation.

4.2.4 Les outils dans l'industrie :

Depuis l'explosion du marché des jeux en ligne, de nombreuses solutions ont vu le jour dans l'industrie, pour aider au développement des jeux en ligne. On peut trouver une étude de ces solutions dans le rapport annuel 2003 de l'*International Game Developers Association*(IGDA) [54] sur les jeux en ligne. Sans donner une liste complète et une critique de ces solutions commerciales, on peut néanmoins les ranger en trois catégories principales selon ce qu'elles permettent de réaliser :

– les moteurs réseau et moteurs de jeux génériques, conçus pour le déve-

loppement d'un jeu particulier et éventuellement raffinés pour devenir plus génériques ;
- les solutions plus ou moins complètes de sociétés qui vendent juste des technologies, comme des moteurs d'Intelligence Artificielle divers et variés, permettant la génération de scénarios interactifs, la modélisation de comportements intelligents pour des agents ou des moteurs physiques ;
- des moteurs pour la réalisation de jeux classiques, adaptés après-coup pour satisfaire les besoins d'un passage à des versions multi-joueurs.

À ce jour, pratiquement tous les projets de développement de jeux massivement multi-joueurs ont commencé par une phase de développement d'outils et de bibliothèques adaptés à chaque jeu. Les composants logiciels développés lors de ces phases ont très rarement été utilisés pour des produits commerciaux différents de celui pour lequel ils ont été réalisés : conçus avec un projet de jeu particulier à l'esprit, ils manquent de généricité, et sont souvent d'un niveau d'abstraction fonctionnelle trop élevé pour permettre un réglage fin des interactions pour modifier celles qui sont prévues pour le jeu initial.

Les recettes techniques et les modèles de communication évoluent constamment, sont comme nous l'avons vu fortement liées au choix de *game-design*, mais sont généralement, malgré tout, parties intégrantes de ces solutions et donc difficilement modifiables ou paramétrables. Ces solutions peuvent être utilisées avec succès pour faire des jeux dont les principes et interactions ne diffèrent pas de trop de ceux pour lesquels elles ont été créées. Un jeu innovant du point de vue des interactions (et donc du *game-play* qu'il propose) ne peut donc pas tirer profit des outils existants.

Un grand nombre des sociétés s'intéressant à ce marché proposent plutôt l'infrastructure nécessaire à l'exploitation des jeux en ligne (par exemple, login sécurisé, services de mise à jour du jeu, services de discussion en ligne par mode texte entre les joueurs). Cette famille de solutions ne fournit aucune solution en terme de modélisation des interactions. Enfin, d'autres sociétés essaient de fournir des architectures de communication distribuées, un découpage en différents services des serveurs gérant le monde virtuel. Mais la plupart n'ont pas fait la preuve de leur généricité et de leur adaptation dans le cadre du développement d'un jeu massivement multi-joueurs [54]. De plus, nous avons vu que chaque architecture a un impact sur la qualité des caractéristiques du jeu, et nous doutons donc qu'une solution générique puisse être trouvée par ce biais.

L'outil idéal pour permettre de modéliser de nouveaux types d'interaction pourrait être quelque chose de similaire dans ses principes de base aux solutions industrielles qui existent pour le développement de jeux-vidéo traditionnels. Par exemple, Criterion Software propose Renderware [86], un ensemble d'outils et de bibliothèques couvrant le savoir-faire actuel dans ce domaine. Cette solution est ouverte, ce qui signifie que de nouvelles bibliothèques peuvent être ajoutées par les partenaires de l'industriel. Virtools dev [99] en est un autre exemple, et propose un kit de développement comportemental qui permet d'utiliser un certain nombre de briques de base à assembler à l'aide d'un langage à syntaxe graphique, ainsi qu'un langage de script et les moyens d'intégrer de nouvelles briques à la solution. Cette solution fournit également des facilités utilisables par un concepteur de jeu pour tester le logiciel ainsi produit.

Une approche aussi ouverte, incluant outil d'intégration et outils de tests, et ne faisant aucune hypothèse sur le modèle de communication et les modèles de données serait intéressante à utiliser pour le développement de jeux massivement multi-joueurs.

Il manque clairement quelque chose dans les outils actuels pour les rendre utiles à la conception de jeux innovants du point de vue des interactions.

4.2.5 Insuffisance des méthodes de développement actuelles

Le processus historique utilisé dans l'industrie pour le développement de jeux-vidéo ne convient plus pour la conception d'un jeu massivement multijoueurs. En effet, l'époque où un bon jeu vidéo se faisait à trois dans un garage, avec un développeur, un graphiste et un *game-designer*, est définitivement révolue avec ces projets impliquant parfois une centaine de personnes et prenant plusieurs années de développement.

Les politiques de gestion de projet sont donc en train de changer. Alors qu'historiquement le développement de jeux-vidéo est longtemps resté un secteur artisanal comparé aux autres développements logiciels, cette industrie se trouve désormais confrontée à la réalisation d'applications parmi les plus contraignantes. Nous avons vu dans le chapitre 3 comment l'industrie du jeu-vidéo tente de s'adapter, et nous avons étudié et critiqué les préconisations

des experts pour la réalisation d'un jeu massivement multi-joueurs.

Nous pensons donc que les méthodes utilisées actuellement dans l'industrie ne conviennent pas, et que les méthodes agiles comme *eXtreme Programming* [35, 7] présentées dans la partie précédente sont une alternative idéale pour la réalisation d'une application aussi complexe.

4.3 Notre proposition : un outil et une méthode

Le prototypage est un bon moyen de démontrer la faisabilité d'une nouvelle technologie dans l'industrie informatique en général, et nous avons vu dans le chapitre 3 comment les méthodes basées sur la réalisation de prototypes successifs permettent de sécuriser le déroulement d'un projet.

Les avantages du prototypage sont aussi valables pour l'industrie du jeu vidéo [73]. Dans le cadre du développement d'un jeu massivement multi-joueurs, la conception des interactions est critique, et directement liée au chiffrage financier des solutions de déploiement et d'hébergement effectué au démarrage du projet, surtout en ce qui concerne la somme des ressources à déployer (nombre de machines serveur, bande-passante). Un prototype a le bénéfice supplémentaire d'encourager l'innovation en fournissant les moyens de détecter les interactions critiques et d'expérimenter différentes recettes techniques avant l'étape du beta test.

Cependant, s'il est développé à partir de rien, un prototype demande en effet, comme le soulèvent les détracteurs de cette solution, un investissement considérable en temps de développement, même s'il n'est pas entièrement fonctionnel.

Il y a deux manières de résoudre ce dilemne :

- utiliser un cycle de vie du développement basé sur le raffinement de prototype, comme les méthodes agiles que nous avons décrites dans le chapitre précédent qui commencent à devenir assez populaires pour que certaines sociétés de jeux-vidéo commencent à s'y intéresser [29] au moins dans le cadre des jeux traditionnels ;
- disposer d'un outil de prototypage qui facilite l'établissement de la correspondance entre les interactions voulues fonctionnellement et les solutions techniques mises en œuvre, et qui soit capable de simuler des

conditions de déploiement réalistes pour valider cette correspondance.

Dans le premier cas, le code du prototype ne sera pas perdu, car raffiné, et réutilisé tout le long du cycle de développement. Les méthodes agiles ont d'autres avantages les rendant particulièrement bien adaptées au domaine du jeu-vidéo, par rapport aux autres méthodes basées sur le raffinement de prototypes successifs :

- elles sont adaptables à des projets de grande taille ;
- elles sont centrées sur les qualités des développeurs, généralement très compétents dans cette industrie attractive pour un professionnel en informatique ;
- elles permettent d'obtenir un logiciel plus facilement maintenable, quand la durée de vie d'un jeu massivement multi-joueurs se compte en années.
- leur principal inconvénient, qui est l'implication totale du client dans le projet, est réglé par le fait que le client sera ici le ou les *game-designers* du jeu massivement multi-joueurs.

Dans le deuxième cas, même si le prototype est jeté, on limite notablement l'investissement en temps nécessaire à sa réalisation.

Nous pensons que ces deux solutions doivent être rassemblées en une seule. En suivant l'exemple des outils qui existent pour réaliser des jeux-vidéo classiques, nous pensons qu'il est possible de proposer un modèle de développement pour les serveurs et les clients d'un jeu massivement multi-joueurs qui ne souffre pas du manque de généricité des solutions existant actuellement. Une approche basée sur des modules élémentaires, couplée avec des outils d'analyse et de prototypage, permet de détecter les fonctionnalités critiques. La modularité d'une telle approche aide à améliorer le prototype en prenant en considération les résultats d'analyses effectuées tout le long du cycle de développement, bien avant le début de la phase d'*alpha-test*, réduisant ainsi le risque de découvrir des défauts majeurs de conception trop tard pour le succès du projet.

4.4 Méthodologie de spécification du *framework*

Le travail réalisé dans le cadre de cette thèse consiste en la conception d'un *framework* (voir glossaire) sur lequel pourra s'appuyer l'outil évoqué dans la section précédente.

Nous allons exposer dans cette partie le cheminement qui nous a amené à la conception de ce *framework*, qui a été décrit tout d'abord dans [15], puis, de manière plus approfondie et détaillée dans [16].

Ce *framework* a donc pour but de servir de base à un environnement de développement pour encadrer la réalisation de jeux massivement multi-joueurs. L'outil final devra permettre de prototyper et de tester rapidement l'adéquation des interactions fonctionnelles exigées par le *game-play* et les solutions techniques utilisées pour ce faire, et d'accompagner le reste du développement dans le cadre de l'utilisation d'un processus agile. Cela implique certaines propriétés que le *framework* devra vérifier.

4.4.1 Philosophie générale

Il y a deux principaux écueils à éviter lors de la conception de ce *framework*.

Tout d'abord, la plupart des *frameworks* existants sont de trop haut niveau et manquent de souplesse en ne permettant pas d'adapter leurs composants. Comme nous l'avons vu précédemment, il n'est pas possible de concevoir un *game-play* novateur lorsque les solutions techniques sont figées. Il est donc probable que la solution que nous cherchons soit plus bas-niveau que celles qui existent actuellement, et nous nous interdirons de fixer les choix techniques.

Cependant, une solution trop bas-niveau risque de ne rien résoudre du tout en restant au niveau d'un langage évolué. Plus la solution sera bas niveau, plus le temps de développement en utilisant le *framework* se rapprochera d'un temps de développement en partant de rien.

Le compromis que nous avons adopté consiste à donner à l'utilisateur

une vision bas niveau du *framework*, pour lui laisser le choix de la solution à chaque problème donné, mais de faciliter au maximum la mise au point de cette solution. Le *framework* ne doit pas résoudre les problèmes, mais simplifier leur expression et leur traitement.

Pourquoi, alors, ne pas avoir choisi de définir plutôt un langage dédié ? En fait, le niveau de détail obtenu sera bien celui d'un langage évolué, dédié à la réalisation d'applications distribuées. Mais un des problèmes auquel on se frotte lors de la conception d'un langage est celui des primitives. Or, notre *framework* doit être ouvert, et permettre de manière naturelle l'adjonction de nouvelles primitives. La distinction langage haut niveau contre *framework* est donc principalement liée à un choix actuel de facilité d'implémentation pour des besoins d'ouverture à de nouveaux composants primitifs. Mais on ne s'interdit pas de réaliser plus tard le noyau d'un langage correspondant à la sémantique de la partie du *framework* représentant son cœur, son modèle de calcul, cette partie n'étant pas conçue pour être étendue par l'utilisateur.

4.4.2 Focalisation sur la mise au point des interactions

La différence technique majeure entre un jeu multi-joueurs en ligne et un jeu classique est bien l'interaction qui existe entre les joueurs et le reste du monde virtuel situé sur des machines distantes.

De même, la différence majeure entre un jeu multi-joueur et une application distribuée quelconque est la variété des interactions et la difficulté de leur calibrage.

C'est pourquoi nous nous sommes concentrés sur l'aide à la mise au point de ces interactions. Le travail présenté dans cette thèse ne concerne pas les aspects communs aux autres jeux-vidéo comme par exemple, les problématiques d'imagerie numérique, ou d'intelligence artificielle pour les personnages non-joueurs. Elle ne concerne pas non plus les problématiques de persistance et d'ingénierie de base de données communes aux applications distribuées en ligne plus classiques.

Cependant, pour pouvoir finaliser un projet de jeu massivement multi-joueurs, ces aspects doivent être pris en compte dans l'ébauche de notre solution. Le travail que nous avons réalisé se veut donc ouvert afin de pouvoir intégrer ultérieurement les autres technologies nécessaires.

4.4.3 Factorisation des aspects communs à la famille d'application visée

Concevoir un *framework* dédié à une certaine famille d'applications est un exercice délicat. Ce travail commence en général par une phase d'étude d'un certain nombre d'applications correspondant à ce qu'on veut pouvoir réaliser. Cependant, si la portée des applications étudiées n'est pas assez complet, le *framework* ne sera pas générique, (ce qui risque d'être le cas quand on veut l'utiliser pour développer des applications innovantes de toutes façons). D'un autre côté, il est pratiquement impossible d'extraire les points communs d'un trop grand nombre d'applications.

Au lieu d'étudier les jeux eux-même, nous avons étudié l'état de l'art des techniques actuellement utilisées pour la réalisation de divers types d'applications de la famille «Monde Virtuel», comme préconisé dans [38]. C'est le travail qui a été présenté dans le second chapitre de ce manuscrit.

Nous avons étudié précisément quels aspects de ces applications pouvaient modifier l'utilisation des ressources rentrant en ligne de compte dans le calibrage des interactions. Nous avons montré dans quelle mesure chaque solution visant à améliorer la qualité d'une caractéristique technique avait un impact sur les autres.

Cela nous a fourni les clés pour détecter les aspects devant rester ouverts et flexibles afin de permettre un éventail maximal de choix pour la réalisation des caractéristiques de l'application, et ceux pouvant être factorisés à l'intérieur du *framework*.

4.4.4 Détection du niveau d'abstraction et du degré de souplesse du *framework*

L'état de l'art présenté dans le deuxième chapitre de ce manuscrit a eu une conséquence majeure sur le niveau d'abstraction du *framework* : il est impossible de fournir une architecture de communication globale. L'impact des choix d'architecture sur l'utilisation des ressources réseau est trop importante pour pouvoir prétendre à la généricité.

Une autre conséquence majeure est que le modèle de communication de

chaque événement du jeu ne peut lui même être figé. Le calibrage du protocole de mise à jour de chaque donnée de l'application est là encore intrinsèquement lié à la consommation des ressources réseau. Compte tenu des interactions fonctionnelles à mettre en place, un calibrage très fin et spécifique à l'application peut être nécessaire.

C'est pourquoi le *framework* que nous proposons est basé sur les données répliquées sur les différents hôtes de la distribution. L'utilisateur doit pouvoir décider quelles données sont répliquées sur quels hôtes de la distribution, compte tenu de l'architecture qu'il a lui même choisi. Ainsi, il peut décider d'une architecture serveur divisée en service, d'une architecture centralisée, ou même d'une architecture *peer to peer*. De plus, le modèle de communication ainsi défini pour chaque donnée n'est pas nécessairement global. Chaque donnée peut avoir son propre modèle.

Enfin, la manière dont les données sont répliquées, c'est-à-dire la façon dont les données sont traitées par la couche réseau, est entièrement ouverte. Quelques briques de base paramétrables sont proposées en bibliothèque additionnelle au *framework*, et nous laissons la possibilité d'augmenter à volonté cette bibliothèque.

4.4.5 Pré-requis pour une intégration dans un environnement de développement

Enfin, puisque nous voulons ultérieurement intégrer le *framework* dans un environnement de développement, quelques autres aspects sont à prendre en considération, dont certains de nature purement ergonomique.

4.4.5.1 Différents niveaux de granularité

Nous avons vu dans les paragraphes précédents que nous avons finalement choisi un modèle de *framework* très bas niveau, où l'utilisateur doit lui même définir de quelle manière les données répliquées qui composent l'application seront traitées.

Ainsi, le découplage entre les données (organisées en état) et leur comportement différencie notre approche des solutions orientées objet, dont le

paradigme repose justement sur l'association des données et des traitements, de même que des approches orientées agent définies dans [87] ou [21].

Or, ce niveau de détail est trop fin pour pouvoir être utilisé à terme dans un outil de prototypage. Nous avons donc défini un modèle d'organisation des données en états. L'utilisateur peut ainsi factoriser les données en ensembles de valeurs à traiter de manière similaire.

Les états eux mêmes peuvent se composer arbitrairement de manière à ce qu'intuitivement, certains états définis par l'utilisateur puissent correspondre aux objets du jeu. Ainsi, une fois les données correctement organisées en états, l'utilisateur peut réfléchir et modéliser les interactions de manière naturelle et relativement intuitive pour un non programmeur.

L'avantage de cette solution est qu'une fois qu'un utilisateur averti a effectué la première étape de conception, un utilisateur moins averti, de profil intégrateur ou *game-designer*, peut lui-même jouer avec le paramétrage de l'application pour tester la qualité des interactions.

En fait, notre solution est à granularité arbitraire. Il sera toujours possible, en cas de besoin, de définir une interaction liée à une donnée de manière très fine et spécialisée par rapport aux autres données de l'application, mais en jouant avec la composition des états, on pourra toujours obtenir un niveau d'abstraction plus élevé.

Notre modèle permet ainsi, par une conception adéquate de l'utilisateur, de mettre en œuvre des approches orientées objet ou agent pour certaines données de l'application. Mais sans pour autant imposer que la totalité des données doive suivre l'un ou l'autre de ces paradigmes.

Cette possibilité de modélisation en plusieurs temps permet également de pallier le principal inconvénient du choix d'un *framework* bas niveau : les jeux simples resteront simples à prototyper, une fois les bonnes briques de base disponibles.

4.4.5.2 Un boîte à outils extensible et paramétrable

Nous avons vu dans les paragraphes précédents que le *framework* doit rester ouvert à des extensions concernant la manière dont sont traitées les fonctionnalités, notamment la façon dont sont propagées les données, ou celle

de calculer à quels hôtes les envoyer.

Nous obtenons ainsi un découplage entre les fonctionnalités et leur traitement, qui peut se rapprocher de l'objectif recherché par les techniques de meta-programmation comme le modèle de programmation par *aspects* [19], grâce à un *framework* ouvert qui permet à l'utilisateur de connaître, de modifier ou d'étendre les différentes implémentations d'une même fonctionnalité. Pour montrer l'adéquation de notre solution, nous proposons une boîte à outils minimale et paramétrable permettant de mettre en application quelques recettes techniques vues dans l'état de l'art présenté dans le deuxième chapitre, et pouvant ainsi servir de modèle à la réalisation ultérieure d'autres briques de base.

4.4.5.3 Du déterminisme et une sémantique bien définie

Enfin, un bon outil de développement doit pouvoir fournir des outils d'analyse du code produit, statiques et dynamiques.

De plus, la mise au point d'une application est toujours facilitée lorsque l'on peut facilement en tracer et en reproduire l'exécution. Or dans le cas d'une application en ligne, cette tâche est compliquée par son caractère distribué : il est très difficile de reproduire à l'identique le fonctionnement de la totalité de l'application, quand les protocoles Internet utilisés ne fournissent pas de garantie d'ordonnancement des messages ou d'assurance qu'ils seront bien reçus. Si on ajoute à cela le fait que beaucoup d'algorithmes utilisés dans le développement des jeux massivement multi-joueurs sont plus naturellement décrits en utilisant un paradigme de programmation concurrente (un système de processus légers par exemple), on obtient des applications très difficiles à corriger et à maintenir.

C'est pourquoi nous avons choisi d'introduire une forme légère de concurrence dans le *framework* lui-même. Le modèle que nous avons conçu est inspiré du modèle *Fair-Thread* (voir section 3.1.3, page 75) développé par Frédéric Boussinot [20]. Ce modèle est déterministe et sa définition sémantique est claire, ce qui facilite son utilisation et sa mise au point. L'adaptation que nous en avons faite rend notre modèle de description des interactions et de comportements des états du jeu facile à exprimer dans un contexte de programmation concurrente.

Une application développée à l'aide de notre *framework* n'est bien sûr pas reproductible dans l'absolu, puisqu'il s'agit d'une application distribuée sujette aux aléas des communications sur l'Internet. Mais il permet à l'utilisateur de disposer d'un cadre clair et bien défini, qu'il peut utiliser pour circonscrire les problèmes rencontrés au cours du développement en sous ensemble pour lequel le fonctionnement de l'application est reproductible (par exemple en scénarisant les événements provenant des autres hôtes de la distribution). Et cela tout en gardant la puissance d'expression que fournit le modèle de programmation concurrente.

4.5 Modèles de réplication et réflexes d'états du jeu

Dans cette partie, nous allons donner une première présentation informelle du modèle réalisé au cours de cette thèse.

4.5.1 Un modèle d'interactions basé sur la définition d'états

Un jeu massivement multi-joueurs peut être vu comme un ensemble de données répliquées sur tous les hôtes de la distribution. Ces données servent à représenter les objets du jeu et nous avons déjà vu que le principal problème dans la mise au point de ces applications était la manière dont les données doivent être répliquées.

Un des aspects à prendre en compte dans la réplication des données est l'architecture distribuée utilisée : par exemple, dans une architecture logique clients-serveur à base de *proxies*, où un serveur central est responsable de gérer la cohérence de l'état de l'application et où les *proxies* sont responsables de la gestion et de l'optimisation des communications en provenance des clients, l'état du jeu est répliqué sur le serveur central et sur les clients de l'application. Dans une architecture basée sur la division en zones géographiques du monde virtuel, seulement une partie de l'état du jeu doit être répliquée sur chaque serveur de zone, et sur les clients qui sont actuellement connectés sur ce serveur de zone.

Pour donner un exemple concret, la position de joueurs dans la région du monde virtuel gérée par un serveur de zone peut être uniquement gérée par ce serveur.

Comme nous l'avons vu dans le chapitre 2, l'architecture de l'application et le choix de l'ensemble des hôtes de l'application pour lesquels les données doivent être répliquées sont des problèmes de conception, qui ont un impact sur la qualité de l'interaction à laquelle ils sont liés.

De plus, il serait intéressant d'utiliser un modèle de communication différent selon la nature des données à répliquer : la gestion de certaines données pourrait être alors centralisée sur un seul serveur logique quand la consistance de l'état de l'application est un facteur très important, tandis que la réplication des données intervenant dans des interactions pour lesquelles l'aspect temps réel prime pourrait suivre un modèle de communication exploitant des responsabilités plus distribuées des serveurs. Le modèle que nous proposons ne se base donc pas sur une architecture de communication pré-définie.

Dans le modèle que nous définissons, l'entité de base que nous manipulons est un *état*.

Définition 4.5.1 *Un* **état** *encapsule un ensemble de données corrélées : elles ont les mêmes propriétés de réplication.*

Les données ne peuvent pas être partagées par deux états différents.

Un état peut avoir des sous-états. Les données représentant l'application sont donc strictement organisées en arbre.

Intuitivement, un état peut servir à représenter un objet du jeu.

4.5.2 Modèles de réplication

Quand un joueur interagit avec un objet du jeu sur son propre terminal, il se produit une modification de l'état du jeu sur sa machine cliente. Cette mise à jour de l'état du jeu doit être propagée au reste de l'application (même si ce changement d'état peut parfois seulement être l'enregistrement de la dernière commande provoquée par le joueur avant l'envoi sur le réseau). De la même manière, le résultat d'un algorithme comme la description du comportement

d'un personnage non joueur, ou n'importe quel autre processus s'effectuant sur le serveur intervient sur l'application et provoque des changements de l'état du jeu. Ainsi, modéliser une interaction dans notre modèle consiste à définir quand, comment et où la mise à jour d'un état doit être propagée le long de l'architecture de distribution de l'application, et quelles sont les conséquences de cette mise à jour sur les autres états. Nous proposons pour ce faire la notion de *modèle de réplication*.

4.5.2.1 Terminologie des modèles de réplications

Un modèle de réplication représente donc le moyen utilisé par un hôte de l'application distribuée pour propager la valeur de l'état aux autres hôtes.

Un modèle de réplication comporte trois attributs, définis comme suit :

Définition 4.5.2 *La* **portée** *est la liste des hôtes à qui propager la mise à jour de l'*état.

Par exemple, si le changement d'état correspond à un déplacement d'un joueur ou d'un personnage non-joueur, seuls les joueurs qui peuvent effectivement observer ces modifications, de par leur localisation dans le monde virtuel, ont réellement besoin de connaître ces modifications.

Définition 4.5.3 *La* **temporalité** *décrit quand propager la mise à jour de l'état.*

Cela peut être défini en donnant un rythme périodique auquel correspondront les réplications, ou en donnant une condition sur les valeurs d'un ou plusieurs états du jeu.

Définition 4.5.4 *Les* **propriétés de communication** *définissent la manière dont la mise à jour de l'état doit être propagée.*

Cet attribut est fortement lié au modèle de communication bas niveau, aux propriétés des protocoles utilisés et inclut par exemple les aspects de fiabilité, d'ordonnancement, de cryptage ou de compression.

Un modèle de réplication est donc dépendant du contexte de l'état du jeu : chaque attribut peut être défini par rapport à un ensemble fini de valeurs locales des états du jeu. Par exemple, si on considère un état représentant le déplacement d'un joueur dans le jeu, on peut vouloir utiliser une technique de filtrage pour décider que seuls les hôtes clients dont l'avatar est à une certaine distance de l'avatar du joueur qui se déplace ont besoin de recevoir la mise à jour. Dans ce cas, l'attribut **portée** consiste à passer en revue les positions des autres joueurs de la zone, afin de décider à quels clients l'état doit être propagé.

Il est possible de combiner plusieurs modèles de réplication pour la gestion des mises à jour d'un seul état : certaines données peuvent en effet nécessiter de répondre à différentes exigences de réplication.

Par exemple, considérons un état représentant la position actuelle de l'avatar d'un joueur donné sur un serveur de zone dans le cadre d'une architecture de distribution hiérarchique où un serveur central gère plusieurs serveurs de zone, chacun dédié à la maintenance d'un sous-ensemble de l'état du jeu suivant les régions du monde virtuel. Plusieurs modèles de réplication définissent la manière plus ou moins temps réel dont la position de l'avatar concerné doit être propagée aux autres clients de l'application, en fonction de sa distance par rapport aux positions des avatars des joueurs correspondants. Le serveur central, relié à une base de données assurant la persistance de l'application, peut utiliser un modèle de réplication pour être prévenu périodiquement afin d'enregistrer de temps en temps la position du joueur, en cas de défaillance de l'application. Il peut en utiliser un autre pour enregistrer la position de l'avatar en cas de déconnexion du joueur.

4.5.2.2 Réflexes

Afin de pouvoir définir les conséquences d'un changement d'état sur un hôte de l'application, on définit également la notion de **réflexe**.

Définition 4.5.5 *Un* **réflexe** *est une partie de code défini par l'utilisateur, opérant sur un ensemble d'états de l'application, et lié à l'attribut* temporalité *du modèle de réplication. Lorsque la* temporalité *est vérifiée, la réplication est déclenchée en utilisant les autres attributs du modèle de réplication, puis le réflexe est exécuté.*

Un réflexe peut servir à effectuer des calculs arbitrairement complexes, allant de la simple réinitialisation de la valeur d'un état à la description d'un comportement autonome d'un objet du monde virtuel.

Les réflexes peuvent également être utilisés pour définir l'impact d'un changement d'état sans qu'il soit nécessaire que cet état soit sujet à une réplication. Dans ce cas, le réflexe est défini sans liaison avec un modèle de réplication, mais avec un attribut temporalité qui permet de spécifier son déclenchement.

Les réflexes, une fois déclenchés par la vérification de leur temporalité, sont exécutés par le *framework* selon une sémantique que nous étudierons dans le chapitre suivant.

Nous pouvons désormais donner une définition d'un modèle de réplication d'un état :

Définition 4.5.6 *Un modèle de réplication d'un état (définition 4.5.1) est la donnée des trois attributs portée (définition 4.5.2), temporalité (définition 4.5.3) et propriétés de communication (définition 4.5.4), et d'un éventuel réflexe post-réplication (définition 4.5.5).*

4.5.2.3 Intuition sur un exemple simple

Nous reprenons le cas de figure évoqué précédemment des modèles de réplication de la position de l'avatar d'un joueur, dans le cadre d'une architecture clients-serveur centralisée où des serveurs de zone sont en charge de chaque zone géographique du monde virtuel.

Le tableau 4.1 décrit ce qui se passe sur un serveur de zone lorsqu'un joueur se déplace. L'état **position** représente la position de l'avatar du joueur, et on décrit les modèles de réplications qui lui sont associés :

- Le premier modèle de réplication (*Voisins*) utilise les états correspondant à la position des avatars connectés au même serveur de zone pour filtrer quels sont les avatars voisins étant donnée une certaine distance (qui peut elle-même également être définie comme un état) pour définir la portée. La temporalité est conditionnée par le changement d'état et la propagation de ce changement est définie comme non fiable et non ordonnée.

- Le deuxième modèle de réplication (*Zone*) a une portée qui est l'ensemble des clients connectés au même serveur de zone, avec une temporalité périodique de 500ms. Les propriétés de communication suivent un protocole non fiable et ordonné comme dans le modèle précédent.
- Le troisième modèle de réplication (*Central*) a une portée constituée uniquement du serveur central, avec une temporalité périodique de 20 secondes, et utilise un protocole fiable et ordonné. Elle permet de gérer la persistance de l'application.

Réplications	*Voisins*	*Zone*	*Central*
portée	ensemble des voisins, calculé à l'aide d'un filtre	clients de la zone	serveur central
temporalité	changement de l'état **position**	toutes les 500 ms	20s
propriétés de communication	non fiable, ordonné	non fiable, ordonné	fiable

FIG. 4.1 – Réplications du mouvement d'un avatar sur un serveur de zone

4.5.3 Vue d'ensemble

La figure 4.2 décrit l'interaction des différents processus intervenant dans la partie cliente d'une application réalisée à l'aide du *framework*. Ce fonctionnement est classique. Une mémoire partagée par les différents processus contient l'ensemble des états définis pour représenter l'état de l'application. Les événements provenant du réseau et les événements provoqués par le joueur à travers l'interface utilisateur du jeu provoquent des changements dans cette mémoire partagée. Le joueur perçoit l'état du jeu à travers un ou plusieurs observateurs utilisés par l'interface graphique, comme un moteur de rendu 3D par exemple. Un processus particulier gère le déclenchement des modèles de réplication et l'exécution des réflexes définis par l'utilisateur. Ce processus est à la fois observateur de l'état du jeu et producteur de modifications des états.

Les messages arrivant par le réseau doivent correspondre à la réplication d'un état par un hôte distant, et cet hôte doit être autorisé localement à modifier l'état pour que la mise à jour soit prise en compte. Si l'état n'existe

pas sur localement et que l'hôte distant y est autorisé, le message réseau de réplication provoquera la création de l'état local.

Des processus supplémentaires peuvent naturellement être intégrés à l'application pour travailler sur ou à partir de la mémoire partagée, mais nous verrons plus tard que cela peut avoir des conséquences sur la clarté de la sémantique de l'application, car ces processus ne s'exécuteront pas selon le calcul que nous avons défini pour l'exécution des réflexes.

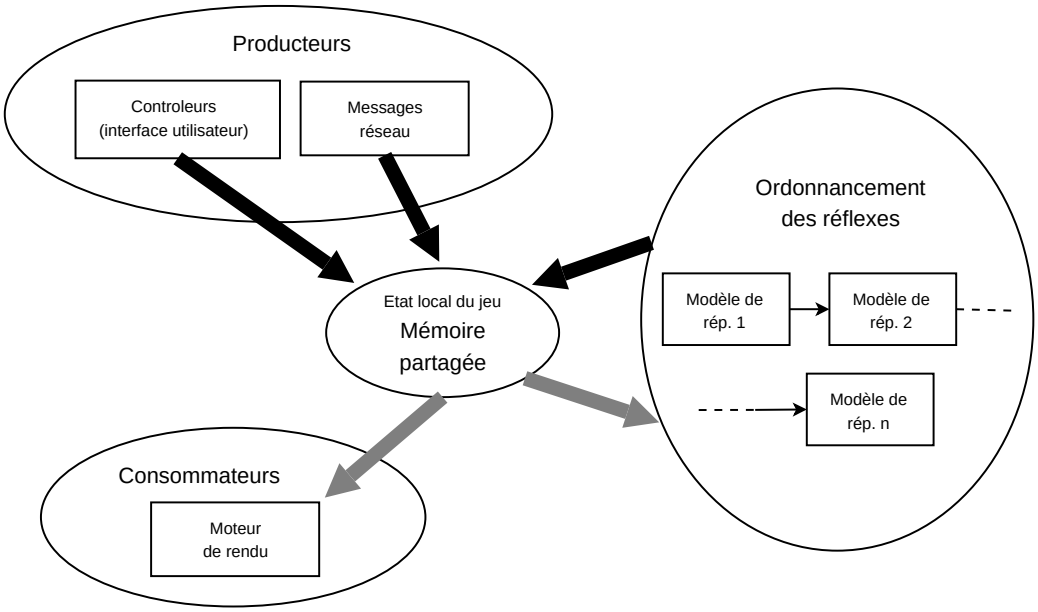

FIG. 4.2 – Architecture des processus de l'application sur un hôte

La figure 4.3 décrit l'organisation des différents éléments intervenant dans le fonctionnement de l'application du point de vue de l'utilisateur du *framework*. Pour chaque différent type d'hôte de l'application (les clients et les hôtes de l'architecture du serveur), l'utilisateur a pour tâche de définir les états à partir des données permettant de représenter l'application.

Une fois les données organisées en états, il utilise les composants de base du *framework* afin de définir les éventuels modèles de réplication pour ces états. Comme représenté sur la figure, un état n'a pas nécessairement de modèle de réplication : il n'est pas toujours nécessaire de propager la modification d'une donnée, soit parce qu'elle est privée à l'hôte et est utilisée pour

la valeur de ses données pour les calculs définis dans les réflexes ou la description des attributs, soit parce que celui-ci n'a pas la responsabilité de sa mise à jour qui s'effectue au travers du réseau par un hôte distant. Un même état peut avoir plusieurs modèles de réplications correspondant à différents besoins de persistance.

Le corps du *framework* s'occupe de déclencher les modèles de réplication et d'exécuter avec une sémantique bien définie les réflexes attachés ou non à ces modèles, modifiant éventuellement les différents états de la mémoire partagée par les processus locaux de l'application.

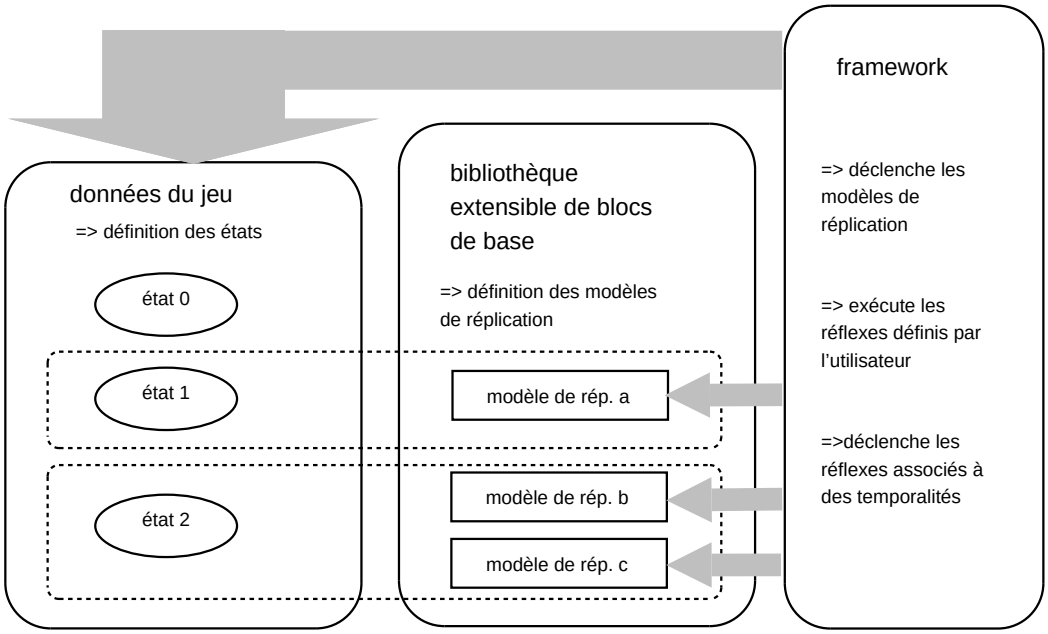

FIG. 4.3 – Modèles de réplications d'états sur un hôte de la distribution

4.5.4 La bibliothèque des blocs de base

Comme nous l'avons vu, l'utilisateur du *framework* que nous proposons s'appuie sur une bibliothèque extensible de blocs de base éventuellement paramétrables. Ceux-ci sont rangés en plusieurs grandes familles, et servent à construire les modèles de réplication ou les réflexes sans réplication attachés aux états.

- les *propriétés de communication* des réflexes.
- les *portées*, qui peuvent être des ensembles statiques de destinataires, ou calculés dynamiquement selon les valeurs d'états de l'application.
- des *réflexes* liés à une *temporalité*, décrivant des comportements d'états, paramétrables par ces derniers, qui peuvent ainsi permettre de constituer des bibliothèques de description de comportements d'objets du monde virtuel.
- des *modèles de réplication* génériques paramétrables, dont la temporalité (les conditions de déclenchement) est retrouvé fréquemment : par exemple une réplication instantanée lors d'un changement d'état.

Les deux dernières familles de composants se situent à un niveau d'abstraction supérieur, les modèles de réplication génériques étant paramétrables par des portées et des propriétés de communication.

L'utilisateur peut étendre cette bibliothèque en utilisant des interfaces fournies par le *framework*, à la manière des *Hooks* décrits dans [38].

4.5.5 Exemples de modélisation, cas d'école

Nous présentons ici trois exemples simples, modélisés volontairement de manière très détaillée, pour donner au lecteur une intuition de la manière d'utiliser le modèle pour concevoir quelques interactions classiques intervenant dans un jeu.

4.5.5.1 Compétition pour accéder à un objet : ouverture d'un coffre

Dans le jeu, un coffre fermé par un cadenas peut être ouvert par les joueurs disposant du bon code. Deux joueurs essaient d'ouvrir le code simultanément.

L'architecture distribuée choisie par les concepteurs est une architecture clients-serveur classique. Le concepteur des interactions fonctionnelles décide que le contenu du coffre est accessible seulement au joueur qui l'ouvre le premier avec le bon code. Il décide également que l'attribution du verrou logique n'a pas à être équitable : si les deux joueurs entrent le bon code exactement au même moment, c'est celui dont l'information sera traitée en premier par le serveur qui pourra explorer le contenu du coffre. C'est donc le

joueur qui aura le moins de latence qui sera favorisé. Dans cet exemple, les objets du jeu concernés sont le *coffre*, et les *avatars* des deux joueurs.

La modélisation bas-niveau que nous donnons est destinée à expliquer comment traiter un modèle de communication par requête, même si celles-ci sont moins naturellement traitées que dans le cas d'objets répliqués. Bien sûr, il y a bien d'autres manières de modéliser cette interaction.

Parmi les états définissant le *coffre*, on trouve :
- **utilisateur :** contient l'utilisateur qui a obtenu le verrou sur le coffre, une valeur par défaut s'il n'y en a aucun.
- **code entré :** la valeur du dernier code entré pour ouvrir le coffre.
- **code secret :** le code secret permettant d'ouvrir le coffre.
- **demandeur :** le client demandant actuellement accès au coffre.

Le tableau 4.4 représente les modèles de réplication associés aux états définissant le coffre et les réflexes. Quand les joueurs entrent le code, l'état

états répliqués	***code entré***	***code entré***	***utilisateur***
localisation	client	serveur	serveur
portée	serveur		**demandeur**
temporalité	**code entré** modifié, non défaut	**code entré** modifié	**utilisateur** modifié
communications	fiable		fiable
états modifiés par les réflexes	**code entré**	**demandeur**, **utilisateur**	

FIG. 4.4 – Réplication et réflexes pour les états représentant le coffre

code entré est modifié sur chaque client. Les modèles de réplication côté client provoquent une communication immédiate de ce changement au serveur, puis le réflexe associé réinitialise la valeur de ces états clients à la valeur par défaut.

Côté serveur, lorsque la première communication de la nouvelle valeur de l'état **code entré** arrive, l'état **code entré** local est mis à jour. Le réflexe serveur associé au changement d'état met à jour l'état **demandeur** avec la valeur du client qui a effectué la mise à jour de **code entré** et fait quelques vérifications afin de modifier l'état **utilisateur** : si l'état **code entré** contient la même valeur que l'état **code secret** et si l'état **utilisateur** contient la valeur par défaut (pas d'utilisateur du coffre), il modifie dans l'état **utilisateur**

la donnée représentant le propriétaire du coffre pour l'attribuer au client demandeur. Sinon, il provoque également un événement de modification de cet état, mais sans modifier la donnée encapsulée représentant le propriétaire du coffre.

Puis, le **framework** applique le modèle de réplication pour l'état **utilisateur** : la valeur courante de l'état **utilisateur** est envoyée au client qui a demandé le verrou en utilisant la valeur de l'état **demandeur**.

Pour modéliser cet exemple, une fois qu'il a correctement défini les états, l'utilisateur du **framework** doit simplement décrire les attributs des modèles de réplication, et définir le code des réflexes. Ce travail consiste à décrire des traitements sur les états définis, et à utiliser des composants de base faisant partie de la bibliothèque.

4.5.5.2 Combat en temps réel dans une zone peuplée par des joueurs

On s'intéresse maintenant à l'interaction suivante : dans un jeu persistant d'action temps-réel de style FPS, un joueur tire sur un autre à l'aide d'une arme.

L'architecture du serveur de l'application suit un découpage de la géographie du monde virtuel en serveurs de zone, et un serveur central est en charge de la coordination des serveurs de zone et de la gestion des données persistantes de l'application. Chaque client est connecté au serveur en charge de la zone géographique du monde virtuel dans laquelle se trouve son avatar. Des techniques d'extrapolation sont utilisées côté client, pour anticiper les futures positions des avatars selon les valeurs précédentes de leurs directions et vitesses.

Si le joueur visé est frappé aux jambes, son total de points de vie et sa vitesse de course décroissent. Comme cette action peut se produire dans une région vaste et très peuplée du monde virtuel et qu'informer tous les joueurs en temps réel consommerait trop de ressources, le concepteur des interactions fonctionnelles décide que les proches voisins du joueur visé doivent pouvoir constater le plus rapidement possible la baisse de la vitesse de course du joueur touché, tandis que les joueurs plus éloignés n'ont pas besoin de constater ce changement aussi instantanément.

En ce qui concerne le total des points de vie du joueur touché, le concepteur décide qu'il doit être communiqué à tous les clients du serveur de zone de manière non fiable, et avec une très haute priorité. La «mort» d'un avatar est un événement crucial, qui doit être communiqué de manière fiable par le serveur de zone au serveur central, qui est en charge des données persistantes de l'application, et à tous les clients de ce même serveur de zone.

Ici, notre but est de décrire comment des techniques de filtrage selon les intérêts peuvent être modélisées. Nous allons décrire cette interaction à partir du moment où le serveur a déjà calculé que l'avatar visé a été touché. L'objet à décrire est l'avatar du joueur. Il comporte notamment trois états qui sont respectivement :

- **déplacement** : les caractéristiques du dernier mouvement de l'avatar, c'est-à-dire, en plus de sa position, sa vitesse et sa direction instantanée ;
- **points de vie** : la quantité des points de vie que l'avatar peut encore perdre avant d'être considéré comme décédé. Comme l'avatar peut être soigné par un joueur partenaire dans le même temps où il est touché par un autre, et que les événements correspondants peuvent arriver dans le désordre par rapport au moment où il se sont produits, on décide d'appliquer une tolérance : un avatar peut brièvement posséder un nombre de points de vie négatif sans être encore considéré comme décédé.
- **vivant** : une valeur indiquant si l'avatar est encore vivant ou non. Cet état n'est pas redondant avec le précédent suite à la tolérance décrite.

La figure 4.5 décrit les modèles de réplication sur le serveur de zone où l'avatar se situe au moment de l'action. Le processus serveur décide que l'avatar est touché, et met à jour les états précédemment décrits selon le *game-design* défini. Ensuite, il exécute les modèles de réplication pour chaque état dont l'attribut temporalité est vérifié. Dans la description de cette interaction, outre la manière dont on décide si l'avatar a été touché, la seule partie spécifique, décrite par l'utilisateur du *framework*, est la manière de calculer la liste des hôtes devant recevoir les mises à jour d'états répliqués. Comme ces techniques de filtrage sont des techniques couramment utilisées, on peut envisager de fournir cette partie spécifique comme un bloc de base paramétrable faisant partie d'une bibliothèque accompagnant le *framework*. On peut tout simplement décider si oui ou non deux avatars sont voisins par rapport aux valeurs respectives de leurs états *position*.

L'algorithme d'extrapolation côté client peut également être un bloc de

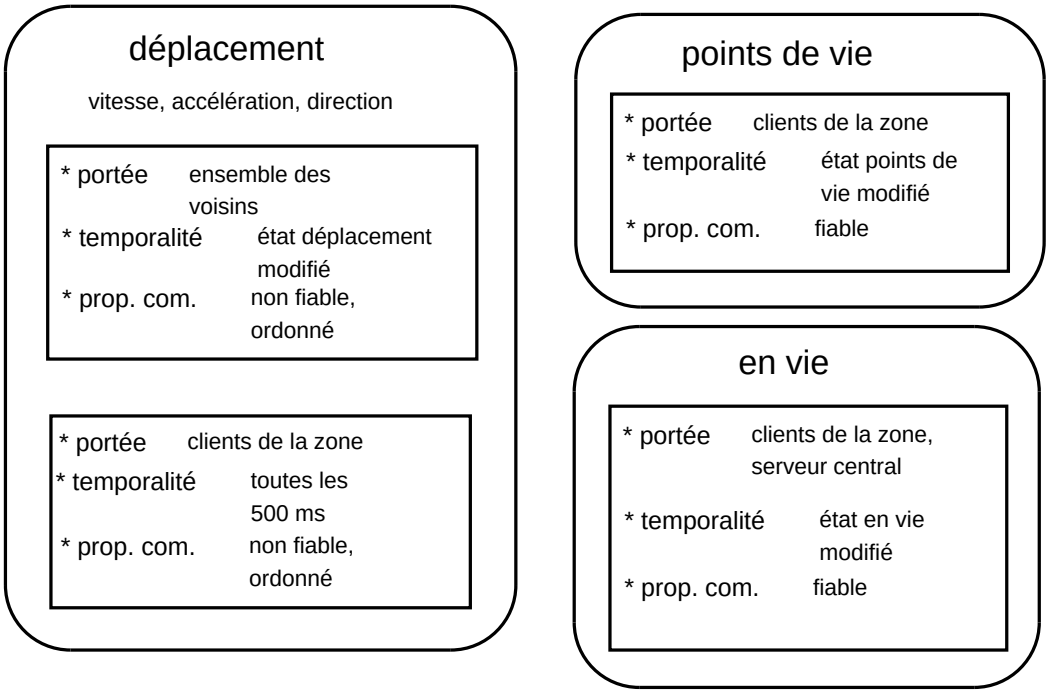

FIG. 4.5 – Réplications du mouvement d'un joueur sur un serveur de zone

base paramétrable inclu dans la bibliothèque intégrée au *framework*, et paramétrable par l'utilisateur.

En utilisant la modélisation de cette séquence à l'aide du *framework*, conjointement avec des outils de simulation réseau introduisant latence et perte de paquets, et avec un outil permettant de simuler la présence d'un grand nombre de clients, le *game-designer* peut alors tester si le *game-play* est réalisable et les choix techniques adéquats. Par exemple, il peut étudier l'impact de différents paramètres de filtrage sur la précision de l'algorithme d'extrapolation et tester si les caractéristiques de jouabilité restent satisfaisantes. Il peut aussi étudier l'impact d'une perte de paquets réaliste sur le nombre des points de vie par rapport à l'information de la mort d'un avatar, toujours afin de vérifier si les joueurs ne risquent pas d'être surpris par une évolution trop rapide de l'action, au cas où la plupart des paquets soient perdus.

4.5.5.3 Adaptation dynamique de consommation de bande passante

Notre modèle peut également être utilisé pour décrire un équilibrage dynamique de consommation de ressources, par exemple pour la consommation de bande passante.

Dans le cadre d'un jeu grand-public, conçu pour pouvoir être utilisé par des joueurs disposant de ressources hétérogènes en terme de bande passante, et qui n'observent pas les mêmes latences moyennes, le concepteur peut décider d'utiliser un bloc de base comportemental qui analyse dynamiquement un état représentant les capacités moyennes observées pour chaque client.

Ainsi, à l'aide des mêmes techniques de filtrage que celles de l'exemple précédent, mais en utilisant non pas la position de l'avatar mais les valeurs des états représentant les capacités du client, le concepteur peut modéliser un mode dégradé de comportement de l'application pour les joueurs disposant de faibles connexions.

4.6 Autres approches génériques

Dans cette partie, nous allons situer notre approche avec deux autres solutions récentes, proposant des *frameworks* génériques dédiés à la réalisation de jeux massivement multi-joueurs.

4.6.1 *Virtual Environment System Object Model* et JADE

Dans [79], Manuel Oliveira propose une architecture de développement flexible pour la réalisation de mondes virtuels allant des jeux massivement multi-joueurs aux FPS, constituée de quatre couches de développement orienté objet construites l'une au dessus de l'autre, nommée *Virtual Environment System Layered Object Model* (VESLOM). Ce modèle a été défini afin de résoudre le manque de souplesse des solutions existantes et de proposer une solution modulaire et évolutive, au fur et à mesure que des nouveaux besoins apparaissent. La volonté de fournir une approche générique est commune avec le travail réalisé dans cette thèse.

Voici les quatres couches successives sur lesquelles reposent VESLOM.

1. La **plate-forme universelle** rassemble les fonctionnalités nécessaires à tout développement de monde virtuel. Lorsqu'un système construit à l'aide de VESLOM devra évoluer pour s'adapter à de nouveaux systèmes et de nouvelles technologies, les modifications devraient donc être principalement localisées sur cette plate-forme. JADE (Java Adaptative Dynamic Environment), la proposition de plate-forme universelle développée par l'auteur en Java consiste en quatre modules. Le premier sert à identifier toutes les ressources de l'application, des textures graphiques aux éléments de code modulaire, qui sont fournies par les couches supérieures. Le deuxième module fournit un modèle événementiel souscription/publication pour la communication entre les composants du système. Un troisième module est dédié à la localisation de ressources, en local ou de manière distante. Le dernier module permet d'adapter des interpréteurs pour différents langages de description de contenu (par exemple *XML*), afin de découpler le code de l'application de son contenu et donc de permettre la réalisation de mondes virtuels de contenus différents à l'aide du même système.
2. Le rôle de la **couche réseau** est d'utiliser la plate-forme universelle pour implémenter des protocoles réseau plus élaborés pour la communication dans le monde virtuel.
3. La **couche logicielle** regroupe tous les composants nécessaires à la constitution d'un monde virtuel. Le but de cette couche est de construire un ensemble modulaire de composants. Cependant, il n'y a pas d'impératifs quant au nombre ou à l'autonomie de ces composants : cette couche est prévue pour être extensible. Le développeur peut ajouter de nouveaux composants, et définir des ensembles de composants dépendants les uns des autres.
4. La **couche applicative** regroupe tous les composants spécifiques au monde virtuel en cours de développement.

Cette approche est donc moins monolithique que celles que nous avons rencontrées en passant en revue les différentes solutions aux problèmes techniques rencontrés dans la deuxième partie de ce manuscrit.

Un des points communs avec le travail présenté dans cette thèse est l'utilisation d'une boîte à outils extensible et paramétrable, construite autour d'un modèle d'exécution (ici la plate-forme universelle) qui se veut générique. Le

modèle de construction en *oignon* de VESLOM nous paraît néanmoins moins souple et moins fin que celui que nous définissons, car les différentes couches sont construites les unes au dessus des autres. Dans notre modèle, les grandes familles de briques de base sont placées au même niveau et se combinent entre elles pour former la manière dont l'application se comportera.

De plus, la méthode accompagnant le modèle VESLOM reste guidée par l'architecture de conception du logiciel pour un hôte de l'application distribuée, alors que notre démarche est guidée par la définition des interactions le long d'une architecture de distribution. Une des conséquences de cette différence est que certaines fonctionnalités de la plate-forme universelle, comme les différents moyens d'accéder à une ressource distante (par HTTP ou Jini [4] dans JADE), deviendraient des briques de base dans notre approche.

4.6.2 OpenPING

Le système OpenPING [78] est une version améliorée du projet européen PING [83] dédié à la création d'une solution pour les applications de simulation massivement multi-utilisateurs tels que les jeux massivement multi-joueurs. Les produits du projet PING sont notamment la plate-forme Continuum [97], accompagnée d'un langage de description de comportement d'agents évoluant dans des mondes virtuels [21] (utilisant les principes de la programmation réactive, et déjà cité dans la partie de ce mémoire consacré à ce modèle de programmation), dont OpenPING reprend les principaux aspects, en y ajoutant des propriétés de réflexion.

Tout comme notre proposition, OpenPING est un *framework* orienté objet ouvert, qui donne une visibilité à l'utilisateur sur les cinq principaux services permettant de réaliser les fonctionnalités de l'application, sollicités lorsqu'un événement se produit dans l'application.

- Le service de *réplication* fournit des mécanismes de mise à jour périodique des données de l'application, pour des rythmes rapides, moyens ou lents.
- Le service de *concurrence* assure le transfert de propriété des objets du jeu entre les différents hôtes de l'application.
- Le service de *consistance* comprend des algorithmes de synchronisation des différents hôtes de la distribution, quand le service de réplication n'est pas utilisé ou pas suffisant. Il définit donc d'autres modèles de

communication.
- Le service de *gestion des intérêts* fournit des algorithmes de communications de groupe, basés sur les coordonnées spatiales des objets de l'application, ou sur un modèle publication/souscription.
- Le service de *persistance* est en charge de la maintenance de l'état de l'application, que ce soit dans la mémoire locale (pour des jeux par session par exemple) ou sur support externe comme une base de données.

La description des comportement des objets de l'application, qu'ils correspondent à des objets du monde virtuel ou à des mécanismes système, est faite dans une couche applicative de niveau d'abstraction supérieur.

Les quatres premiers services utilisent un composant en charge de la transmission des événements aux autres hôtes de l'application, consistant en un mode non fiable non ordonné (UDP), un mode fiable et paramétrable au niveau du composant, et un mode paramétrable au niveau de la couche applicative.

L'amélioration principale apportée au projet PING par Paul Okanda et Gordon Blair est l'ajout de propriétés de *réflexion* au système originel. Cette propriété est généralement définie comme la possibilité pour un système de raisonner sur sa propre structure et sa propre exécution à l'aide d'une représentation de lui-même. Le travail présenté dans [66] passe en revue quelques langages permettant d'exprimer cette propriété et décrit un exemple d'architecture réflexive pour un langage orienté objet. Les auteurs re-définissent cette propriété dans le cas particulier des mondes virtuels : *un principe de conception qui permet à une plate-forme de jeu et/ou à l'application d'avoir une représentation d'elle-même afin de rendre possible son adaptation à un environnement qui évolue.*

Les auteurs d'OpenPING utilisent donc ce concept afin de fournir un mécanisme qui permet à l'application de s'adapter automatiquement et dynamiquement à son propre état : par exemple, lorsque les communications se déroulent parfaitement bien au sein de l'application distribuée, la gestion des intérêts utilise le mécanisme reposant sur la localisation spatiale des objets du jeu, et s'adapte en passant à un modèle publication-souscription en mode dégradé, si le système observe que le réseau sature.

Il nous semble que ce système ouvert, d'une architecture plus complexe mais plus haut-niveau que la nôtre, est équivalent en terme d'applications

réalisables à notre modèle.

Le découplage entre les fonctionnalités à réaliser et la manière dont elles sont réalisées, au moyen de services extensibles, se rapproche d'un modèle de programmation par aspect [19]. Dans notre *framework*, la possibilité de définir plusieurs manières dont un état sera répliqué à travers plusieurs modèles de réplications, se rapproche d'une telle approche transverse. De plus, nous avons montré dans la section 4.5.5.3 (page 125) comment le découplage entre les états et les différentes manières de les répliquer peut être utilisé à profit pour fournir une application qui s'adapte automatiquement à un changement de contexte d'exécution (en l'occurrence la quantité de bande passante du client).

4.7 Synthèse

Le modèle que nous proposons est donc centré sur une modélisation très fine de la manière dont les données du jeu, organisées en états, doivent être répliquées le long de l'architecture distribuée. La conception d'une application utilisant ce modèle est donc guidée par la définition des interactions.

À la différence des modèles orientés objets et orientés agents, il découple les données des traitements, et de leurs comportements. Mais il est possible de réaliser des applications correspondant à ces paradigmes en modélisant de manière adéquate l'agencement des états et leur association avec des modèles de réplications et des réflexes sans réplications associées.

Nous avons également étudié dans cette partie d'autres propositions de *framework* génériques qui semblent disposer des mêmes propriétés de modularité que la nôtre, mais guidées par l'architecture de l'application ou une définition des services à mettre en oeuvre.

À notre connaissance, notre démarche est la seule qui prend en compte les impératifs liés à la construction d'un environnement de développement et de prototypage destiné à la mise au point des interactions pour les jeux massivement multi-joueurs.