Synchronisation dans le domaine applicatif des jeux en réseau sur mobile

Rapport de stage de DEA

Étudiant :
Falempin Sven.
DEA Intelligence Artificielle et Optimisation Combinatoire.
Institut Galilée, Université Paris 13
99 av. Jean-Baptiste Clément 93430 Villetaneuse
http://www-galilee.univ-paris13.fr/

Encadrant :
Chabridon Sophie.
Enseignant chercheur INF/Répartition.
Institut National des Télécommunications,
Département INFormatique, 9 rue Charles Fourier
91011 Évry Cedex - FRANCE
http://www-inf.int-evry.fr/

Résumé

Dans le cadre du projet de MEGA, nous menons une étude destinée à proposer des jeux multijoueurs sur téléphone portable et nous nous intéressons plus particulièrement aux problèmes de synchronisation. Nous présentons, en premier lieu, les jeux multijoueurs pour évaluer les besoins des jeux vidéo multijoueurs sur mobile en terme de synchronisation. Les types de jeux sont nombreux et les besoins spécifiques à chaque jeu. Les jeux d'action ont besoin de temps de latence faibles pour gérer des interactions, le temps de latence restreint directement les choix d'interfaces. Pour garantir le déroulement de l'application distribuée, on utilise des techniques de synchronisation de processus optimiste ou pessimiste. Nous présentons ces techniques pour évaluer leurs capacités à synchroniser des jeux vidéo sur téléphone portable. Puis nous présentons des techniques plus spécifiques destinées à passer à l'échelle ou à compenser le temps de latence. Enfin dans le cadre d'un jeu de plate-forme sur mobile prévu pour trois joueurs, et utilisant HTTP via GASP pour communiquer, nous présentons une technique de synchronisation adaptée au monde mobile et au GPRS.

Sommaire

Glossaire

Sigles:

ADSL: Asymetric Digital Subscriber Line : débit maximal environs 3 Mbit/s.
BREW: Binary Runtime Environment for Wireless
CLDC: Connected Limited Device Configuration
CWML: Compact Wireless Markup Language.
DIS/HLA: Distributed Interactive Simulation/High Level Architecture.
EDGE: Enhanced Data GSM Environment : débit maximal 384 Kb/s.
FPS: First Person Shooter.
GASP: Gaming Services Plateforme.
GDC: Game Developer Conference.
GET: Groupe des Ecoles des Télécommunications.
GPRS: General Packet Radio Service.
GSM: Global System for Mobile Communications : débit maximal 14,4 Kb/s .
HTTP: HyperText Transport Protocol
IEEE: Institute of Electrical and Electronic Engineers.
IP: Internet Protocol.
IRC: Internet Relay Chat.
JDR: Jeu de rôle.
Kb: Kilo bit.
KB: Kilo Byte.
Mb: Mega bit.
MEGA: Mobile multi-playEr online Game Architecture.
MIDP: Mobile Information Device Profile
MMORPG: Mass Multiplayer On-line RPG.
NTP: Network Time Protocol.
OMA: Open Mobile Alliance
OTA: Over The Air.
P2P: Peer To Peer.
PDA: Personal Digital Assistant.
RPG: Pole Playing Game.
RTCP: Real Time Protocol Control.
RTI: Run-Time Infrastructure.
RTP: Real Time Protocol.
RTS: Real Time Strategy.
UDP: User Datagram Protocol.
UMTS: Universal Mobile Telecommunications Services, ou 3G : maximal 2 Mbit/s.
WAP: Wireless Application Protocol.

Termes :

Avatar: Par définition (Larousse): “Descente sur la terre d’un être divin || Nom générique des incarnations divines,… “. Il s’agit ici de la “descente dans le monde virtuel d’un être humain“. C’est le nom qu’on donne aux incarnations virtuelles des joueurs.
BackTracking: Annulation des derniers calculs (pour appliquer des corrections).

Bluetooth: Nouvelle norme de communication OTA
Game Boy: Console de jeu NINTENDO portable.
Gameplay: Qualité de la communication homme-machine et plaisir de jeu.
JAVA : langage de programmation portable de Sun Microsystems.
i-Mode : Service de téléphonie mobile de la société japonaise NTT DoCoMo concurrent du WAP et fondé sur CWML et non sur WML.
Jouabilité: traduction de gameplay.
Lag : temps de latence.
N-Gage: Console de jeu NOKIA portable
Routage : manière de transporter les données.

Au Japon la sortie d'une nouvelle console de jeux est un événement. Le jour où Nintendo sortit sa ‘Game Boy advance’, il s’est vendu plus de 500 000 exemplaires en une journée. Les 'millions sellers' sont maintenant nombreux (Warcraft, Tomb Raider ...) et certains jeux représentent à eux seuls 700 millions de dollars de chiffre d'affaires.[8]

Le phénomène s'implante plus lentement en Europe où il y a moins de joueurs et où l’accès (rapide) au réseau Internet ou autre n’est que récent (en France L’ADSL n’est disponible que depuis 4 ans). Aujourd'hui, presque tout jeu se doit de posséder un mode multijoueur. Ce mode multijoueur nécessite un réseau (Internet la plupart du temps). Jusqu'à présent, les consoles portables, privées d'accès à Internet, ne permettaient que des modes multijoueurs via un câble reliant les consoles, des écrans divisés ou encore des jeux permettant à tous de s’affronter sur un même écran (Bomberman®, Jeux de Sport). Ces dernières années, les consoles s’équipent de modem.

Aujourd'hui les PDA, portables Centrino® et autres technologies sans fil montrent que l'on peut accéder au réseau mondial via des postes mobiles. Les jeux sur équipements mobiles devraient donc bénéficier d’un réseau de plus en plus rapide et vaste, et les fans de jeux en ligne pourront sûrement s'adonner à leur passion d'ici quelques années. En effet, les réseaux actuels ne sont pas assez rapides et n'offrent que des ‘ping’ proches de la seconde sur réseau GPRS.

Ce stage s’inscrit dans le cadre du projet MEGA (Mobile multi-playEr online Game Architecture), financé par le GET, (Groupe des Ecoles des Télécommunications) qui a pour objectif d’identifier les verrous technologiques et/ou scientifiques qui limitent les jeux sur mobiles (aspects techniques) et les usages et pratiques sociales liés à ce nouveau support (aspects sociologiques). Nous commencerons par présenter le jeu multijoueur, puis nous présenterons les techniques de synchronisation existantes pour enfin proposer un jeu en réseau multijoueur qui utilise HTTP pour communiquer via GPRS.

1.Le Jeu Multijoueur

Les applications distribuées, telles que le jeu multijoueur, ont des besoins variés fonction de leurs sémantiques. Avant d'aborder les solutions existantes aux problèmes de synchronisation dans la partie suivante, nous décrirons les types de jeux multijoueurs qui existent et les moyens mis en œuvre pour leurs réalisations.

1.1 Types de jeu

Le jeu vidéo est maintenant considéré par certains comme une forme d'art. Les contributions artistiques sont, en effet, nombreuses. Les budgets grandissent énormément pour certaines productions. Il n'est pas vraiment possible de classer les jeux. Nous présentons donc ici les différents types d'une manière arbitraire, en commençant par ceux qui font l'objet de compétitions internationales : les jeux de stratégie, les FPS et les jeux de combat.

1.1.1 Jeux de stratégie

Les jeux de stratégie et les RTS sont des jeux où un joueur dirige de nombreuses unités dotées de capacité spécifique sur le champ de bataille. Deux critères permettent de les classer : Ceux qui ont des ressources gérées par le joueur ou non, ceux en temps réel ou au tour par tour. Pour mieux comprendre, nous présentons quelques exemples :

- Medieval Total War®[1] est un jeu temps réel à la stratégie très élaborée (Gestion du relief, du moral ..) et où les ressources ne sont pas gérées. On ne crée donc pas de nouvelles unités soi-même (il se peut que les développeurs prévoient l'arrivée de renforts). On peut remarquer le nombre important de celles-ci mais elles sont regroupées par bataillon (si cela est possible). C'est une simulation de général d'armée.
- Cossacks®[2] lui aussi temps réel, permet de construire soi-même des unités : pour ce faire, un schéma classique d'utilisation des ressources dans les jeux RTS est mis en place. Premièrement l'on doit construire un bâtiment d'extraction (à placer sur le champ de bataille) et un de stockage (dans certains jeux, cela nécessite des unités particulières) puis, si l'on a assez de ressources, on peut construire d'autres bâtiments permettant de construire plus d'unités et plus de bâtiments. La figure 1 présente l'exemple de Starcraft des studios Blizzard.
- Panzer General®[3] est une série réputée pour une intelligence artificielle de qualité mais aussi pour son niveau de précision (le carburant, les munitions, le moral, le terrain, rien n'est laissé de

côté). C'est un wargame classique qui se joue au tour par tour. Chaque joueur choisit ses déplacements et ses autres actions de jeu à tour de rôle.

Ces jeux gèrent beaucoup de données de manière très dynamique. L*e tour par tour* est par

Figure 1 – Starcraft™ de Blizzard

nature synchronisé. Même si les jeux de stratégie au tour par tour sont en perte de vitesse, ils pourraient revenir à la mode sur des machines portables.

1.1.2 First Person Shooter

Aussi appelés ‘Kill’em All’ (tuons les tous), ces jeux sont basés sur les réflexes et nécessitent pour jouer en réseau, à l’heure actuelle, des ‘ping’ bas (100ms) et équitables pour une bonne jouabilité. Ces types de jeu sont basés sur les ‘System Development Kit’ de Unreal tournament® , Quake 3 arena®[4] ou Half Life®[5] (la version multi joueur est le célèbre Counter Strike®) pour les versions commerciales.

Figure 2 - DOOM sur Game Boy Advance

Les interactions avec l’environnement sont peu nombreuses: on peut ouvrir une porte, déplacer un ou deux objets mais rarement plus. Par contre, il est très important de connaître la position exacte des joueurs (humains ou non). Les versions actuelles proposent des FPS gérant jusqu'à 64 joueurs simultanément (150 en théorie) via des serveurs dédiés disposant d’une adresse IP Internet et d’une connexion rapide [42]. Il est nécessaire de posséder une connexion supérieure à 512/128 Kb par seconde (montant/descendant) pour absorber le flux de données de plus de 32 joueurs.

La base de ces jeux est simple : on incarne un guerrier via une vue subjective (à la première personne) et l’on peut se mouvoir à loisir, récupérer des objets (armes et munitions en général), tirer... Chacun propose plusieurs types de jeux en réseaux :

- *Capture The Flag*: deux équipes s’affrontent, il faut aller chercher un drapeau et le ramener près du sien.
- *Objectif* (depuis *Counter Strike*) : Deux équipes s’affrontent, l’une devant empêcher l’autre de remplir sa mission. Les objectifs deviennent plus variés et plus complexes dans les derniers jeux comme Enemy Territory.
- *Free For All* : Chacun pour soi.
- *FreezMode* : Deux équipes s’affrontent. Si vous êtes touché par un adversaire, vous devenez incapable de faire la moindre action jusqu'à ce qu’un coéquipier reste plus de 10 secondes à côté de vous. La première équipe qui ‘freez’ tous les joueurs de l’autre équipe marque un point.
- *Last Man standing* : Comme le Free for all mais on ne revient pas dans la bataille avant que le vainqueur de la manche précédente ne soit désigné.
- *Chasse à l’homme* : L’un des joueurs est la cible de tous les autres; celui qui l’attrape, le tue (le ‘frag’) et devient le pourchassé…

Certains modes de jeux sont développés par la communauté de joueurs (Système de ‘Mods’) comme par exemple *orange smoothie production* [63] pour le multijoueur ou *True Combat* [http://www.truecombat.co.uk/home/] une conversion de quake3. Ils sont donc nombreux et variés.

1.1.3 Jeux de Rôle

Figure 3 - JDR, phase stratégie au tour par tour.

Les jeux de rôle (JDR-RPG en anglais) sont souvent basés sur le moteur de jeu d20 system[7] et sont directement issus de leur déclinaison non informatique. Dans ces jeux, l'on doit incarner un personnage fictif, qui le plus souvent part à l'aventure, pour sauver le monde, se faire de l'argent ou pour n'importe quelles autres motivations suffisantes. Un 'maître du jeu' est présent et doit aider à résoudre les problèmes de règle et décrit toutes les actions (il est, en quelque sorte, l'auteur final du scénario du jeu). Ultima Online® est un bon exemple de monde persistant massivement multijoueur (MMRPG), assez poussé graphiquement : Le monde du jeu, accessible par zone, est complètement représenté. On peut y voir des arbres, des grottes, des villages, etc… De nombreux avatars, représentant les joueurs humains, peuvent apparaître simultanément sur le même écran, chacun ayant des interactions spécifiques. On ne peut ignorer *NeverWinter Nights(Figure 4))* et *Baldur's Gate* qui utilisent les règles officielles de Advanced Dungeon & Dragon ® basées sur les nouvelles de Tolkien.

Certains jeux de rôle prennent la forme d'une simple liste de destinataires, parfois agrémentée d'une interface pour gérer certains aspects du personnage incarné. Ils peuvent aussi se décliner sous la forme de FPS (Deux Ex®) et les interactions avec l'environnement sont innombrables (on peut déposer des objets, en prendre, en détruire, en transformer, modifier des lieux ..). La taille de ces objets est aussi importante (chacun ayant de nombreuses caractéristiques), et la quantité de règles à gérer peut être énorme (voir le d20 system). Malgré cela, la plupart sont en tour par tour (imperceptible, si l'on n'attend pas à la fin de chaque tour) et surtout régis par de très nombreuses règles à usage précis. L'un des caractères spécifiques des jeux de rôle est que l'avatar du joueur atteint des niveaux lui permettant d'accéder à de nouveaux pouvoirs.

Figure 4 - Screenshot Neverwinter Nights

1.1.4 Jeux de Combat

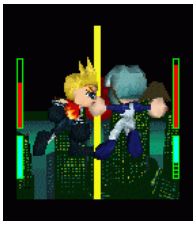


Figure 5 - Jeu de combat sur portable.

Les Jeux de Combat (fighting game) sont des jeux à 1 ou 2 joueurs très célèbres en salle d'arcade ou sur console. Chaque personne joue un guerrier capable de donner des coups et de se protéger. Le but est de mettre KO son adversaire et/ou hors du ring. La plupart de ces jeux sont basés sur des enchaînements de boutons précis qui déclenchent une action précise. Certaines actions s'enchaînent mieux que d'autres et sont appelées combo. Leur niveau de réalisme est varié. Dans Bushido Blade®, si l'on est blessé au bras, plus question de s'en servir. Beaucoup sont très orientés action comme Soul Calibur®. Ce type de jeu est basé sur les réflexes et la quantité de données à gérer est minime. Comme les FPS, le jeu est en temps réel et les actions prises par le joueur dépendent de la situation actuelle : le temps de latence est donc un facteur critique (et doit être inférieur à 200 ms).

1.1.5 Action, Aventure

Les jeux d'aventures 'pure' n'ont pas de déclinaison multijoueur, ce sont des suites d'énigmes (de mini puzzle game) à résoudre telle l'ancienne série de type 'point and click' Indiana Jones® de LucasArts ou encore les jeux de plates-formes comme Les Sables du Temps®. Leurs déclinaisons multijoueurs se rapprochent des jeux de rôle.

La catégorie action est assez vaste. Un jeu d'action est en temps réel. Un jeu d'aventure à la première personne (Deus Ex®) est un jeu de type Action/Aventure et de genre FPS.

Dans cette catégorie, Action, on peut aussi ajouter les jeux sportifs tels les Jeux Olympiques où l'on doit appuyer à répétition sur un bouton pour faire avancer un athlète ou les 'shoot'em up' où il faut éradiquer la menace dans un petit vaisseau spatial surarmé [9].

1.1.6 Puzzle

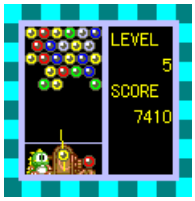


Figure 6

Tétris® est l'un des jeux les plus joués au monde. Simple et abordable il a même une déclinaison à 2 joueurs. Ses clones sont nombreux (la série des Bubble® voir Figure 6.). D'autres jeux sont aussi basés sur la réflexion et la mise en place de petits blocs ou leur destruction permettant de marquer des points. 'Sokoban' et 'Stones' appartiennent à cette catégorie. Citons aussi les lemmings® où l'on doit créer une route en donnant des ordres basiques (creuse, prends ton parachute si tu tombes de haut, construis une échelle…), pour sauver ces petites bêtes.

1.1.7 Simulation

Figure 7 - Simulation de train sur téléphone portable

Quel plaisir de dépasser des vitesses folles, de piloter un avion de chasse ou un train ! Tout véhicule ou presque a aujourd'hui sa simulation : Need For Speed®, Midtown Madness® (voiture), EF2000®, Flight Simulator® (avion), Train Simulator® [6] …. Proche des FPS au niveau des données gérées, l'interaction utilisateur->environnement (la route) est quasi inexistante (le monde n'est pas ou très peu déformable). La vitesse de ces véhicules est une caractéristique importante, ainsi que la taille de l'environnement où ils évoluent. Dans Need For Speed, on suit un circuit, alors que dans Mitdown Madness on se déplace dans une ville. Les simulateurs d'avions ont des environnements beaucoup plus larges (New york->Paris). Tous ces jeux sont déclinables en multijoueur.

1.1.8 Jeux de Gestion

Figure 8 - Sim City sur téléphone portable

L'aspect gestion de ressources des jeux de stratégie est ici porté à son paroxysme. Dans la série des Sim City, on est maire d'une ville et l'on organise le territoire. La version multijoueur permet de dialoguer avec ses voisins pour passer des contrats (à propos d'import/export par exemple) et l'on peut attirer les citoyens d'un maire peu éclairé : l'échange de données est quasi nul. Space Colony® permet de construire une colonie spatiale où chaque habitant est représenté avec ses envies et ses désirs qu'il faut combler (ils ne travaillent que s'ils sont assez satisfaits). D'autres colonies peuvent être créées sur d'autres planètes permettant des échanges du même type que dans Sim City. Par contre, dans un jeu tel que Transport Tycoon® ou Capitalisme® (Simulation de directeur d'entreprise) le monde est partagé par les (impitoyables) concurrents. Dans Space Colony® une autre colonie peut ainsi s'installer sur la même planète. Le jeu devient alors un RTS, les colonies sont en concurrence pour les ressources disponibles et elles peuvent s'attaquer. Les besoins en vitesse de communication augmentent donc et sont proches des FPS qui demandent des temps de latence très bas. Les entités présentes, telles que les bâtiments ou les ouvriers, ne sont pas directement contrôlées par l'humain. Le joueur donne des ordres qui modifient leurs comportements. Ce caractère permet à ces jeux de ne pas être plus gourmands en bande passante qu'un FPS alors que la taille des données à gérer peut vite exploser. Cossacks®[2] gère plus de 10 000 entités partagées en temps réel, alors que pour le même réseau,

on ne pourra jouer qu'à 32 sur un FPS (pas plus de cinquante entités partagées mais 32 gérées directement par le joueur).

1.1.9 Sport

Figure 9 - Jeu de Base Ball sur portable

La plupart des jeux de sport individuel sont des jeux d'arcade comme Tony Hawk's Pro Skater® (jeu de skate), Virtual Tennis® ou SSX Super Tricky® (jeu de snow-board). Chacun a une version multijoueur en écran divisé pour les jeux de planches.

Les jeux de course sont assez nombreux et les véhicules variés. Pour les jeux d'équipes, citons ISS Pro Evolution Soccer 3® et les NBA2K®, NHL2K ® qui sont tous des simulations de respectivement foot, basket, hockey.

La précision et le niveau de réalisme de ces jeux agissent directement sur leur complexité. L'aspect multijoueur de ces jeux est naturel : on devient un sportif virtuel en concurrence avec les autres. Comme les FPS, certains sports ont besoin de temps de latence réduits (les sports d'équipe à ballon par exemple). Par contre, un jeu de base-ball pourra plus facilement intégrer des méthodes pour rendre un temps de latence élevé transparent. En effet, le jeu peut se séparer en phases de mini-jeux d'adresse : lancer, frapper (à la batte, Figure 9), réceptionner plus vite que l'adversaire ne parcourt une base si la balle est renvoyée par le batteur.

1.2 Jouer en Réseau

Il existe de nombreux jeux multijoueurs. Pour pouvoir participer, le joueur doit avoir une machine reliée par un réseau aux autres machines des joueurs. Nous présentons ici, rapidement, les différents moyens mis en oeuvre. Proposer des services identiques sur poste mobile et sur poste fixe est un défi. Les réseaux mobiles sont moins fiables, plus coûteux et souvent plus lents.

1.2.1 Accès à un réseau

La première solution pour jouer à plusieurs est de demander à des amis de se retrouver pour organiser un réseau local et donc privé. Totalement gratuit si l'on est déjà équipé du matériel nécessaire : Un hub (une multiprise pour le réseau) si l'on est nombreux et des interfaces réseau sur les machines. Les consoles ont souvent ce matériel prêt à l'emploi. La N-Gage dispose par

exemple d'une interface Bluetooth et les Game Boy se relient par câbles. Le nombre de joueurs est alors restreint.

La deuxième solution est de joindre le réseau public Internet. On peut alors rencontrer les joueurs du monde entier. De nombreuses applications [33] permettent de connaître les adresses (IP) des serveurs de jeu actifs et les joueurs présents. Les serveurs sont soit maintenus par le producteur du jeu [34], soit par la communauté de joueurs [32][42]. Les joueurs communiquent avec Internet via des applications spécifiques telles que IRC, des sites Internet de type forum ou encore TeamSpeak [51] et Ventrilo [52] pour communiquer en vocal.

Le rapport [62] décrit les solutions mises en œuvre pour proposer des services de jeux multijoueurs sur des postes mobiles ayant accès aux réseaux OTA.

1.2.2 Architecture réseau

En fonction des spécifications de l'application, et de l'argent disponible, plusieurs choix peuvent être faits. L'architecture Client/Serveur est en général préférée au P2P pour des raisons de sécurité. La présence du serveur permet un enregistrement externe de certaines données et peut servir d'arbitre dans les processus de synchronisation [47]. Le problème est qu'alors le serveur doit être capable de communiquer avec tous les clients simultanément. Ainsi un serveur de FPS pour 12 joueurs aura besoin d'au moins 2Mb/s de bande passante montante et descendante (au moins 128 Kb/s côté client). Un serveur de jeu de rôle est souvent un cluster de plusieurs machines parfois aidées de Proxy (Figure 10). Les Proxy sont des répliquas du serveur rendant les données plus rapidement accessibles. Utiliser plusieurs serveurs permet de couvrir des zones géographiques plus larges avec un temps de latence réduit [24] et d'accueillir plus de joueurs [14]. On approche alors de solutions mixtes [45] où certains clients jouent le rôle de Proxys ou de serveur. Les solutions purement P2P sont rares [35].

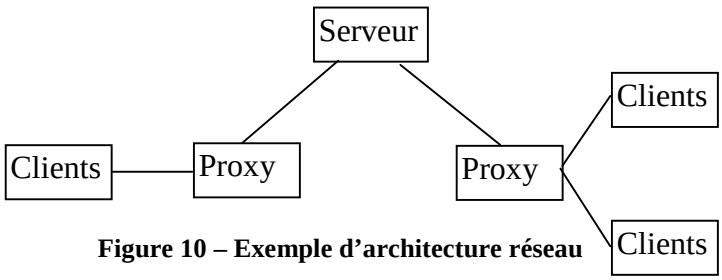


Figure 10 – Exemple d'architecture réseau

Les nouveaux dispositifs de communication OTA demandent plus de souplesse. Le rapport d'Eugeniusz Hetmanski [18] décrit les architectures réseaux existantes et propose une solution pour le jeu sur poste mobile.

1.3 Jeux et Calcul.

Le client d'un jeu est un morceau d'une application partagée. Il peut participer au calcul d'un environnement partagé. Il doit donc communiquer et aussi représenter l'environnement calculé. Nous proposons une architecture logicielle basique pour mieux définir les différents processus relatifs à la synchronisation nécessaire à l'application jeu multijoueur. Ensuite nous expliquerons pourquoi un jeu peut utiliser des techniques génériques de synchronisation, puis nous finirons par présenter les limitations du monde mobile.

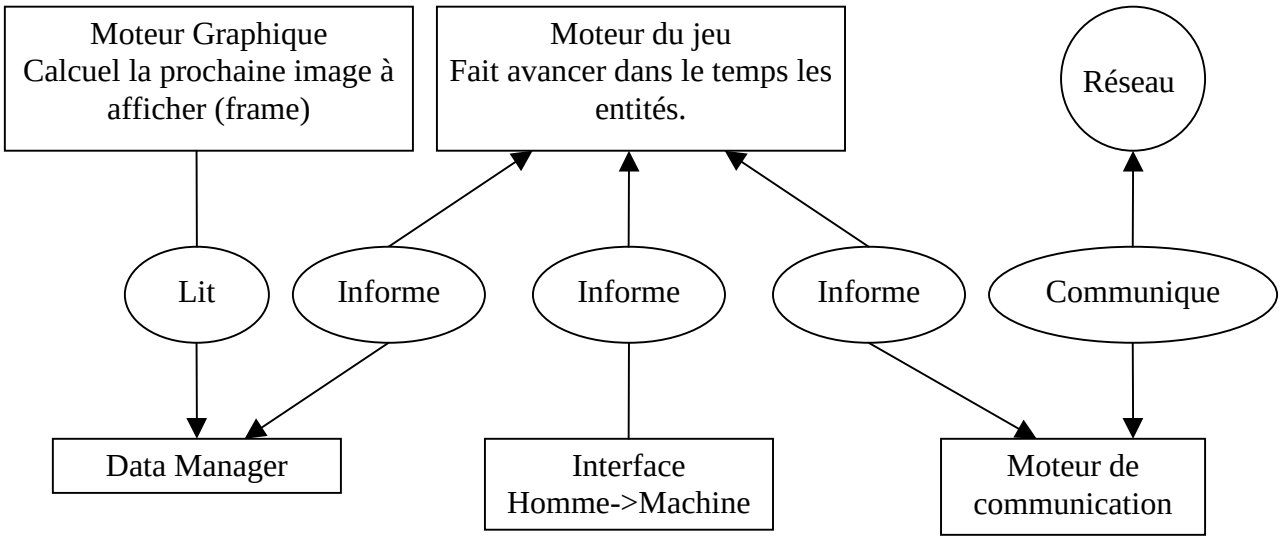


Figure 11 - Architecture Logicielle basique pour jeu d'action multijoueur

1.3.1 Architecture logicielle basique.

Le moteur du jeu s'informe des actions demandées par l'utilisateur et des informations reçues du réseau. Ensuite il met à jour les données et envoie aux autres participants des informations sur l'état des entités gérées par le programme. Le moteur Graphique affiche l'état suivant selon les spécifications du jeu, soit par une prédiction à partir de l'état actuel, soit l'état actuel ou un mariage des deux : Chaque type d'entité peut implémenter ses propres méthodes et l' 'état affiché' peut être une union entre une prédiction, le dernier état valide connu, et même une extrapolation (destinée par exemple à garantir une cohérence au niveau graphique). Un jeu au tour par tour n'a pas besoin d'extrapolation ou de prédiction. Bien séparer la partie moteur Graphique peut permettre de garder une bonne jouabilité malgré des temps de latence élevés. Le moteur de communication peut s'occuper d'ordonner les messages reçus ou de faire une partie du routage. Certains moteurs

proposent de placer les données à envoyer dans des files pour avoir un meilleur flux de communication. Les moteurs de communication génériques sont nombreux [16].

Les services de synchronisation d'horloges logiques sont liés au moteur de communication. On peut par exemple ne distribuer que des messages valides : qui se sont produits dans le passé. Par contre, d'autres techniques de synchronisation s'implémentent à d'autres niveaux : Le Data Manager peut garder une trace des modifications de données pour permettre des corrections ultérieures. Le Moteur du jeu peut implémenter des comportements spécifiques pour certaines données. Le calcul de l'affichage peut utiliser des techniques pour augmenter la cohérence visuelle de l'action : on affiche une représentation locale qui n'est pas forcément la 'plus juste réalité' possible.

1.3.2 Le jeu : un système causal.

Tous les jeux représentent un monde régi par des règles. De plus les jeux sont souvent basés sur une boucle discrète de gestion d'évènement. Comme dans les modélisations mathématiques de la physique, les lois ou règles sont simulées : les jeux sont des simulations [55]. Elles respectent donc le principe de causalité. Le futur n'influence pas le passé. Ainsi les recherches sur la synchronisation de processus peuvent être utilisées pour le jeu [48]. Ces techniques garantiront une bonne exécution des programmes distribués.

1.3.3 Limitation du monde mobile

Sur téléphone mobile et autres machines portables les limitations de l'interface homme machine freinent le développement de jeux [10]. Les illustrations ci-dessus viennent de jeux sur mobile ou sur console portable [11] excepté pour *Starcraft*, *et Neverwinter Nights*. Les capacités des machines portables augmentent (tableau 1).

Au Japon [11] le secteur est beaucoup plus actif qu'en Europe : il existe même des magazines dédiés aux jeux sur mobile. Les opérateurs essayent de proposer un meilleur accès sans fil à leurs clients et de nombreuses bornes de communication sont installées.

Les coûts de développement de jeux sur mobiles sont réduits [13] mais l'implémentation peut devenir complexe à cause de la multiplicité des plateformes [12]. Les opérateurs américains sont plus orientés sur BREW [37], une plateforme C++, alors qu'en Europe [19] les systèmes (DoJa, I Mode ..) utilisent plutôt une plateforme JAVA, parfois au-dessus du système d'exploitation Symbian qui propose des fonctions systèmes [38] en langage C.

Si la navigation Internet se démocratise sur plateforme mobile, on pourra espérer une totale portabilité via un navigateur Internet. Ainsi des applications développées en 'flash' [40] telles que

http://www.shockwave.com/sw/content/capoeirafighter, plus faciles à déployer, pourraient représenter la majeure partie des jeux à venir sur téléphone portable [39]. A l'heure actuelle, seul le protocole HTTP est toléré par les opérateurs (en France). De plus, les coûts actuels de transport d'informations sont élevés.

Matériel	Processeur	Fréquence	Résolution
GP32	ARM	max. 99Mhz	320x240
Nokia N-Gage	ARM	~100Mhz	176x208
Nokia 3650/7650	ARM	~100Mhz	176x208
Orange SPV e100	ARM	~132Mhz	176x220
Palm Tungsten-T	ARM	~175Mhz	320x320
Sony P800	ARM	~200Mhz	240x320
Compaq 3600 series	StrongARM	~200Mhz	240x320
HP HP1915	XScale	~200Mhz	240x320
Sony CLIE PEG-NZ90	StrongARM	~200Mhz	320x480
Tapwave Zodiac	ARM	~200Mhz	480x320
Asus A620	XScale	~400Mhz	240x320

Tableau 1 - Capacité actuelle des équipements portables.

1.4 Besoin

Une exécution sans erreur n'est pas le but d'un jeu vidéo. Contrairement à une expérience physique où l'on s'intéresse aux résultats et à la justesse du déroulement, le développeur doit avant tout s'intéresser à la jouablilité : c'est-à-dire à la manière dont le joueur sera capable d'apprécier son jeu.

1.4.1 Données

Les types de données gérés par un jeu sont nombreux. Chaque jeu a des besoins spécifiques. Une entité est un élément, représenté dans le jeu, qui a des interactions avec la simulation. Par exemple : un joueur, un coffre, une maison, un projectile ou encore une brique. Les jeux gérant le plus d'entités sont les RPG massivement multijoueurs, suivis des wargames puis des jeux d'aventure/action. En bas de cette échelle, on retrouve les jeux d'arcade et les 'puzzles games'. Un

jeu d'échec par exemple est un 'puzzle games' qui gère 32 pièces sur 64 cases (quantité de données très faible). Par contre, un algorithme pour simuler un joueur est extrêmement complexe. De même, les besoins en communication ne sont pas uniquement en rapport avec la quantité de données à traiter mais aussi fonction de l'interface homme-machine choisie. Les entités du jeu doivent être mises à jour en fonction de tous les joueurs. C'est donc une application distribuée qui nécessite un protocole de communication spécifique. Les réseaux publics sont souvent chargés, peu fiables et très hétérogènes. La communication sera donc un élément clef de la réussite du jeu [12][13].

De [14], on peut déduire trois grandes classes de données : les données locales, les données distantes, et les données partagées (tolérant quelques incohérences). Les données locales peuvent par exemple être en rapport avec l'affichage et sont indépendantes des autres utilisateurs. Les données distantes sont protégées sur un serveur, elles ne sont lisibles que via des procédures d'appels distants permettant l'absence totale d'erreur : On peut penser à un trésor dans un coffre ou à des données stratégiques telles que le nom d'utilisateur. Ces données auront le désavantage d'être sur le serveur et leur utilisation dans le jeu sera donc différente et totalement soumise au temps de latence. Enfin les données partagées telles la position, ou certaines caractéristiques des entités qui pourront tolérer des erreurs de courte durée au cours du jeu. Ces dernières ont besoin d'algorithmes de synchronisation spécifiques. Leur quantité et leur complexité influeront directement sur la bande passante requise alors que l'interface déterminera le temps de latence maximum requis.

Si l'on veut avoir X états distribués valides par seconde avec un jeu d'action où chaque joueur (ils sont N) gère une entité directement (de manière continue) de taille E, et qu'il existe 3 portes (3 entités partagées de taille 1 (ouvert/fermée) alors le joueur doit envoyer entre X*E et X*(E+1) octets par seconde et recevoir au pire N*X*(E+3) octets par seconde [36].

1.4.2 Cohérence

Pour des raisons de calcul, il est essentiel de maintenir une cohérence entre les données du monde partagé par les joueurs. Mais pour des raisons de jouabilité, il faut garder une vision plausible et cohérente au niveau du rendu pour l'utilisateur. Le joueur doit être persuadé d'interagir avec les autres joueurs de façon équitable et réaliste.

Plus le jeu est de type action et basé sur les réflexes comme les FPS ou les 'fighting games', plus le temps de latence requis est bas [17]. En effet, le joueur prend beaucoup de décisions en de très courts laps de temps et ses décisions sont fonction de l'état actuel du jeu. S'il est trompé par un système qui tolère trop de fautes, il prendra des décisions sans rapport avec l'action : le jeu n'aurait donc plus de sens. D'après [41], 150 ms est le temps de latence requis pour un jeu type FPS et 500ms pour un jeu de type RTS d'après [35]. Un temps de latence trop long est très mal perçu par

les joueurs qui n'ont plus l'impression d'interagir avec le monde partagé. Bien entendu, cela n'est valable que pour des applications temps réel (FPS, RTS, Simulation). Un jeu au tour par tour (Heroes of Might & Magic) ne souffre pas du temps de latence en terme de cohérence entre les données. Par contre, si l'utilisateur n'est pas patient, l'attente entre chaque tour pourra le rebuter.

L'un des exemples de techniques destinées à garder une cohérence dans le programme est l'utilisation de représentation spécifique. Par exemple : si le temps de latence devient très grand (similaire à une déconnexion), le joueur n'est plus actif. On affiche donc en lieu et place de sa dernière position connue un avatar, rouge ou clignotant, permettant aux autres joueur d'identifier que le joueur n'est plus présent ou qu'il n'a pas la capacité d'interagir avec le monde.

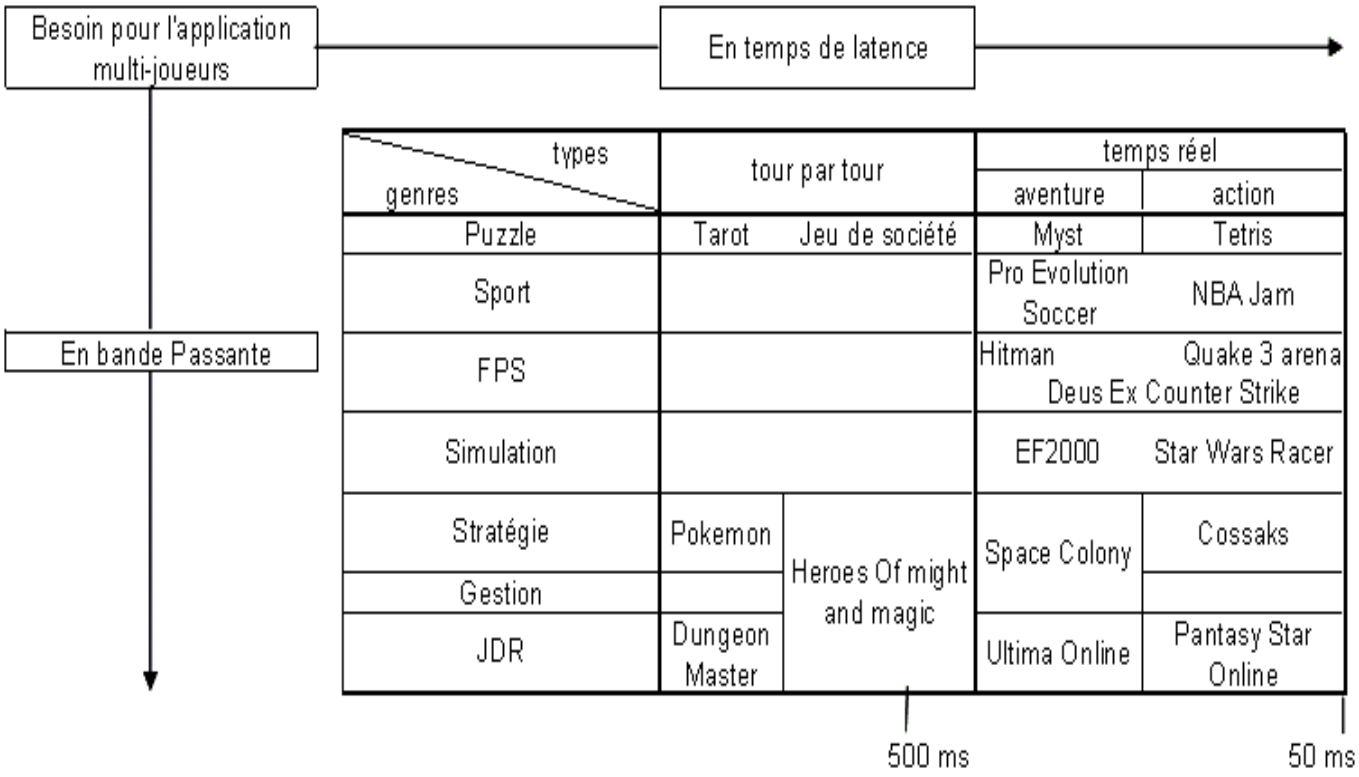
Comme vu dans la section 1.4.1, toutes les entités ne nécessitent pas une forte synchronisation avec leurs représentations distantes chez les autres joueurs. Si celles-ci tolèrent des fautes, il faudra définir le temps durant lequel elles peuvent diverger de l'état réel et/ou dans quelle mesure. Si ces conditions ne sont pas remplies, il faudra bloquer le jeu et faire une pause dans l'application temps réel ou perdre de la cohérence et prévoir des réconciliations.

La synchronisation totale des événements n'est possible que si l'on affiche un état antérieur ou, ce qui est équivalent, si l'on retarde le message : après son envoi, on attend son arrivée auprès de chaque client du jeu pour que chaque client le traite en même temps.

Si l'on utilise des communications de type non fiable, comme avec le protocole UDP, pour certaines entités du jeu, il faudra garantir la cohérence de celles-ci en détectant les messages perdus ou en corrigeant leurs valeurs de façon cyclique. Le dernier état est par essence faux. En effet, tout message émis est reçu en retard. Il manque donc les derniers instants de communication et la mise à jour n'est jamais complète. On peut utiliser plusieurs techniques pour prédire l'état supposé d'une entité par rapport à un état valide antérieur dont le 'Dead Reckoning' qui dépend de l'évaluation du temps de latence. Cette technique sert principalement à la gestion des collisions entre entités partagées, elle peut aussi être utilisée par le moteur d'affichage pour garantir une cohérence visuelle entre les images du jeu (frames).

1.6 Conclusion

Les jeux d'action ont besoin de temps de latence bas pour gérer des interactions (Tableau 2). Le nombre de joueurs influe linéairement sur le besoin en bande passante qui est défini par la somme des tailles des entités partagées. Les capacités actuelles des réseaux mobiles ne sont pas suffisantes pour



types / genres	tour par tour		temps réel	
			aventure	action
Puzzle	Tarot	Jeu de société	Myst	Tetris
Sport			Pro Evolution Soccer	NBA Jam
FPS			Hitman	Quake 3 arena, Deus Ex, Counter Strike
Simulation			EF2000	Star Wars Racer
Stratégie	Pokemon	Heroes Of might and magic	Space Colony	Cossaks
Gestion				
JDR	Dungeon Master		Ultima Online	Pantasy Star Online

Tableau 2 - Besoin des jeux

proposer autant de services que sur réseau fixe. L'un des problèmes majeurs est le temps de latence important sur les réseaux sans fil 2G/3G. Une technologie récente telle que Bluetooth est plus rapide mais son rayon d'action est restreint : 10 mètres.

Il faut bien analyser les spécificités du jeu avant d'évaluer les besoins en communication pour celui-ci. Par exemple le jeu Magic™, se joue avec des cartes autour d'une table comme la belotte mais certaines cartes peuvent être jouées en dehors de son tour de jeu en réponse à une action de son adversaire. Le jeu est donc au tour par tour mais aura besoin d'un temps de latence plus réduit qu'un jeu de tarot. A chaque carte jouée (on peut en jouer plusieurs par tour), il faudra attendre l'éventuelle réaction du joueur adverse qui peut annuler les effets de la carte. La demande en bande passante dépend des entités partagées, au pire, elle est égale à la somme des tailles des valeurs partagées multipliée par le nombre de mises à jour par seconde. Le cinéma utilise 24 images par seconde, la télévision 25. Nous pouvons donc penser qu'une application temps réel aura besoin d'au moins 20 mises à jour par seconde.

2 Synchronisation

Nous présentons tout d'abord les techniques de synchronisation généralistes, puis les techniques plus spécifiquement utilisées dans les jeux. L'objectif de la synchronisation est de garantir que l'exécution distribuée des programmes permet l'évolution du monde partagé.

2.1 Optimiste ou pessimiste

Il existe deux manières d'aborder la synchronisation : la manière optimiste qui laisse aux programmes clients le droit de diverger (d'avoir des états différents au cours de l'exécution) et qui prévoit des réconciliations pour faire converger vers le même état final les valeurs distribuées. La manière pessimiste qui cherche à éviter toutes formes d'erreur dans le déroulement de l'application. Celle-ci nécessite le plus souvent des méthodes de verrou comparables aux sémaphores et aux mutex qui bloquent l'application. Enfin, les applications peuvent choisir d'envoyer des commandes ou des états en fonction de leurs besoins. Si l'on envoie des états, les algorithmes nécessaires sont peu complexes, alors que si le programme envoie des commandes, on permet plus de flexibilité [54].

2.1.1 Synchronisation de processus

Un jeu peut être considéré comme un système physique ayant ses propres règles. Les systèmes physiques obéissent toujours au principe de causalité. Celui-ci peut être énoncé de la façon suivante : le futur n'influence jamais le passé. La causalité impose donc un ordre des transitions d'états dans les systèmes physiques. Lamport [30] a défini une méthode basée sur l'utilisation d'estampilles pour permettre aux processus de respecter la causalité. Pour garantir que les transitions entre états de jeu soient correctes et pour éviter l'effet domino qui propagerait une erreur à travers l'exécution, l'ordre dans lequel le simulateur produit les événements doit être cohérent pour que le jeu soit un reflet juste de la partie jouée. Par exemple, si la transition A du système physique intervient à la date 3 et a une influence sur la transition B intervenant à la date 5, le simulateur doit générer l'événement modélisant la transition A avant celui concernant la transition B.

Le problème de l'ordre se pose aussi en transmission vidéo où si l'on reçoit les paquets dans le mauvais ordre, l'image sera faussée (en mode streaming). Deux protocoles sont utilisés pour pallier ce problème: RTP [53] pour l'ordre et RTCP qui permet de connaître les délais de routage.

2.1.2 Synchronisation d'horloge

Synchroniser deux horloges est similaire au problème du calcul du temps de latence et est indécidable :

Soit t et t' deux horloges de valeur différente, sur deux machines distantes reliées par un réseau quelconque (fig.12).

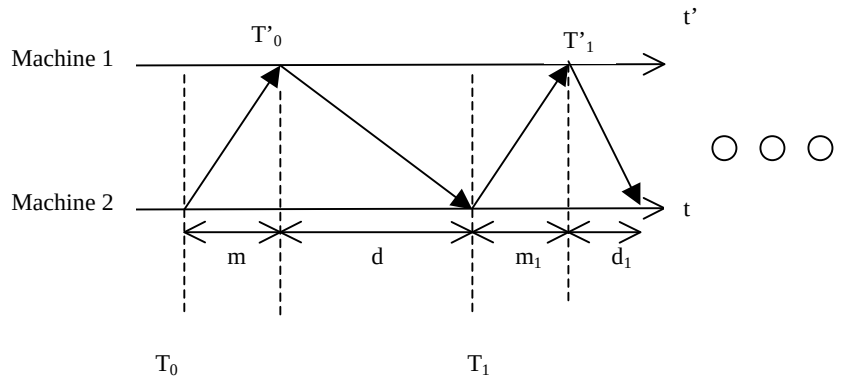


Figure 0 - Schéma de communication entre deux processus distants

On a : $m+d=T_1-T_0$

$d+m_1= T'_1-T'_0$

$m_1+d_1=T_2-T_0$

D'où on peut logiquement déduire :

$m-m_1 = (T_1-T_0)-(T'_1-T'_0) = K_m$ et

$d-d_1=(T'_1-T'_0)-(T_2-T_0) = K_d$.

Ce qui n'est que rassurant. Si l'on rajoute une équation, telle que $d_1+m_2=T'_2-T'_1$, on rajoute une inconnue et il reste donc toujours un paramètre de plus que le nombre d'équations linéairement indépendantes.

Or m>0, d>0, m1>0 et d1>0, nous pouvons donc déduire un encadrement de m et d : K_m <m<(T_1-T_0)- K_d et K_d <d<(T_1-T_0)- K_m. Malheureusement K_m et K_d sont normalement proches de zéro. Le système d'équations donne donc plusieurs solutions dont une seule est exacte. Si le routage de l'information a été similaire en voies montante et descendante, m et d devraient être presque égaux. De plus si T1-T0 est petit, alors m et d sont presque égaux à T1-T0/2 [26]. On ne peut donc que faire converger l'heure locale vers l'heure globale.

Les performances dépendent beaucoup du type de réseau. En effet, si le réseau permet de définir le routage des paquets d'informations, on pourra calculer un temps de latence (montant et descendant) de façon beaucoup plus précise. Reste le problème de l'imprévisibilité d'une congestion momentanée du réseau qui faussera les valeurs. C'est un secteur ouvert en recherche [27] et l'IEEE suit l'évolution des recherches [21].

Certaines solutions sont proposées comme celle de NTP [43]. Des machines spécifiques basées sur GPS permettent d'atteindre des synchronisations d'horloges à moins d'un millième de seconde [61]. Pour des simulations précises avec des temps de latence élevés, on préférera une horloge logique qui définit une relation d'ordre total sur la communication.

2.1.3 DIS / HLA

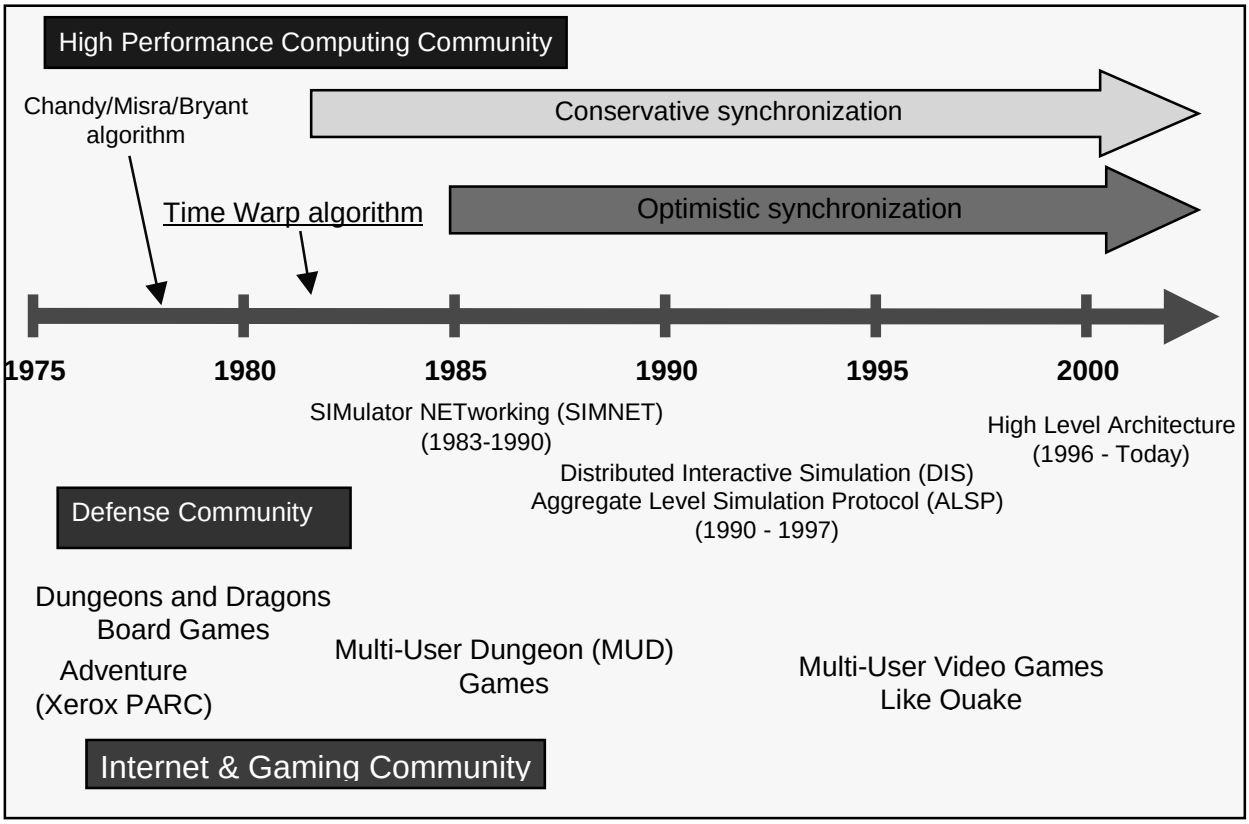
Un jeu d'action multijoueur gère un ensemble d'entités qui interagissent. Celles-ci communiquent en "temps réel" : dès que l'on effectue une action, celle-ci est propagée. On peut donc s'appuyer aisément sur les spécifications de DIS/HLA[46]. DIS/HLA est un standard de l'IEEE. C'est une solution complète pour la simulation partagée ou distribuée. Elle intègre de nombreuses techniques permettant de synchroniser des programmes distribués.

HLA utilise une méthode de distribution (routage) des informations envoyées sur le réseau. La définition du protocole de communication permet au serveur (appelé RTI) d'être un programme générique capable de gérer plusieurs types d'applications. Chaque groupe de programmes communiquant entre eux est appelé 'federation'. Les programmes clients sont appelés 'federate'. Chaque 'federate' peut définir ses propres méthodes de synchronisation et choisir les paquets d'informations qu'il veut recevoir. C'est donc une architecture clients/serveur(s) (le 'federate' étant le client).

A cause de la latence, les entités (données partagées entre les 'federates') ne sont pas à jour. HLA propose d'afficher le dernier état (faux), un état précédent (le dernier état valide connu) ou une interpolation. Pour garantir la cohérence, DIS/HLA propose une technique basée sur une horloge logique gérée par le serveur (à défaut d'un système tel que NTP). Cette horloge peut avancer via un service de temps proposé par le RTI. Cette horloge permet d'utiliser une méthode pour ordonner certains paquets et garantit le principe de causalité (cf 2.1.3.1).

HLA englobe des techniques pour les applications de type jeu multijoueur (simulation partagée) mais aussi pour le calcul parallèle [25].

Contribution du projet DIS/HLA :



2.1.3.1 Time Service

Dans HLA, le temps est géré par le serveur qui délivre une autorisation au 'fédéré'. Le 'fédéré' demande l'autorisation d'avancer dans le temps, jusqu'à une date donnée.

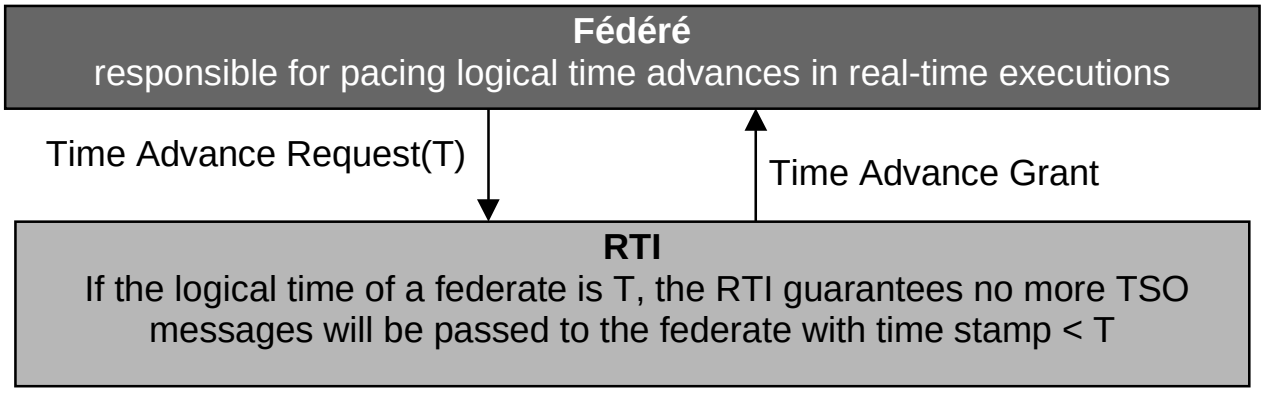


Figure 13 - HLA Time Service

Si le 'Fédéré' est au temps T=17, il demande d'avancer au temps T=18. Le RTI délivrera l'autorisation lorsque tous les messages d'estampille T inférieure ou égale à 17 auront été reçus

par le ‘Fédéré’. Ainsi le principe de causalité est respecté [20]. Le RTI sert d’horloge globale virtuelle.

2.2 Jeux

2.2.1 Approche Pessimiste

C’est le principe de base des jeux au tour par tour : Soit N programmes distribués d’un jeu, chacun appelé P_0, P_1 …, P_N. Au début du jeu, on choisit le premier joueur : chacun lance par exemple un dé et le plus grand résultat commence suivi du joueur qui obtient le résultat décroissant

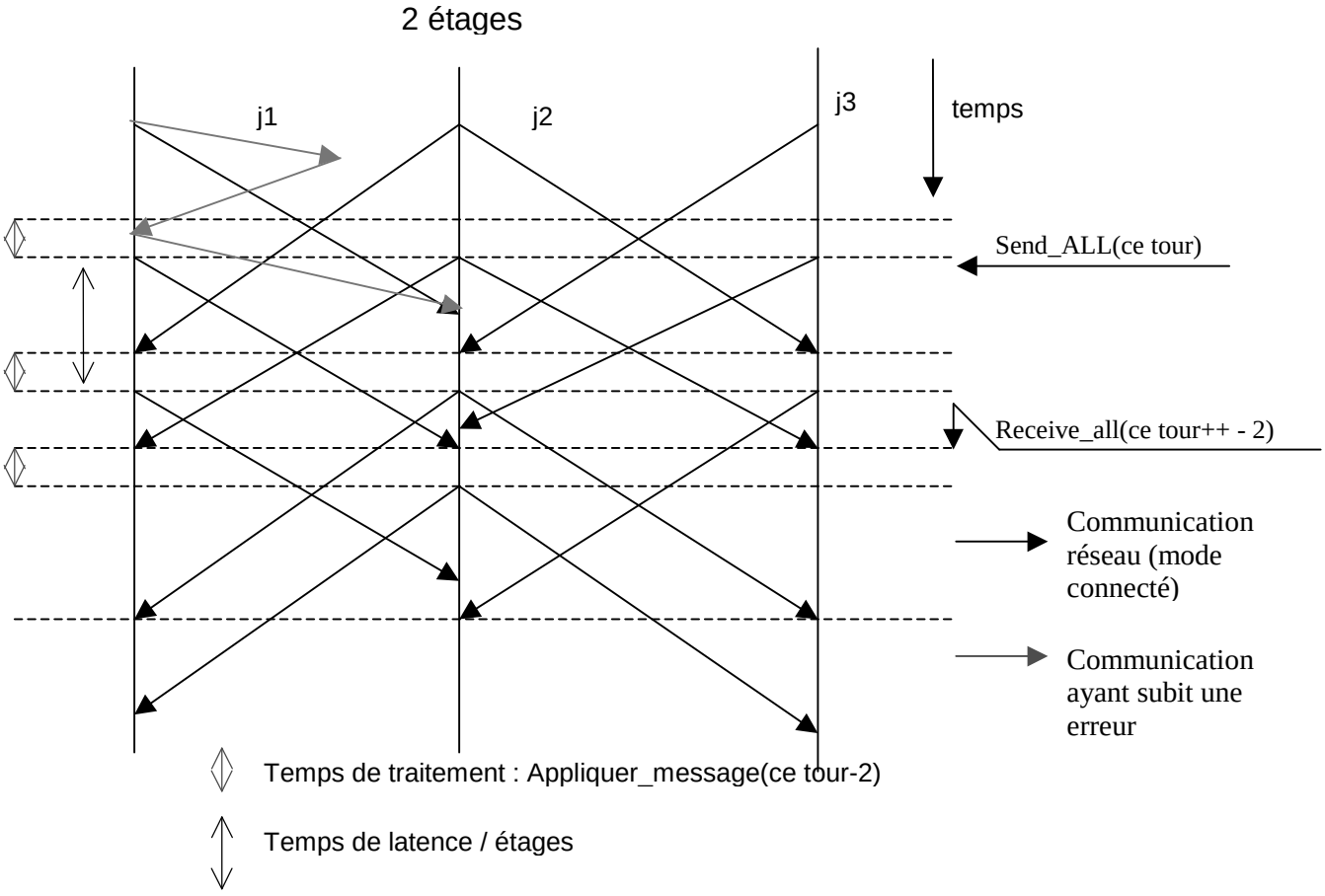


Figure 15 - Algorithme de Synchronisation Pessimiste

suivant. Pour départager des résultats identiques, les joueurs concernés relancent un dé entre eux et font de même. P_0 est le premier joueur suivi de P_1 jusqu'à P_N. P_0 commence le jeu, l’utilisateur

choisit ses actions de jeu qui sont envoyées aux autres joueurs. On peut définir un temps maximum pour cette phase de jeu. Ensuite le programme attend de recevoir les informations du joueur qui le précède dans l'ordre (P_{moi-1} ou si moi=0 P_N).

Cette méthode est celle utilisée par des joueurs de cartes autour d'une table et n'est pas 'temps réel'. Les jeux en réseau sur machine sont variés et les tours de jeu de chaque joueur peuvent parfois se dérouler de manière simultanée. En réduisant le temps du tour de jeu, on réduit le temps d'attente entre chaque tour. S'il tend vers zéro, alors on peut l'utiliser pour des applications 'temps réel'. L'idée est de permettre à chaque joueur de donner des actions de jeu pour un tour puis de passer au suivant lorsque que le tour a été traité par tous les clients. L'état du jeu sera donc validé à chaque tour et chaque tour de jeu sera espacé d'au moins un temps égal au temps de latence.

Si ce temps est trop long : on peut pallier ce problème. Microsoft a employé une méthode de pipeline pour cet algorithme dans son jeu de stratégie Age of empires [35]. Les actions de jeu prises par le joueur au tour i sont effectuées au tour i+2. Ainsi, entre deux états valides du jeu, il se passe (temps de latence maximum/2 + temps de traitement) secondes au lieu de (temps de latence maximum + temps de traitement) si on ne prend pas en compte la bande passante (figure 1). L'avantage de cette technique est la garantie d'équité entre les joueurs malgré des temps de latence et de traitement différents. De plus l'approche pessimiste garantit que l'état du jeu restera cohérent entre les programmes distribués évitant d'avoir recours à des algorithmes de réconciliation. Par contre si l'un des joueurs voit son temps de latence augmenter, tous les joueurs en souffriront. [35]

2.2.2 Time Warp, Approche Optimiste

Time Warp [31] utilise un temps logique global pour la synchronisation et est un algorithme optimiste. TW définit des 'coupes' qui représentent les présents (figure 16) (car la temps s'écoule).

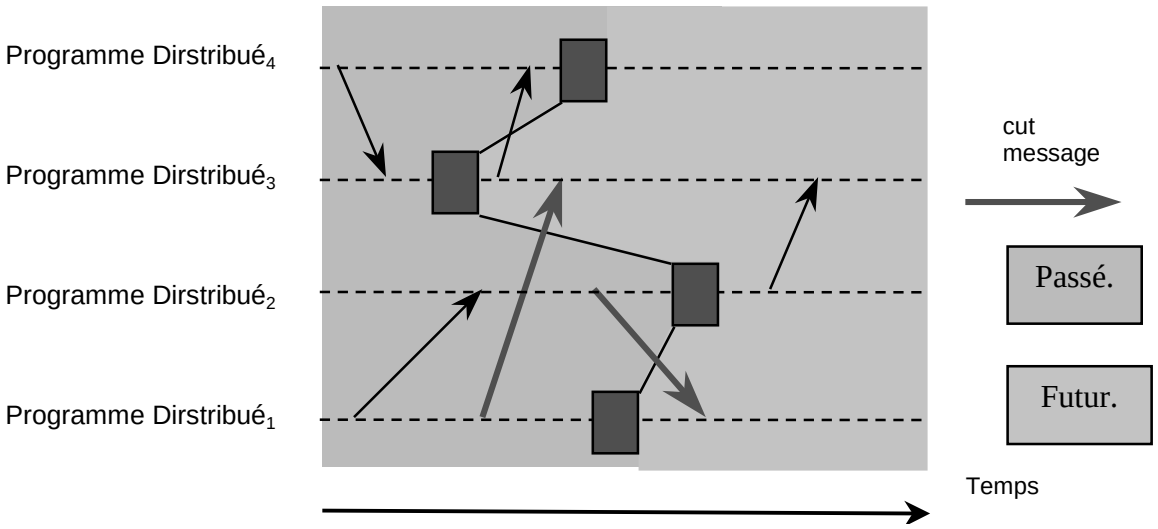
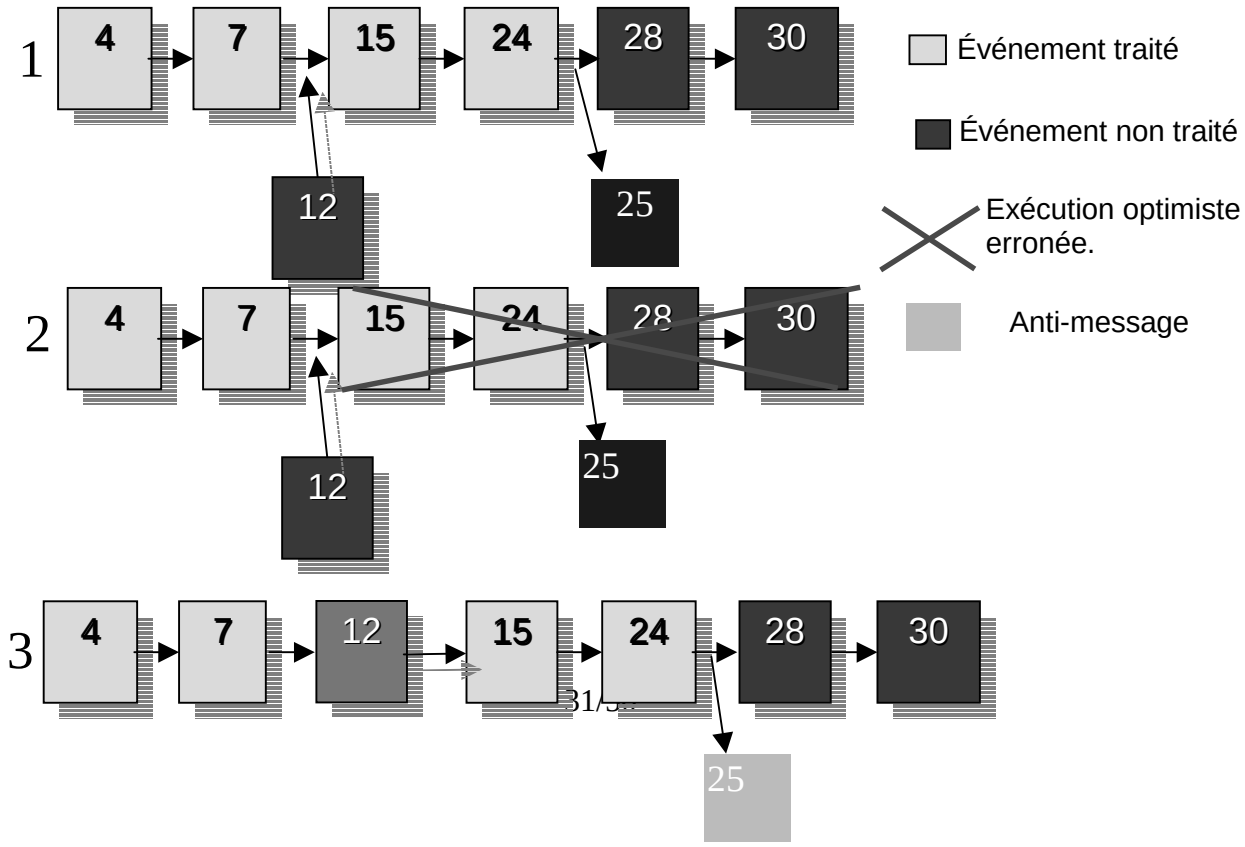


Figure 16 - Time Warp Cut

Ces coupes permettent de déterminer les messages à recevoir et les messages déjà reçus par unité de temps (entre chaque coupe). Sauvegarder les ‘coupes’ avec les ‘cut message’ permet de reconstruire l’état actuel à partir des états antérieurs.

Lorsqu'un processus s'aperçoit que le principe de la causalité n'est plus vérifié, c'est-à-dire que la date courante de la simulation est postérieure à la date du dernier événement reçu, il annule tous les messages déjà expédiés. Pour cela, il faut gérer un historique des états du système, mais aussi le contenu de tous les messages reçus et expédiés afin de pouvoir avertir les autres processus de faire des annulations. Ce retour en arrière dans le temps est appelé “BackTracking”. Pour cela, on utilise des anti-messages qui possèdent exactement les mêmes caractéristiques que



les messages d'origine à l'exception d'un bit de signe.

Ces messages sont générés dès qu'un problème de causalité survient. Ils sont envoyés aux simulateurs pour annuler certains messages. Par exemple, un simulateur A envoie un message M1 au simulateur B qui arrive dans le futur de celui-ci. Si B envoie un message M2 dans le passé de A, celui-ci repasse à l'état sauvegardé où il se trouvait à la date d'arrivée du message M2. Le processus A doit annuler tous les envois de messages qu'il a fait dans le futur, tel que le message M1, en utilisant les anti-messages.

Time Warp est un algorithme clef pour la simulation distribuée [56]. Il existe de nombreuses extensions plus ou moins spécialisées sur un domaine particulier [57][58]. TSS [45] propose une architecture autour de Time Warp pour les jeux de type FPS. Alors que [59] propose une implémentation de Time Warp au dessus de Actor Foundry.

2.2.3 Local Lag & Lag Compensation

Dans MyMAze3D [50], on choisit de fixer le pas de la simulation (temps de l'élément discret de la simulation) en fonction du réseau. Comme dans [35] les actions prises par les joueurs sont retardées puis effectuées en 'même temps' dans des 'buckets' (zone tampon) : c'est-à-dire que les actions prises durant les derniers instants de jeu sont traitées dans le même 'processus' et à la même date de simulation. Si le temps de latence est petit, cette technique peut parfaitement synchroniser des applications temps réel (surtout si le temps de latence est inférieur au temps de calcul d'une frame !). Les différents 'buckets' sont aussi utilisés pour réordonner les informations. L'algorithme est ainsi optimiste mais conserve une cohérence forte garantissant l'équité entre les joueurs (ils ont tous 'les mêmes cartes en mains').

C'est la méthode du 'Local Lag'. Au lieu d'effectuer immédiatement les actions du joueur, elles sont retardées dans le temps. Cette méthode est compatible avec Time Warp [48]. Par contre, les algorithmes de prédiction doivent prendre en compte le temps de retard actuel.

Si les collisions sont gérées par le serveur, celui-ci peut considérer le temps de latence de l'utilisateur et corriger leur valeur pour calculer si la collision a eu lieu aux yeux de l'utilisateur. Cette technique, Lag Compensation, est présentée dans [56]. Elle est proche du 'Local Lag' mais au lieu de retarder les actions de l'utilisateur, on calcule ses interactions personnelles en retardant l'état du serveur. Cette technique n'est donc pas vraiment compatible avec des algorithmes de prédiction. De plus, elle favorise les incohérences augmentant le nombre de retours en arrière.

2.2.4 Le Dead Reckoning

Le Dead Reckoning n'est pas une technique de synchronisation [48]. Elle permet d'évaluer la position d'une entité à partir d'un état antérieur connu (Position, vitesse, accélération). Elle a pour but de compenser directement la latence dans les simulations temps réel.

Tant que l'on ne reçoit pas de mise à jour, on considère que l'utilisateur continue d'effectuer sa dernière action (au lieu de rien). On calcule donc une prédiction de sa position en prolongeant son mouvement [46] avec la mécanique du point:

- P(t) = Position d'une entité au temps t
- Message de modification de position: $P(t_1)$, $P(t_2)$… $P(t_i)$
- $v(t_i)$, $a(t_i)$ = $i^{ème}$ vitesse, Modification de l'accélération.
- L'Algorithme de Dead Reckoning (ADR): estime D(t), position au temps t
  - t_i = moment de la dernière modification avant t
  - $d_i = t_i - t$
- ADR à l'ordre 0 :
  - $D(t) = P(t_i)$
- ADR du premier ordre:
  - $D(t) = P(t_i) + v(t_i)*d_i$
- ADR du second ordre:
  - $D(t) = P(t_i) + v(t_i)*d_i + 0.5*a(t_i)*(d_i)2$

Le premier raffinement est de prendre en compte l'environnement et d'éviter de prédire une position absurde comme derrière une barrière infranchissable. Si une voiture avance vers une épingle à cheveux et que sa vitesse est assez grande pour que la prédiction la place hors de la route on peut penser que l'utilisateur cherchera à tourner avant. On peut donc choisir une technique de Dead Reckoning qui cherche à suivre un chemin par défaut [22].

- ADR du premier ordre avec correction par rapport à 'I' (fig 17,18) une position par défaut:
  - $D(t) = ((P(t_i) + v(t_i)*d_i)+ I)/ 2$

— — — — — — Résultat de l'ADR

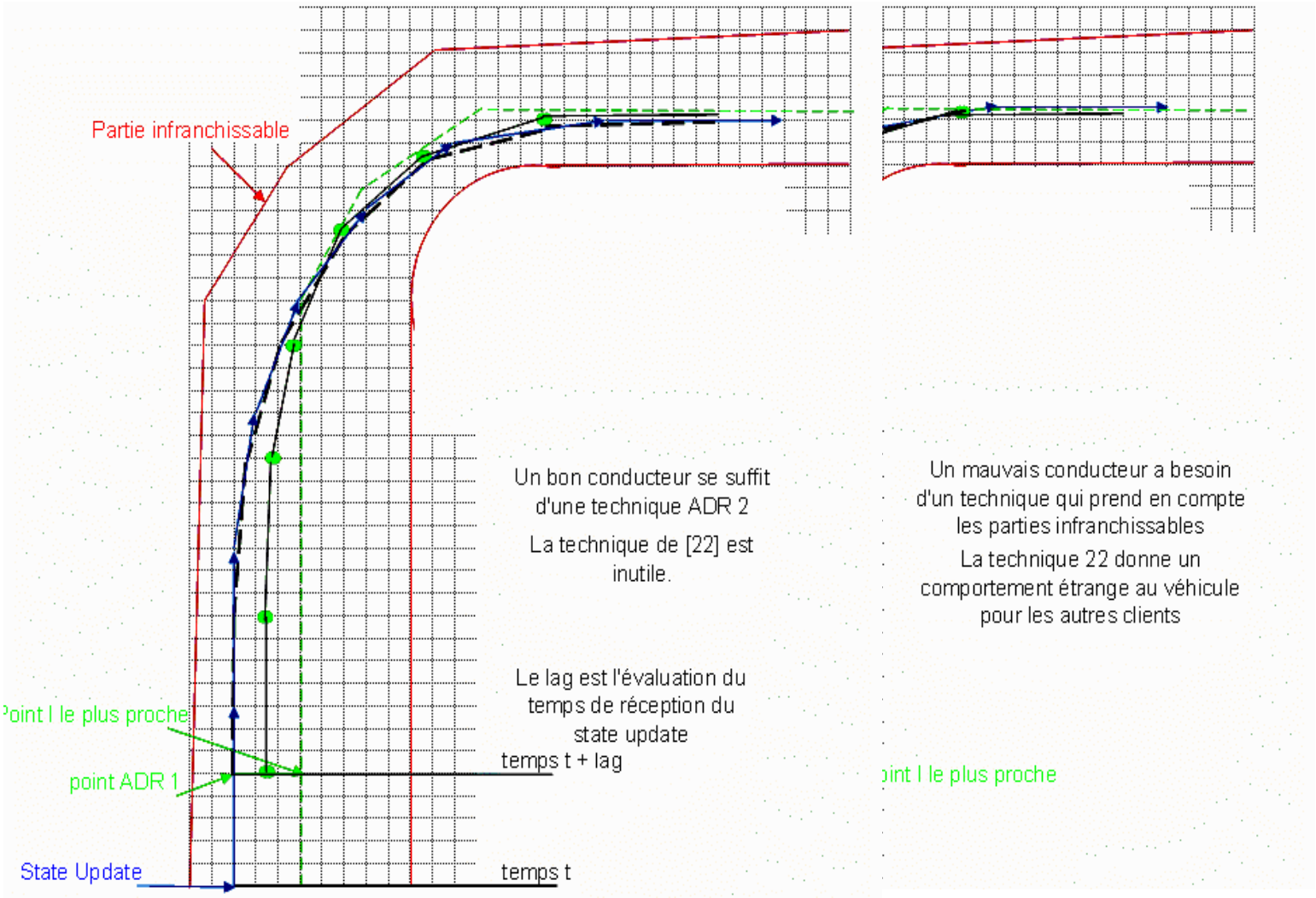


Figure 18 - effet du Dead Reckoning 1

Figure 17 - effet du Dead Reckoning 2

Pour une entité dirigée directement par le joueur et sans contrainte (adhérence …) :

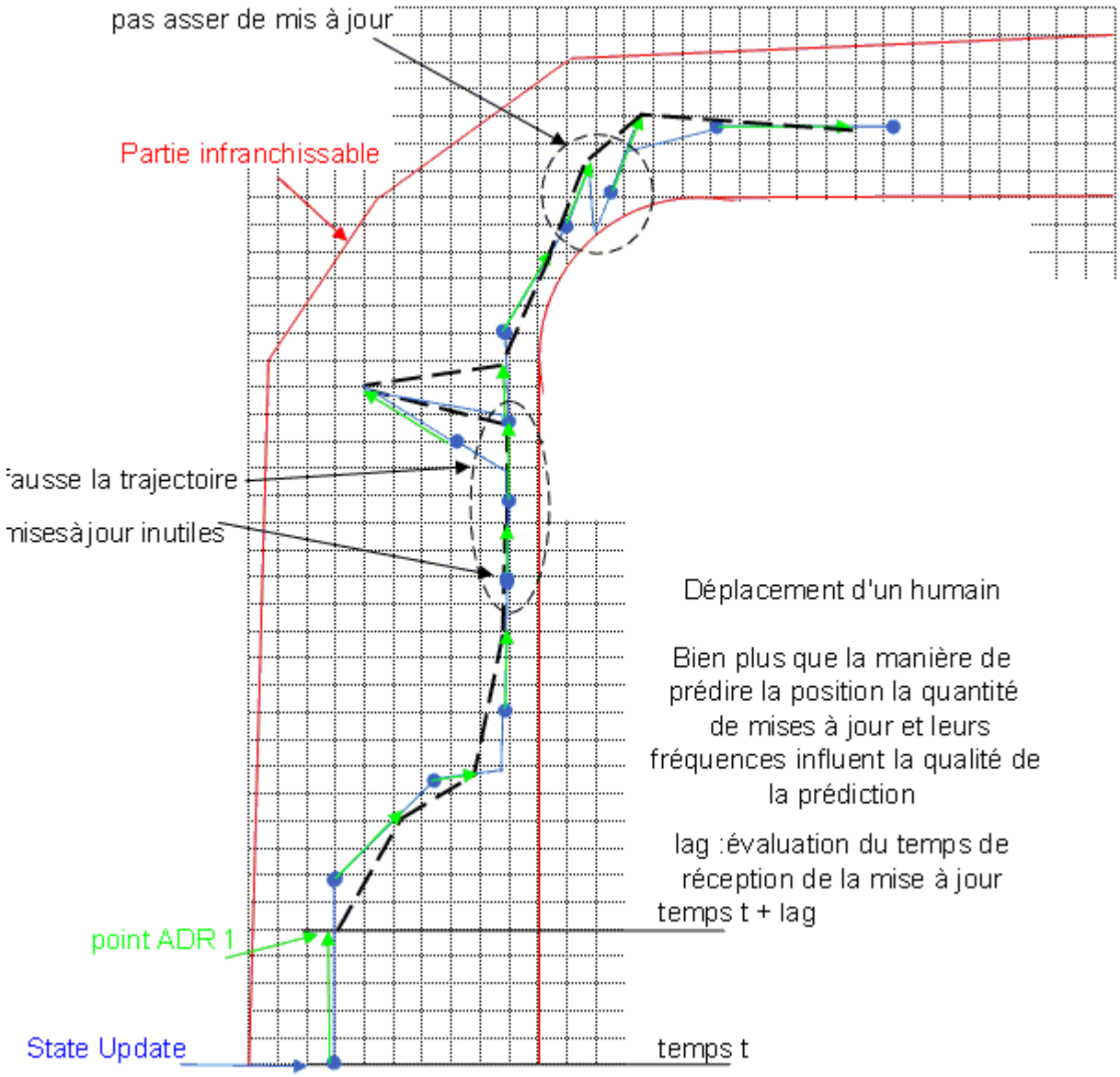
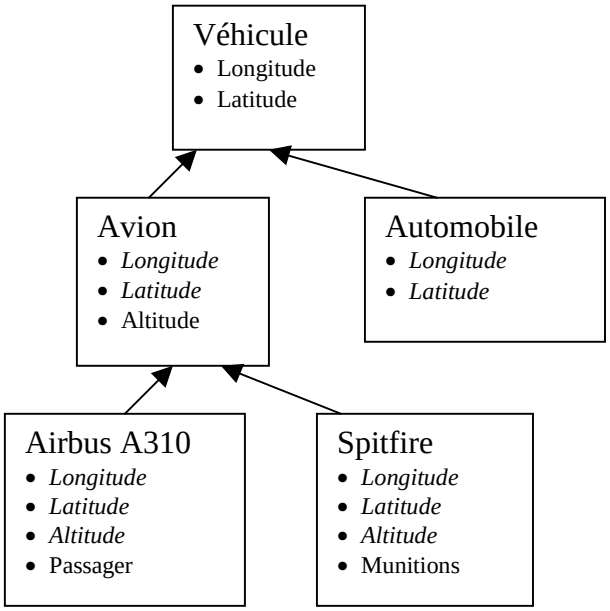


Figure 19 - effet du Dead Reckoning 3

Le Dead reckoning est particulièrement efficace lorsque le déplacement est soumis à des règles physiques restrictives comme pour une voiture (figure 17 et 18) [44]. Lorsque les déplacements sont plus aléatoires, le nombre de mises à jours envoyées devient plus important (fig 19). L'un des meilleurs moyens d'envoyer des mises à jours est d'attendre un changement de direction ou d'accélération (si la valeur existe pour l'entité).

2.2.5 Publication et abonnement

Le publish-subscribe est une méthode essentielle pour espérer pouvoir passer à l'échelle [14]. En effet, quel que soit le type de synchronisation adopté, la quantité d'informations à transporter augmente au moins de façon linéaire. Or les réseaux sont limités en bande passante. Il faut donc réduire les quantités de données à transporter. Le publish-subscribe permet de décider quels messages je veux recevoir (subscribe) et quel message je suis susceptible d'envoyer (publish). Cette méthode permet de configurer le routage des données dans l'application distribuée. Les paramètres de la souscription peuvent être implicites : en fonction, par exemple, de la zone 'géographique' virtuelle d'activité [23] ou générique comme dans DIS/HLA. HLA propose de classer les entités entre elles pour créer des expressions permettant de souscrire à des groupes ou de publier pour des groupes.



Exemple d'expressions DIS / HLA :

Publish (Description expression):
– (automobile, (15, 135))
– (avion, (X, Y)) where 35 < X < 38 and 98 < Y < 100

Si on envoie des paquets d'informations DIS, ces informations sont marquées dans l'en tête du message et le serveur livrent les paquets aux clients qui ont une 'Interest expression' unifiable avec.

Subscribe (Interest expression) :
• (avion, (X, Y)) for any X and any Y
• (automobile, (X, Y)) where 10 < X < 20, and 130 < Y < 150

" automobile (15, 135)" est unifiable à " (automobile, (X, Y)) where 10 < X < 20, and 130 < Y < 150" avec X=15 et Y=135.

Exemple de classement d'entités : *Attribut hérité*

2.2.6 L'aléatoire distribué

Lorsque le jeu effectue en local une opération destinée à éviter d'attendre un résultat du réseau, c'est une prédiction locale. Si le calcul ne dépend que de données locales il est exact, sauf

s'il dépend d'un nombre aléatoire. Les programmes doivent pré-calculer les valeurs aléatoires puis les récupérer, à distance si nécessaire, pour avoir des valeurs aléatoires identiques. Un service de synchronisation complet doit donc proposer sa propre fonction rand() pour obtenir des nombres aléatoires identiques en mode distribué. Le serveur peut calculer un grand nombre de valeurs puis les envoyer aux clients qui auront alors une pile locale de nombres aléatoires qui pourra être alimentée aux moments propices (à la fin d'un paquet pour le remplir par exemple). Si l'application doit effectuer un calcul sur une donnée réconciliable et que ce calcul dépend d'un nombre aléatoire, il faudra piocher dans la file. Le nombre tiré sera indiqué aux autres pour qu'ils vident leurs files, s'ils n'avaient pas à faire le calcul, et pour vérifier que la pile actuelle est valide.

2.3 Conclusion

Effectuer une simulation interactive distribuée, ou un jeu multijoueur, est parfaitement possible dans le cadre de machines fixes telles que des ordinateurs de bureaux ou des consoles de salons. Le jeu vidéo a participé à l'élaboration des techniques de synchronisation. Les jeux multijoueurs sont de plus en plus équitables, pour les joueurs, et sécurisés. Il reste deux problèmes majeurs : le passage à l'échelle ou la complexité des algorithmes de synchronisations et la mobilité. La mobilité n'est pas un problème si l'on fait abstraction du temps de latence. Le passage à l'échelle est lié à la qualité de la gestion des entités et de leurs interactions : Il ne faudra mettre à jour qu'une partie des entités distribuées. Nous verrons dans le chapitre suivant comment mettre en œuvre un algorithme de synchronisation optimiste, sans anti-messages ou ré-exécution, et donc plus adapté au monde mobile.

3 Un jeu de plateforme multijoueur avec HTTP

Pour réaliser des expérimentations, nous utilisons la plateforme GASP, destinée aux jeux en réseau sur téléphones mobiles, et le jeu FreePussy™ de la société Pastagames. Le jeu a tout d'abord été modifié pour en faire un jeu multijoueur. A travers ce prototype de démonstration, nous montrerons qu'il est possible de faire des jeux vidéo multijoueurs sur mobile via GPRS. Ces jeux devront tout de même être conçus en gardant à l'esprit les contraintes dues aux temps de latences.

3.1 Vers un jeu multijoueur

Figure 21 – FreePussy™ capture d'écran

FreePussy™ est un jeu de la société Pastagames disponible aujourd'hui sur portables DOJA. Le jeu a été porté pour être compatible MIDP. Pour tester le jeu nous utilisons l'émulateur de Sun Microsystems compatible MIDP 2.0 et CLDC.

Le but du jeu est de se déplacer en évitant les ennemis pour atteindre son amie (vous incarnez un chat). Pour vous aider, des objets sont disséminés dans le niveau (bombes, jetpack, carburant, nourriture…). Certains objets sont essentiels à votre réussite.

Nous présentons l'architecture de FreePussy pour bien définir comment le temps est géré dans le jeu. Puis nous présentons une version multijoueur de FreePussy développée dans le cadre du projet de MEGA.

Mur fragile.

Chatte à libérer.

Dangereux ennemis.

Mur.

Vous/Le joueur (équipé d'un système de propulsion).

3.1.1 Présentation du moteur de jeu

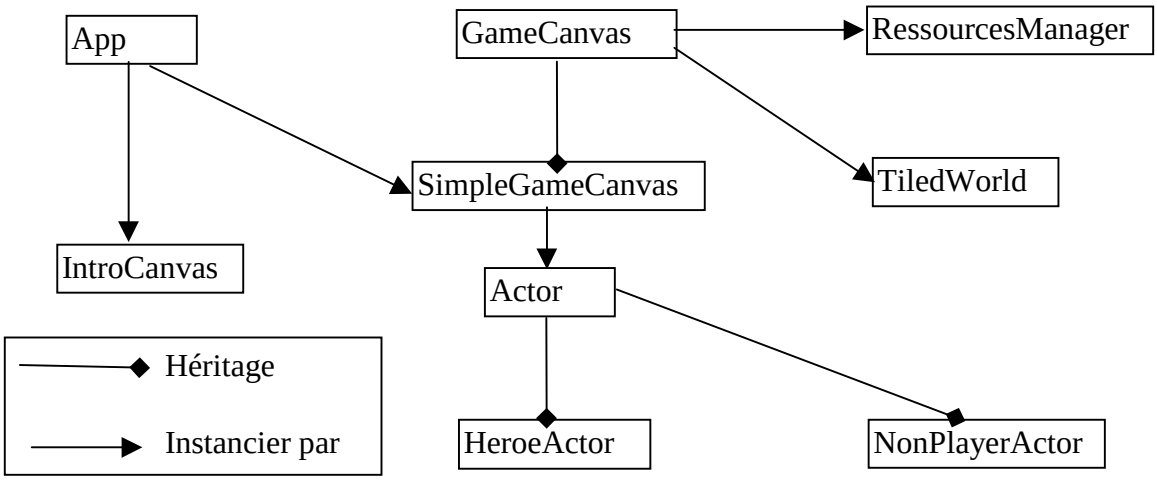


Figure 22 – FreePussy™ Architecture (survol)

La classe *App* est le point d'entrée de l'application. *App* hérite de *MIDlet*. Un *MIDlet* est pour le téléphone une application MIDP/CLDC, il est comparable aux applets pour les navigateurs.

IntrosCanvas Affiche le logo Pastagames avant de lancer le jeu.

GameCanvas est le cœur du moteur du jeu : c'est ici que sont gérées les collisions et que l'on appelle les méthodes des autres objets pour les faire avancer dans le temps. Le monde est représenté via la classe *TiledWorld* et les acteurs ou objets du jeu héritent tous de la classe *Actor*. La classe *RessoucesManager* permet d'accéder aux ressources telles les images ou les informations sur la forme du monde. *SimpleGameCanvas* implémente les caractéristiques spécifiques au jeu (gestion des niveaux ..).

HeroeActor gère l'avatar du joueur. C'est ici que l'utilisateur modifie le comportement de son avatar. Les instances de *NonPlayerActor* permettent de gérer tous les autres objets visibles dans le jeu.

3.1.2 Horloge locale

FreePussy gère son horloge d'exécution. L'objet *GameCanvas* instancie une variable (game_time) pour définir le temps actuel de la simulation. Cette technique est utilisée dans les jeux pour définir des vitesses de déplacement indépendamment de la capacité calculatoire de la machine. Cette horloge peut être utilisée pour la synchronisation : tout le monde commence la

partie à 0.

A chaque tour de jeu :

```
void GameCanvas::PlayingGameRound()
    {
            calculateGameTimes();
            if (levelIsFinished())
            {
                quit();
            }
            executeUniverse(game_time, delta_time );
    }
```

Le moteur incrémente le temps de la simulation :

```
void GameCanvas::calculateGameTimes( ) {
    current_system_time = System.currentTimeMillis();
    delta_time = (current_system_time - previous_system_time);
    previous_system_time = current_system_time;
    if (delta_time > 150) delta_time = 150;
    game_time += delta_time;
}
```

Delta_time est borné car si le tour de jeu dure plus de 150 ms alors le déplacement de l'avatar sera trop important et le jeu ne sera plus jouable.

3.1.3 Intégration de GASP

Figure 23 - FreePussy Multijoueur

GASP est une plateforme développée dans le cadre du projet MEGA qui permet la communication entre des téléphones portables compatibles GPRS en respectant une partie des normes de l'OMA. GASP fournit deux classes *MGPServer* et *MGPClient*. A partir de ces classes nous pouvons implémenter des comportements spécifiques pour les clients ou le serveur (cf. fig. 24).

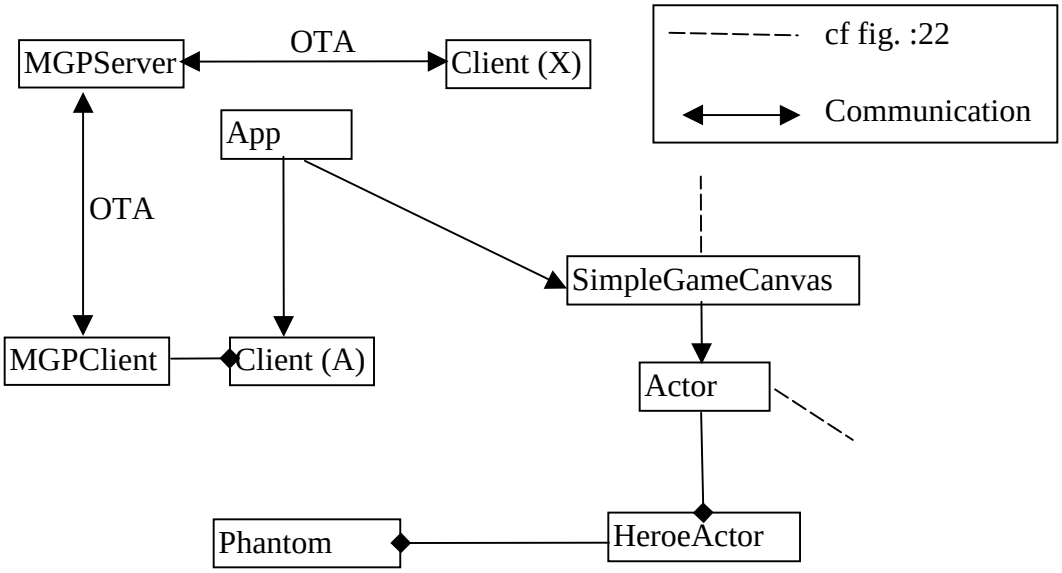


Figure 24 - FreePussy Clients/Serveur

Le temps de latence étant important, nous avons choisi de minimiser les interactions entre les joueurs et d'éviter toutes concurrences. Les entités partagées sont les instances de la classe *actor* ainsi que l'objet *TiledWorld* qui reprèsente le monde. Les instances de '*non player actor*' et les instances de la classe *phantom*, représentant les avatars des autres joueurs sont mises à jour dans l'objet *client*. Chaque joueur a maintenant une tâche unique (lanceur de bombe, pilote de jetpack, libérateur).

3.1.4 Reconnexion

GASP permet à un utilisateur déconnecté en cours de partie de rejoindre celle-ci ultérieurement. Pour que le jeu supporte cette fonctionnalité il faut prévoir de maintenir : soit l'état actuel du jeu sur le serveur, soit les actions permettant de reconverger vers le même état. Il faut utiliser le serveur (*MGPServer*) pour stocker ces informations car rien ne garantit que les autres joueurs n'ont pas quitté la partie (et donc détruit les informations nécessaires à la reconnexion du tiers).

3.2 Synchronisation dans le jeu

FreePussy est maintenant un jeu à 3 joueurs utilisant HTTP. Le héros a été scindé en plusieurs joueurs : chacun a une capacité spéciale. Le jeu est coopératif et les interactions entre joueurs assez

faibles. Le fait que le jeu ne propose pas de s'affronter permet d'éviter les problèmes d'équités entre joueurs.

L'un des premiers enjeux a été de chercher à gérer un nombre relativement important de messages. Nous présentons ici un protocole minimal, ainsi qu'une architecture, permettant une évolution vers un plus grand nombre de joueurs. Enfin, nous présentons une gestion du temps qui permet de proposer des accès à des ressources en concurrence.

3.2.1 Protocole pour le jeu

Dans cette section, nous décrivons les messages utilisés pour le jeu. Ces messages spécifiques au jeu sont indépendants de ceux utilisés pour le menu ou pour les communications GASP.

3.2.1.1 Algorithme Naïf

Il n'y a pas beaucoup de résultats sur les besoins en terme de ressources pour le jeu multijoueur sur mobile. Avant de chercher à mettre en place des algorithmes de synchronisation souvent coûteux, nous cherchons à gérer un flux important de messages. En effet, plus il y aura de joueurs, plus le nombre de messages à gérer augmentera (linéairement). Comme FreePussy n'a que trois joueurs, nous utilisons un protocole naïf qui, à chaque tour de jeu, envoie un message contenant la position du joueur et le temps actuel de la simulation.

Nous utilisons un type unique de message comportant les coordonnées de l'avatar et le temps actuel de la simulation (GASP informe sur l'émetteur du message). Ces messages sont créés à chaque tour de jeu.

Ces premiers tests démontrent que le coût de gestion des messages n'est pas négligeable. Le nombre d'images par seconde du jeu diminue grandement et le plaisir de jeu aussi. Pour éviter que le coût d'envoi des messages ne fausse ces résultats, nous avons créé une classe permettant d'envoyer des messages sans en ré-instancier de nouveau (FifoStaticMessage). Nous verrons comment les messages sont reçus et traités dans la section 3.2.3. Dans la section suivante, nous proposons un protocole optimal.

3.2.1.2 Protocole

Dans FreePussy nous pouvons utiliser un protocole similaire à celui proposé dans [60] mais qui est rejeté pour des raisons de sécurité. Dans ce protocole, les clients doivent être sûrs car ils effectuent tous les calculs pour faire évoluer la simulation. La

plateforme GASP permet de passer outre grâce à une gestion d'identification double : un identifiant unique donné par l'opérateur ainsi qu'un identifiant délivré par GASP lors du login permettant de gérer les utilisateurs lors de l'exécution. Ainsi, seul les clients sûrs peuvent effectuer des requêtes sur le serveur de jeu. Pour recréer le mouvement (et ses interactions) de l'avatar sur les mobiles, il suffit de connaître les modifications de vitesses horizontales, les débuts de sauts, l'utilisation du jetpack et les lancés de bombes. Pour encore minimisé la fréquence d'envoi des messages, certaines modifications de vitesse peuvent être ignorées. Ce protocole sous-entend que les interactions des avatars des autres joueurs sont gérées en local. Certains de ces messages influent sur la causalité, ils sont donc essentiels, si le message de 'lancé de bombe' n'est pas traité les conséquences de l'acte seront ignorées en local. Il faudra donc réconcilier pour éviter des incohérences. La figure 26 décrit les messages utilisés, les messages de types 3 et 5 sont les seuls à influer sur la causalité dans le cas de FreePussy™.

Les messages utilisés peuvent rester génériques (similaires à ceux utilisés dans l'algorithme naïf avec un tableau d'entier) ce qui permettra de garder *FifoStaticMessage* (et la possibilité de créer une classe similaire pour le flux descendant). Dans le cas d'un protocole minimal qui ne rend compte que des actions essentielles, la perte de message devient critique.

```
public class Message
{
	private long mTimestamp; //temps de simulation lors de l'action
	private byte type; /*type : définie le type d'action
		0- Id update // utilisé pour le menu
		1-just moving (when direction/speed change)
		2-jumping (when jump)
		3-firing (when lauching a bomb)
		5-Hitting (when destroying a wall)
		6-dying
	*/
	private int coord[];    //position du joueur lors de l'action
	private int dummy[]; //données spécifique à l'action

}
```

Figure 26 – Descriptions des messages

3.2.2 Architecture

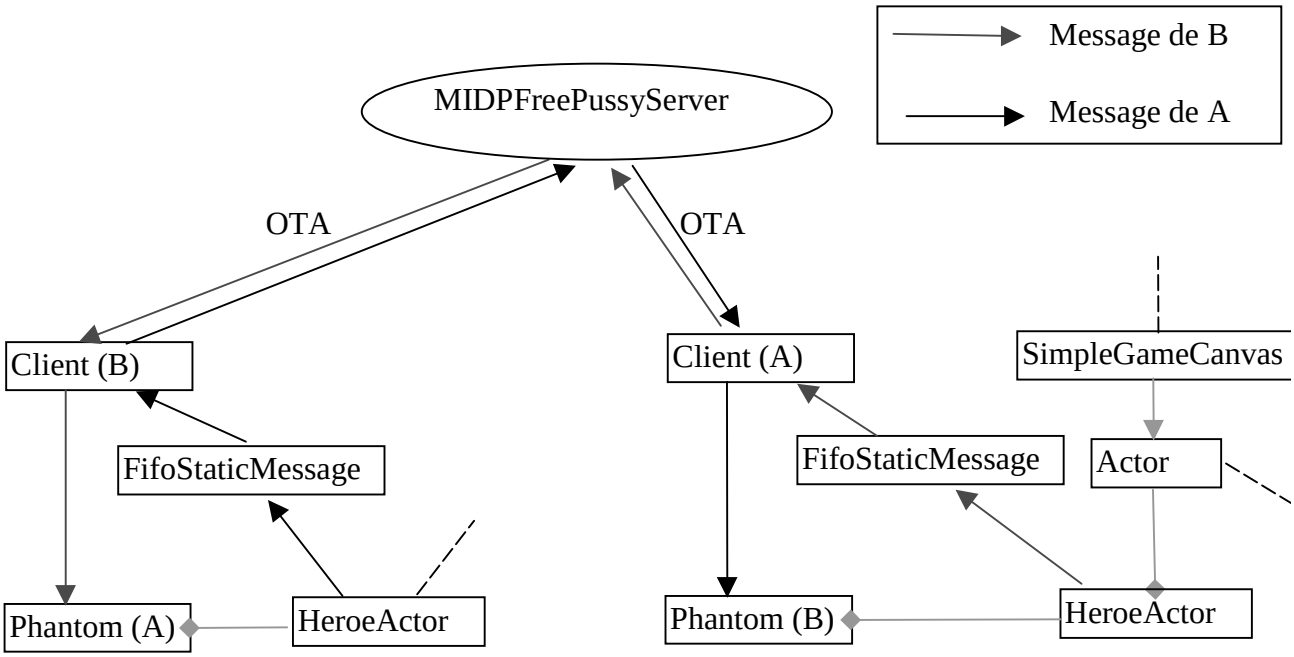


Figure 25 - Architecture FreePussy, routage

Chaque client conserve un tableau pour associer un avatar à chaque joueur de la partie. Chaque message contient un identifiant permettant de connaître sa provenance (son créateur). L'objet Client doit gérer les messages de GASP (login, join, ..) ou du menu et router les messages vers les instances de *Phantom* (entité partagée d'un joueur). Il effectue les requêtes HTTP dans un thread spécifique pour éviter tout blocage dû à l'attente d'une réponse HTTP. Pour les expériences, nous effectuons une requête s'il y a plus de 12 messages à envoyer ou si 20 tours de jeu ont été joués sans aucune requête. L'architecture est assez similaire à celle proposée dans les spécifications HLA. On verra dans la section suivante que la gestion du temps est similaire mais totalement effectuée au niveau du client.

3.2.3 Gérer les messages entrants

Comme vu en 3.1.2, chaque client gère un temps de simulation; on ajoute donc dans la classe *phantom* un temps de simulation qui lui est spécifique. Comme la latence est trop élevée, on n'utilise pas de technique de prédiction pour les avatars des joueurs. Les résultats des prédictions seraient trop erronés au vu des temps de latence. Chaque message représente une action. L'objet phantom exécute ces actions en prenant les messages disponibles dans une zone tampon.

3.2.3.1 Algorithmes

Pour les premiers tests, la zone tampon de réception des messages était une FIFO; nous avons pu ainsi observer comment les messages arrivaient et dans quel ordre. Certains messages mettent beaucoup plus de temps que d'autres à arriver. Ce désordre est principalement lié à l'implémentation de GASP utilisée pour les tests. Nous avons ensuite utilisé une table de hashage pour ranger les messages entrants en fonction de leurs temps de simulation. En utilisant un deuxième estampillage pour définir un ordre total entre les messages, le ré-ordonnancement peut être effectué en temps constant. De plus, un ordre total permet de détecter d'éventuels messages perdus (cf 3.2.3.2).

La zone tampon reçoit les messages qui concernent son *phantom*, routé par l'objet Client. Si l'un des messages est plus vieux que le temps de simulation de son *phantom,* il est supprimé, sauf si le message influe sur la causalité (cf 3.2.3.4 et 3.2.1.2). La zone tampon fournit le message le plus vieux qu'elle contient.

L'objet *phantom* lit dans la zone tampon les messages à chaque tour de jeu et fait avancer son temps de simulation. Dans le cas d'un protocole minimal, gérer un message peut prendre plusieurs tours de jeu.

Algorithmes de gestion des messages:
. Initialisation
0 *t=0*
1 *a=buffeur.prochain_message(t),goto 2;*
. Gestion
2 *si (a est instancié) b=buffeur.prochain_message(t),appliquer a, goto 3 sinon goto 1;*
3 *si (b est instancié) goto 5,t++,goto 4 sinon goto 2;*
4 *si (applyed b) a=b,b=rien, goto 2 sinon goto 3*
5 *si (this.on screen || b.on screen || b.priotaire) appliquer a->b sinon a=b,b=rien, goto 2*

Les tests 'on screen' sont un exemple d'optimisation pour gérer plus de joueurs : s'il ne sont pas à l'écran, on n'effectue que les messages influant sur la causalité.

Au lieu de prédire un nouvel état à partir du dernier connu, nous traitons les messages par paire. A l'aide de l'estampillage de 3.2.1, on connaît le temps que le joueur a mis pour passer de l'état fourni dans le message précédent jusqu'au prochain message. Nous pouvons donc recréer les évolutions du joueur de façon fidèle et continue. Enfin nous utilisons la zone tampon locale pour gérer le temps sans contrainte de latence.

3.2.3.2 Gestion du temps et ordonnancement.

En fonction des caractérisques du middleware de communication, les messages sont reçus de diverses façons. Dans le cas d'une communication non sûre, il faut utiliser un estampillage spécifique et différent de celui proposé en 3.2.1. Cet estampillage, plus classique doit permettre de définir un ordre logique total sur l'ensemble des messages d'un seul joueur. Une numérotation des messages peut donc suffire. A l'aide de celle-ci, nous pouvons détecter les messages perdus et complètement optimiser leurs ordonnancements. Il suffit de ranger les messages dans un tableau de listes de messages à T cases à la case du numéro de message modulo T.

Pour gérer les messages, nous choisissons d'avancer nous-même le temps de simulation, en l'incrémentant sous certaines conditions; ainsi lorsque l'on gère une transition, on peut modifier le temps nécessaire pour passer au prochain message en fonction du nombre de messages disponibles. Cette technique permet de conserver un nombre raisonnable de messages en mémoire en accélérant leur traitement, mais aussi d'essayer de conserver un mouvement pour l'avatar du joueur distant : si la zone tampon est presque vide, on rallonge le temps d'exécution des actions.

3.2.3.3 Collision et prédiction locale

L'interaction entre un joueur et l'environnement est calculée par le client. Lorsque l'avatar d'un autre joueur interagit avec le décor, par exemple en passant sur un bonus, l'action doit être traitée. La classe *phantom* utilise le moteur de collision du jeu et ce de la même façon que HeroeActor. Notons que cela permet aux avatars des joueurs de rentrer en collision.

Si l'on veut privilégier la rapidité du jeu et non la taille des messages envoyés, on peut ne pas utiliser le moteur de collision local et inscrire dans les messages les informations nécessaires à l'accomplissement de l'action.

3.2.3.4 Réconciliation

La gestion optimiste des actions à effectuer permet d'ignorer certains messages : soit si l'entité partagée reçoit un message plus ancien, soit en avançant son temps de simulation alors qu'aucun message n'est traité.

Si l'on ignore un message de type 'collision avec un mur', on aura alors une incohérence forte si celui-ci devait être détruit. Certains messages sont donc prioritaires sur d'autres : dans le cas où ils pourraient être ignorés, nous les traitons directement en calculant le résultat de l'action à l'aide des données supplémentaires du message.

3.2.4 Accès concurrent aux ressources

Le client gère des instances de la classe *phantom*; celles-ci reçoivent des messages de la part des autres clients. Chaque instance de ces classes représente l'avatar d'un autre client et a un temps de simulation utilisé pour la gestion des messages.

Le vecteur des temps de simulation de chaque instance de la classe *phantom* représente une horloge logique qui permet de gérer des accès à des ressources en concurrence. Malheureusement, si certains messages arrivent en retard, il faut vérifier l'absence d'un message dans la chaîne de traitement pour gérer le temps de simulation du *phantom*. Ce problème résolu (cf 3.2.3.2), il suffit d'attendre que toutes les instances de la classe *phantom* aient dépassé un temps 't' pour connaître qui a accès à une ressource qu'au moins un joueur aura essayé d'acquérir avant 't'. Une prédiction de type dead-reckoning permet d'accélérer ce calcul : Soit L la longueur maximale que peut parcourir l'avatar en T=t – 'son temps de simulation', si la distance ressource/joueur est plus grande que L=T*'vitesse maximum' ou s'il n'existe pas de chemin entre la ressource et le joueur, alors on peut considérer que, pour cette ressource, le temps de simulation du *phantom* est plus grand que 't' et donc ce *phantom* ne pourra pas accéder à cette ressource.

4 Conclusion

La relation très forte entre l'interface et les besoins en terme de synchronisation est générale. Par contre, dans le cadre du jeu vidéo, il s'y ajoute le problème de la jouabilité. L'interface doit être efficace mais elle doit aussi favoriser l'amusement du joueur. Pour développer un jeu multijoueur sur mobile, il faut absolument prendre en compte les restrictions du réseau dès le début du projet.

La simulation distribuée est un secteur actif de la recherche et est proche du jeu multijoueur. Le passage à l'échelle reste un problème à la vue des tailles mémoires disponibles sur équipements portables. Les services réseaux actuels ne sont pas adaptés aux jeux vidéo temps réel. L'émergence de nouvelles consoles comme la Playstation portable de Sony ou la console/téléphone N-Gage de Nokia montre qu'il existe pourtant un débouché pour ce genre de produits.

Les nouveaux réseaux sans fil pourraient être une alternative au GPRS dans un proche futur. Bluetooth permet de se connecter à environ 6 autres machines. Le GPRS garde l'avantage de couvrir des zones beaucoup plus larges. Nous avons montré qu'il est possible de réaliser un jeu multijoueur aux interactions modestes et utilisant HTTP sur GPRS. Il est possible que de nouveaux types de jeux, adaptés à ces caractéristiques, voient le jour.

5 Bibliographie

[1] http://www.totalwar.com/community/medieval1.htm
[2] http://www.cossacks.com/
[3] http://panzergeneral3.com/
[4] http://www.idsoftware.com/games/quake/quake3-gold/
[5] http://games.sierra.com/games/half-life/
[6] http://www.gamekult.com/tout/jeux/fiches/J000004351.html
[7] http://www.wizards.com/default.asp?x=d20/welcome
[8] http://www.gamekult.com/
[9] http://www.dwango.com/games/star_diversion.htm
[10] Mobile Games In Japan, Collier David, Game Developer 2003.
[11] http://www.idsoftware.com/games/consoles/doom-gba/
[12] Game Mobility Portability, Henkel Guido, Game Developer 2004.
[13] Mobile Games, Matt Kelland, Lasse Seppännen, David Fox, Thomas Puha, PatrickGardner. Gamasutra features.
[14] Massively Multiplayer Middleware, Michi Henning, ZeroC.
[15] Object Placement in Distributed Multiplayer Games, Jeffrey Pang, Justin Weisz, 12/12/2003
[16] Differences between Ice and CORBA. ZeroC, http://www.zeroc.com/iceVsCorba.html.
[17] On the Impact of Delay on Real-Time Multiplayer Games, Lothar Pantel, Lars C. Wolf
[18] ARCHITECTURES POUR JEUX SUR MOBILE ADAPTEES AUX TEMPS DE LATENCE DES RESEAUX MOBILES 2G/3G, Eugeniusz HETMANSKI, 07/2004, Projet MEGA.
[19] Développer des Jeux sur Mobiles, DESS Jeux Vidéo, cours par Fabien Delpiano, Pastagames
[20]Time Management in the High Level Architecture, Richard M. Fujimoto, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332-0280, fujimoto@cc.gatech.edu
[21] The new standard, IEEE 1588 (TM), "Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems.
[22]PRE-RECKONING ALGORITHM FOR DISTRIBUTED VIRTUAL ENVIRONMENTS, Thomas P. Duncan, Denis Graèanin, Department of Computer Science, Virginia Tech, 7054 Haycock Road, Falls Church, VA 22043, U.S.A.
[23] Mercury : A scalable Publish-Subscribe System for Internet Games Ashwin R. Bharambe, Sanjay Rao, Srinivasan Seshan, School of Computer Science, Carnegie Mellon University Pittsburgh, PA, 15213
[24] SP: Tailoring Game State Information Processing and Synchronisation for Improved Playability in Wide Distributed Network Games, Jeremy Brun, Farzad Safaei, Paul Boustead, Telecommunications and Information Technology Research Institute, University of Wollongong, Australia,http://www.smartinternet.com.au/SITWEB/publication/files/18_$$$_88138/P05_006.pdf
[25] COMPARAISON D'APPROCHES DE SIMULATIONS DISTRIBUEES A EVENEMENTS

DISCRETS D'ENTITES SPATIALISEES, Gauthier Quesnel, Raphael Duboz, Eric Ramat.
[26]Probabilistic clock synchronization, Flaviu Christian
[27]Clock Synchronization Algorithms for Network; Measurements ,Li Zhang, Zhen Liu and Cathy Honghui Xia, IEEE INFOCOM 2002
[30] Lamport L., "Time, clocks and the ordering of events in a distributed system.", Communications of the Association of the Computing Machinery, Vol. 21, No. 7, p.558-565 (1978).
[31] Jefferson D. R. "Virtual time", ACM Transactions on Programming Languages and Sytems, Vol. 7, No. 3, p. 404-425 (1985)
[32] http://www.clanservers.com/, Location de serveurs aux joueurs.
[33] www.udpsoft.com/eye , The All Seeing Eye, UDP soft Ltd.
[34] http://www.blizzard.fr/press/040628wow_signup.shtml , BLIZZARD ENTERTAINMENT® ANNONCE L'OUVERTURE DES INSCRIPTIONS AU BETA-TEST EUROPEEN DE WORLD OF WARCRAFT™.
[35] 1500 Archers on a 28.8: Network Programming in *Age of Empires* and Beyond, Paul Bettner, Mark Terrano. Game Developer Conference 2001.
[36] Multiplayer Math, Larry O'Brien. Gamasutra : 29/08/1997
[37] http://brew.qualcomm.com/brew/en/press_room/press_kit.html. Press Kit
[38] Symbian OS Version 8.0, Symbian, SymbianOSv8_funcdesc_2.1.pdf
[39] http://www.dofus.com/, MMORPG Flash.
[40] http://www.macromedia.com/
[41] How to Keep a Dead Man from Shooting, Martin Mauve, Proc. of the 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services 2000.
[42] https://secure.jolt.co.uk/index.php? page=clanserver.php&cpaction=showserver&id=35, Location de serveurs.
[43] Using NTP to Control and Synchronize System Clocks, David Deeths, Glenn Brunette, Sun BluePrints™ OnLine - July 2001. http://www.sun.com/blueprints/0701/NTP.pdf
[44] On the suitability of dead reckoning Schemes 4 games, Lothar Pantel, Lars C. Wolf.
[45] an Efficient Synchronization Mechanism for Mirrored Game Architectures, Eric Cronin, Burton Filstrup, Anthony R. Kurc, Sugih Jamin, Netgames 2002.
[46] IEEE Standard for Distributed Interactive Simulation Application Protocols, IEEE Std 1278.1-1995, Annexe B.
[47] Cheat-Proof Playout for Centralized and Distributed Online Games,Nathaniel E. Baughman Brian Neil Levine, INFOCOM 2001.
[48] Local Lag and TimeWarp : Providing Consistency for Replicated Continuous Applications. Martin Mauve, Jürgen Vogel, Volker Hilt, Wolgang Effelsgerg, IEEE, Février 2004
[49] IEEE Standard for Distributed Interactive Simulation, IEEE Std 1278.
[50] Design and Evaluation of Mimaze, a Multi-player Game on the internet. Laurent Gautier, Christophe Diot, International Conference on Multimedia Computing and Systems, pages "233-236", 1998.
[51] http://www.teamspeak.org/, logiciel de communication vocal.

[52] www.ventrilo.com, logiciel de communication vocal.
[53] RTP: A Transport Protocol for Real-Time Applications, H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, The Internet Society, 2003.
[54] Replication: Optimistic Approaches, Yasushi Saito, Marc Shapiro.
[55] Introduction to Multiplayer Game Programming, 2003, http://www.bookofhook.com/Article/GameDevelopment/MultiplayerProgramming.html
[56] Time Warp, Basic Algorithm, Tutorial, Richard Fujimoto, 1995.
[57] Optimistic Fossil Collection for Time Warp Simulation, Christopher H. Young, Philip A. Wilsey, Computer Architecture Design Laboratory, 29th Hawaii International Conference on System Sciences (HICSS'96) Volume 1: Software Technology and Architecture ,1996.
[58] Distributed Snapshots for Mobile Computing Systems, Adnan Agbaria, William H. Sanders, University of Illinois at Urbana-Champaign, Second IEEE International Conference on Pervasive Computing and Communications,2004.
[59]Temporal Uncertainty Time Warp: An Agent-based Implementation, Roberto Beraldi, Libero Nigro, Antonino Orlando, Francesco Pupo, Simulation: Transactions of the Society for Modeling and Simulation International, Vol. 79, Nr. 10.
[60] "Latency Compensating Methods in Client/Server In-Game Protocol Design and optimization" de Yahn W. Bernier. Valve software. (520 kirkland Way, Suite 200, Kirkland,WA 98033) e-mail: yahn@valvesoftware.com
[61] GPS Synchronization Clock, Model 200, http://www.gpsclock.com/specs.html
[62] GASP, Pelletier Romain, 08/2004, Projet MEGA.
[63] OSP, http://www.orangesmoothie.com