Thèse de doctorat de l'INSTITUT NATIONAL DES TELECOMMUNICATIONS dans le cadre de l'école doctorale S&I en co-accréditation avec l'Université d'Évry-Val d'Essonne

Spécialité :
Informatique

Par
M. Abdul Malik KHAN

Thèse présentée pour l'obtention du grade de Docteur de l'INSTITUT NATIONAL DES TELECOMMUNICATIONS

Communication Abstraction for Data Synchronization in Distributed Virtual Environments Application to Multiplayer Games on Mobile Phones

Soutenue le 17 Juin 2010 devant le jury composé de :

Rapporteurs :

M. Salah Sadou	Maître de conférences (HDR), Université de Vannes
M. Eric Gressier-Soudan	Professeur, CNAM, Paris

Examinateurs :

M. Mourad Oussalah	Professeur, Université de Nantes
M. Guy Bernard	Professeur, Télécom SudParis (Directeur de thèse)
M. Antoine Beugnard	Professeur, Télécom Bretagne (co-Directeur de thèse)
Mme Sophie Chabridon	Maître de conférences, Télécom SudParis (Encadrante)

Thèse Numéro : 2010TELE0011

Acknowledgements

I would like to express my gratitude, appreciation and sincere thanks to my supervisors Dr. Guy Bernard, Dr. Antoine Beugnard and Dr. Sophie Chabridon, for their excellent guidance, helpful and useful discussions, and continuous encouragement which made this work possible. They always helped me with pleasure in the problems that I faced during this work.

I am deeply indebted to all professors and students in my team for their support. I am especially grateful to Michel Simatic for his support both in terms of resources and encouragement.

Deep thanks to my fellow doctoral student Jérôme Sicard. I have always benefited from him through discussion both technically and socially. He has always encouraged me to improve my french language.

I cannot forget the constant encouragment and support of my whole family including my brothers and sisters.

I dedicate this work to my beloved mother whose constant support and prayers are gospel of encouragement for me to keep struggling for ambitions.

Abstract

Multiplayer games users' have increased since the widespread use of the internet. With the arrival of rich portable devices and faster cellular wireless networks, the multiplayer games on mobile phones and PDAs are becoming a reality.

For multiplayer games to be playable, they should be highly interactive, fair and should have a consistent state for all the players. Because of the high wireless network latency and jitters, the issue of providing interactive games with consistent state across the network is non-trivial.

In this report, we propose different approaches for achieving consistency in mobile multiplayer games in the face of high latency and, large and variable jitters. Although absolute consistency is impossible to achieve because information takes time to travel from one place to another because of the limited light speed, our proposed approaches exploit the fact that strong consistency is not always required in the virtual world and can be relaxed in many cases. Based on the underlying network latency and the position of different objects in the virtual world, we decide when to relax consistency and when to apply strong consistency mechanisms. We evaluate our approach by implementing these algorithms in J2ME based games played on mobile phones.

Also, the consistency resolution is dependent upon the choice of the network. Games companies prefer client-server architecture for its control over the users and its simplicity of consistency resolution. But such an architecture has some disadvantages such as a limited scalability and a high latency. P2P architectures on the other hand are open to cheating and have no central command over the game servers. As a second contribution, we propose a hybrid architecture to achieve high interactivity and consistency as well as game services providing companies with a central control over the game.

The algorithms for consistency mechanism are very complex and are often intermixed with the game core logic's code, which makes it hard to program a game and to change its code over time. Our third and last contribution takes the form of an approach to separate consistency mechanisms from the game logic and put them in a distributed component responsible for both consistency maintenance and communication over the network. We call this reusable component a 'Synchronization Medium'. We show the reusability of our synchronization medium by deploying two different multiplayer games on the top of it.

Keywords: Multiplayer Games, Distributed Virtual Environments, Consistency Maintenance, Architecture

Publications relevant to the thesis

- Khan, A. M., Chabridon, S., and Beugnard, A. (2007). Synchronization medium: a consistency maintenance component for mobile multiplayer games. In Armitage, G. J., editor, Proceedings of the 6th Workshop on Network and System Support for Games, NETGAMES 2007, Melbourne, Australia, September 19-20, 2007, pages 99-104. ACM.
- Khan, A. M., Chabridon, S., and Beugnard, A. (2008). A dynamic approach to consistency management for mobile multiplayer games. In CDUR'08 Workshop of NOTERE'08, 8th International conference on New technologies in distributed systems, pages 1-6, Lyon, France, June 23, ACM.
- Khan, A. M., Arsov, I., Preda, M., Chabridon, S., and Beugnard, A. (2010). Adaptable client-server architecture for mobile multi-player games. In DISIO'10: Proceedings of the Workshop on DIstributed SImulation & Online gaming of the SIMUTools Conference, Torremolinos, Malaga, Spain, March 15.
- Khan, A. M., Chabridon, S., and Beugnard, A. (2010). A Reusable Component for Communication and Data Synchronization in Mobile Distributed Interactive Applications, International Workshop on Component and Service Interoperability (WCSI-10), in conjunction with Tools 2010 Federated Conferences, Malaga, Spain, June 29.

Contents

List of Figures

List of Tables

Résumé long

Ce résumé a pour vocation de rendre facilement appréhendable le travail réalisé au cours de cette thèse à une personne francophone. Pour une description détaillée des solutions proposées, la version anglaise est recommandée.

I Introduction

Depuis l'apparition du premier jeu vidéo interactif en 1958, "Tennis for two" de Higgingbotham, les joueurs demandent toujours plus d'immersion, de réalisme et une forte interactivité avec d'autres joueurs. Ceci conduit les industriels (Terraplay, 2000; Unreal,) et les chercheurs à mettre en œuvre des techniques de systèmes répartis déjà connues et de les adapter au domaine des jeux multijoueurs (Natkin, 2003; Griwodz, 2002; Ferretti, 2005; Pellerin, 2010). Depuis quelques années, l'explosion du marché des terminaux mobiles est une opportunité pour une nouvelle génération de jeux multijoueurs sur mobile. Quelques jeux multijoueurs mobiles ont été proposés tels que *Pirates!* (Falk et al., 2001), *SpyGame*, *Human Pacman* (Cheok et al., 2003), *AR-Soccer*, mais certaines limitations techniques freinent encore le confort de jeu et demandent à être considérées avec attention.

Le développement de jeux multijoueurs en réseau est beaucoup plus complexe que pour les jeux fonctionnant sur une seule machine car ils font appel à des techniques supplémentaires telles que la programmation réseau, l'algothmique répartie pour la gestion de l'état global du jeu et l'administration de l'infrastructure réseau. Cependant, plusieurs avantages en résultent :

- l'interaction d'un grand nombre de joueurs potentiellement mobiles et dispersés à travers le monde, utilisant des terminaux très variés;

- la maintenance d'un état persistant du jeu que les joueurs peuvent retrouver après une déconnexion intempestive ou après une pause dans le jeu pour réaliser une autre activité (Okanda and Blair, 2004). Des exemples de jeu fournissant un état persistant sont Everquest (Everquest,), Lineage (Lineage,) ou Ultima Online (Unreal,);

- la possibilité d'utiliser des méthodes marketing adaptées aux mobiles, telles que la publicité par SMS selon les préférences de l'usager, pour inviter les

utilisateurs à participer à un jeu.

I.1 Problématique

Le confort de jeu dans un jeu multijoueur en réseau est optimal lorsque le réseau arrive à se faire oublier et que les délais de communication entre les joueurs sont invisibles. L'objectif poursuivi lors du développement d'un jeu en réseau est donc de favoriser l'immersion des joueurs dans le monde virtuel comme si celui-ci n'était pas réparti physiquement sur plusieurs machines. Un facteur clé est de permettre une évolution en temps réel du jeu, de telle sorte que les joueurs puissent vivre une expérience de jeu similaire à celle qu'ils auraient en étant présents ensemble dans la vie réelle avec une interactivité directe. Par conséquent, le système doit garantir à la fois la satisfaction de contraintes d'interaction en temps réel et le maintien d'un état cohérent du jeu de telle sorte que les joueurs en aient tous la même vision. Ces deux exigences d'interactivité et de cohérence sont dépendantes des caractéristiques du réseau de communication sous-jacent et en particulier de sa latence. Dans les jeux sur réseau filaire, des solutions existent déjà pour masquer la latence qui est de l'ordre de quelques centaines de millisecondes. Il a été montré que pour les jeux les plus réactifs tels que les jeux de tir, une latence supérieure à 250 ms n'est pas tolérée par les joueurs (Pantel and Wolf, 2002) qui en viennent à rejeter le jeu. Dans un réseau sans fil, une latence plus élevée pouvant atteindre plusieurs secondes, le risque de déconnexions et la mobilité des utilisateurs sont autant de contraintes supplémentaires nécessitant la mise en place de solutions nouvelles de synchronisation pour préserver l'interactivité et la cohérence au sein du jeu.

Dans cette optique, l'un des objectifs de cette thèse est de répondre à la question suivante : Comment maintenir la cohérence entre les différents participants à un jeu multijoueur potentiellement mobiles avec risque de déconnexion et en étant confronté à une forte latence réseau pouvant varier grandement ? De plus, l'utilisation d'un terminal mobile s'accompagne de limitations en capacité de calcul et de mémoire et également de limitations en communication en raison du coût attaché à chaque message.

Les mécanismes de maintien de la cohérence, que nous appelons également mécanismes de synchronisation dans cette thèse, font appel à des algorithmes complexes, difficiles à programmer, à mettre au point et à faire évoluer. La préoccupation de maintien de la cohérence peut être considérée comme une propriété extra-fonctionnelle, puisqu'elle ne concerne pas le jeu en lui-même, et il est donc souhaitable de séparer le code des algorithmes traitant de la cohérence de celui de la logique du jeu. De cette manière, le développeur de jeu bénéficie d'une architecture logicielle lui permettant de se concentrer sur le jeu sans avoir à traiter de la gestion de la cohérence et de l'interactivité. De plus, cela favorise la réutilisation des algorithmes de cohérence dans plusieurs applications de jeu et permet de proposer une plate-forme intergicielle intégrant plusieurs algorithmes de cohérence adaptés à des situations différentes.

I.2 Contributions

Nous résumons ci-dessous les trois principales contributions de cette thèse en montrant leur complémentarité chacune considérant un point de vue différent.

1. Du point de vue de l'utilisateur d'un jeu multijoueur, notre contribution permet de diminuer l'impact des délais de communication et de la mobilité en réseau sans fil par l'utilisation des mécanismes spécifiques de maintien de la cohérence améliorant l'immersion dans le monde virtuel.
2. Du point de vue de l'éditeur de jeu, nous proposons une infrastructure multi-serveurs avec serveurs de session, facilement contrôlable et administrable et financièrement attractive.
3. Pour le développeur de jeu, nous offrons une architecture logicielle lui permettant de se concentrer uniquement sur la logique de jeu en déléguant la gestion de la communication et de la cohérence à un tiers.

I.3 Organisation de ce résumé

Dans la section II.1, nous présentons une étude de l'état de l'art sur les trois thématiques qui nous intéressent. Nous discutons tout d'abord des algorithmes de synchronisation utilisés dans les environnements virtuels distribués, en particulier dans les jeux multijoueurs, puis nous présentons les architectures systèmes candidates pour les jeux multijoueurs et nous terminons par une étude des architectures logicielles permettant la séparation des préocupations de synchronisation et de jeu.

Dans la section III, nous décrivons les points importants de nos contributions sur chacune des thématiques précédentes.

La section IV conclut ce résumé et présente des perspectives à cette thèse.

II État de l'art

II.1 Algorithmes de maintien de la cohérence pour les environnements virtuels distribués

Les applications distribuées peuvent être classées en deux grandes catégories, les applications discrètes et les applications continues, selon la manière dont elles évoluent au cours du temps. 1) Les applications discrètes changent d'état à des instants précis à la suite de l'occurrence d'un événement particulier, tel que l'action d'un utilisateur. Une base de données, qui est modifiée par des mises à jour ponctuelles, est un exemple d'application discrète. Pour ce type d'applications, la cohérence est généralement garantie par une approche pessimiste visant à éviter l'apparition de divergence entre les données en maintenant le système dans un état global cohérent à tout instant. 2) Les applications continues ont la particularité d'être modifiées par les actions des utilisateurs mais également par le simple passage du temps. Les jeux vidéo, où un monde virtuel évolue au cours du temps, la réalité virtuelle

comme dans les simulations militaires, les performances musicales ou théâtrales, sont des applications continues. Le fait que ces applications soient distribuées introduit des contraintes supplémentaires sur les communications pour maintenir un bon niveau d'interactivité entre les joueurs et a amené à considérer les applications multi-utilisateurs interactives et distribuées comme un sujet d'étude et de recherche à part entière depuis plus d'une dizaine d'années (IEEE, 1995; Gautier and Diot, 1998; Bouillot, 2006). Le maintien de la cohérence pour les applications continues est le plus souvent géré par une approche optimiste n'imposant pas au système d'attendre un état global cohérent mais le laissant évoluer avec le temps (Palazzi, 2006). Ce critère étant indispensable aux applications étudiées dans cette thèse, nous nous limitons ci-après à une synthèse de l'état de l'art des méthodes optimistes pour le maintien de la cohérence des applications distribuées interactives continues.

Le tableau II.1 présente une analyse de plusieurs algorithmes optimistes de maintien de la cohérence. La prédiction d'état ou DR (Dead-Reckoning) (IEEE, 1995) permet de masquer la latence du réseau en faisant une prédiction de l'état futur d'un objet du monde virtuel en fonction de sa position actuelle, de sa vitesse et de sa direction de déplacement. DR n'implique donc pas de retour arrière mais prévoit un algorithme de convergence pour passer de la position calculée à la position réelle reçue d'un site distant. Les algorithmes TW (Time Warp) (Jefferson, 1985; Lin and Lazowska, 1991; Mauve et al., 2004), TSS (Trailing State Synchronization) (Cronin et al., 2002) et OOS (Optimistic Obsolescence-based Synchronization) (Ferretti and Roccetti, 2005a) sont optimistes et effectuent des retours en arrière en cas de divergence entre les états des participants. Ils sont utilisés principalement avec des architectures de serveurs miroir. La synchronisation par *bucket* ou seau (Steinman, 1990; Steinman, 1991; Gautier and Diot, 1998) est comparable à l'approche OOS car elle ignore certains événements, mais elle ne fait pas de retour en arrière et doit être déployée sur chaque machine dans une architecture pair-à-pair. Les approches de cohérence adaptative (Ikedo and Ishibashi, 2006) et de cohérence relâchée (Li et al., 2004) limitent également les risques d'incohérence en ignorant certains événements, mais ne peuvent garantir qu'un faible niveau de cohérence. *Rendezvous* (Chandler et al., 2004; Chandler and Finney, 2005a; Chandler and Finney, 2005b; Chandler and Finney, 2005c) propose une solution de cohérence faible pour applications mobiles de jeux multijoueurs, mais reste complexe à mettre en œuvre. Nous pouvons conclure à partir de cette synthèse qu'il n'existe pas une approche unique capable de maintenir la cohérence pour différents types d'applications de jeux, et qu'il faut envisager une combinaison de plusieurs approches, voire une adaptation de l'approche utilisée en fonction des conditions de l'environnement et de l'état du jeu.

II.2 Architectures Systèmes pour Jeux Multijoueurs

Le principal obstacle pour des interactions en temps réel dans les applications distribuées est l'inaptitude d'Internet à pouvoir garantir des communications à une faible latence. Les jeux multijoueurs ont des exigences de cohérence forte, au moins à certains moments du jeu. Ceci est difficilement compatible avec la latence non négligeable des réseaux de communication, d'autant plus que le choix de l'architecture système mise en place pour le support du jeu influence directement

Table II.1: Comparaison de différents algorithmes de synchronisation

Approche	Architecture	Suppression de messages	Rollbacks	Déploiement	Compatible pour mobiles	Remarques
DR	Pas de contrainte	Non	Convergence	Client	Oui	—
TW et TSS	Serveur Miroir	Non	Oui	game server	for wired networks	cohérence forte
OOS	Mirrored Server	Oui	Oui	game server	Oui	cohérence faible
Synchronisation par "Bucket"	P2P	Oui	Non	Client	Réseaux filaires	—
Approche Adaptative	Pas de contrainte	Oui	-	Client	–	Cohérence faible
"Relaxed Consistency"	Pas de contrainte	Oui	-	–	–	Cohérence faible
RendezVous	Pas de contrainte	-	Non	Client	Prévu pour mobiles	Cohérence faible

les délais pour transmettre les messages de mise à jour et donc le maintien de la cohérence.

Dans cette section, nous présentons brièvement les architectures systèmes utilisées le plus souvent pour les jeux multijoueurs en réseau filaire, afin de choisir une architecture compatible avec les jeux en environnement sans fil. La figure 2.2.3 présente différents systèmes envisageables pour le déploiement de jeux multijoueurs.

II.2.1 Architecture Client-Serveur

Dans une architecture Client-Serveur, la partie cliente du jeu transmet au serveur les touches actionnées par les joueurs. Cette architecture est montrée à la figure 2.2.3(a).

Le serveur collecte les commandes reçues de tous les clients, calcule le nouvel état du jeu et transmet ensuite un message de mise à jour d'état aux clients. Les applications clientes peuvent alors rafraîchir l'écran du jeu. Dans une telle architecture centralisée, l'état du jeu est simple à maintenir, puisque l'état est géré et mis à jour uniquement par le serveur. De plus, un fournisseur de jeu pourra plus facilement contrôler le jeu dans une telle architecture et gérer la facturation des joueurs. Cependant, la réactivité du jeu est faible car une commande issue d'un client n'est prise en compte qu'après un échange de messages avec le serveur. De plus, un serveur centralisé représente à la fois un point de défaillance unique pouvant paralyser le système et un goulet d'étranglement pour les communications en raison de la largeur de bande limitée d'une machine unique, limitant le passage

Table II.2: Comparaison de différentes architectures pour jeux multijoueurs

Architecture	Algorithme de Cohérence	Contrôle du Jeu	Triche	Latence
Client-Serveur	simple	centralisé	difficile	élevée
Pair-à-pair	complexe	Distribué	facile	plutôt faible
Serveurs Miroirs	relativement simple	Distribué sur réseau privé	relativement difficile	plutôt faible
PP-CA	relativement simple	Distribué	relativement difficile	comparable P2P

à l'échelle en nombre de joueurs. Les besoins en bande passante augmentent de manière quadratique avec le nombre de joueurs (Pellegrino and Dovrolis, 2003a).

II.2.2 Architectures Distribuées

Plusieurs architectures décentralisées peuvent être envisagées pour le support des jeux multijoueurs.

Dans une architecture pair-à-pair, chaque site participant maintient une copie locale de l'état du jeu à partir des messages de mises à jour reçus des autres sites (voir Figure 2.2.3(b)). Le passage à l'échelle est favorisé, les besoins en bande passante augmentent effectivement de manière linéaire avec le nombre de joueurs (Pellegrino and Dovrolis, 2003a), cependant le maintien de la cohérence est relativement complexe et sensible à l'ordre de réception des messages sur les différents sites. La gestion des comptes des utilisateurs elle-même décentralisée est difficile, rendant plus complexe le contrôle du jeu par les fournisseurs de jeu.

(Cronin et al., 2002) propose une architecture avec deux serveurs miroir pour des jeux multijoueurs en ligne. Dans cette architecture (figure 2.2.3(c)), les serveurs de jeu sont répartis géorgraphiquement et les joueurs peuvent se connecter au serveur le plus proche. C'est donc un compromis entre une architecture client-serveur et une architecture pair-à-pair, diminuant la latence des communications mais impliquant une gestion complexe pour maintenir la cohérence de copies multiples

Dans l'architecture PP-CA (Pellegrino and Dovrolis, 2003b) (figure 2.2.3(d)), les joueurs échangent des mises à jour en communiquant directement entre eux et avec un arbitre central qui est chargé de détecter les incohérences dans l'état du jeu. Si nécessaire, il envoie un message de synchronisation à tous les sites, qui vont éventuellement faire un retour en arrière.

Le tableau II.2 effectue une comparaison de différentes architectures candidates pour les jeux multijoueurs.

II.3 Séparation de la préoccupation de cohérence de la logique du jeu

Dans la section II.1, nous avons vu qu'une seule approche ne suffit pas et que plusieurs approches de maintien de la cohérence doivent être combinées pour atteindre une vue globale de l'état du jeu. Ces algorithmes sont très complexes, et donc difficiles à programmer, à mettre au point et à maintenir. Il apparaît donc souhaitable de séparer le code de ces algorithmes de la logique de jeu. Nous présentons ci-après les travaux sur une abstraction de communication (Cariou et al., 2002) servant de base à notre contribution qui sera présentée dans la section III.3.

Le concept de composant de communication ou *médium* (Cariou et al., 2002) permet de séparer la logique d'interaction du code fonctionnel d'un composant. Un médium représente donc une réification d'un service ou protocole d'interaction, communication ou coordination en tant que composant logiciel. L'avantage de cette architecture est qu'un même médium peut être réutilisé dans différents types d'applications. Le médium étant un composant logiciel, il spécifie les services qu'il offre et ceux qu'il utilise. Le médium peut prendre différentes formes selon le niveau auquel il est considéré. Il existe en tant que spécification pour décrire l'abstraction de communication qu'il réifie, et aux niveaux implémentation et déploiement. Il est ainsi possible de manipuler une abstraction de communication de haut niveau durant toutes les étapes du cycle de vie du logiciel.

II.4 Conclusion

Nous avons présenté une brève étude de l'état de l'art sur trois aspects différents des jeux multijoueurs : le maintien de la cohérence, les architectures systèmes et la séparation entre la logique de jeu et la gestion de la synchronisation des joueurs. La grande majorité des algorithmes de maintien de la cohérence considère un réseau filaire et seuls très peu de travaux commencent à s'intéresser aux jeux multijoueurs sur terminaux mobiles en prenant en compte une latence élevée et des risques de déconnexion. Il en est de même pour les architectures systèmes qui sont pour l'instant principalement dédiées aux environnements connectés. Par ailleurs, un médium de communication peut être enrichi pour séparer les aspects de gestion de la cohérence de ce qui concerne la logique de jeu ceci afin d'intégrer la gestion de la cohèrence au médium. Nous présentons dans la section suivante nos contributions sur les trois aspects considérés.

III Contributions

III.1 Algorithmes de Synchronisation pour Jeux Multijoueurs sur Téléphone Mobile

Dans cette section, nous proposons plusieurs mécanismes pour maintenir la cohérence dans un jeu multijoueur en présence d'un réseau sans fil à forte latence et non fiable, sur des terminaux mobiles ayant des capacités de calcul et de mémoire limitées.

III.1.1 Vers une Approche Adaptative

Nous énumérons ci-après plusieurs observations motivant la proposition d'une approche adaptative.

Observation 1 Au cours d'une session de jeu en multijoueur, des incohérences apparaissent entre les joueurs en raison des délais de communication réseau. Les programmeurs de jeu font en général une estimation de ces délais pour compenser les conséquences d'un message arrivant tardivement. Cependant, en raison de la gigue dans un réseau de communication, en particulier dans un réseau sans fil, ces délais peuvent varier grandement. La variabilité des délais de communication a un impact direct sur l'approche de maintien de la cohérence utilisée. Les contraintes de cohérence risquent de ne plus être vérifiées si ces variations dynamiques ne sont pas prises en compte. Par conséquent, il est nécessaire d'observer le comportement du réseau au cours d'une session de jeu et de compenser dynamiquement les fluctuations constatées.

Observation 2 Dans un jeu multijoueur, de nombreux types d'objets évoluent dans un monde virtuel avec des vitesses et des directions de déplacement différentes. Ainsi, leurs besoins de cohérence sont généralement différents. Dans un jeu de tennis, par exemple, la vitesse de la balle est bien supérieure à celle du déplacement d'un joueur, et ces deux objets ne doivent donc pas être traités de la même manière.

Observation 3 Les besoins de cohérence d'un objet ne dépendent pas seulement de sa vitesse, mais aussi de sa position dans le monde virtuel. Dans l'exemple d'un jeu de course de voitures, la cohérence doit être gérée de manière stricte quand les voitures sont proches les unes des autres et qu'elles arrivent près de la ligne d'arrivée. Par contre, une cohérence relâchée est suffisante dans une autre partie du circuit.

Ces observations indiquent qu'un algorithme de maintien de la cohérence doit tenir compte à la fois de la latence des communications réseau et du contexte de jeu.

Retard local adaptable À la réception d'un message contenant des informations sur un objet telles que sa position, vitesse, et direction de déplacement, cet objet est à une certaine distance, potentiellement nulle, d'une cible ou d'une entité particulière dans le monde virtuel que nous appelons pivot. La distance entre l'objet et le pivot est essentielle pour déterminer les contraintes de cohérence et d'interactivité à ce moment précis dans le temps.

Nous introduisons un retard local (*local lag* (Mauve et al., 2004)) avant la prise en compte d'une commande locale ou distante selon les trois règles suivantes :

1. Proximité du pivot : Si l'objet pour lequel un message de mise à jour est reçu d'un utilisateur distant se rapproche du pivot, la valeur du retard local est diminuée. Si l'objet s'éloigne du pivot, le retard local est augmenté jusqu'à une limite donnée par $Local\text{-}lag_u$. Cette augmentation peut être continue en fonction du mouvement de l'objet, ou peut être discrète et effectuée lors de l'entrée dans une zone particulière (Santos et al., 2007). Le taux de variation de la valeur du retard local dépend de l'application et doit être spécifié par le programmeur lors du développement du jeu. Cela se fait au travers d'un composant responsable de la gestion de la cohérence tel que nous le décrivons dans le chapitre 6 de la thèse.
2. Charge réseau : La valeur du retard local varie également en fonction de la charge du réseau. Lorsque le nombre de messages arrivant au-delà d'une certaine limite de temps dépasse un niveau donné par N_d, le retard local est augmenté. Cette augmentation du nombre de messages en retard peut être due à la gigue dans les délais de communication. La valeur du retard local est proportionnelle à la latence réseau, mais il n'est pas possible de l'augmenter au-delà d'une certaine limite car cela affecterait la réactivité du jeu. Cette limite dépend du rythme du jeu et peut être spécifiée par le développeur de jeu.
3. Type d'objet : La valeur de retard local peut varier selon le type d'objet. Par exemple dans un jeu de tennis, la balle aura un retard local associé inférieur à celui d'un joueur afin de privilégier la réactivité du jeu pour la prise en compte d'une action sur la balle (Zhou and Shen, 2007). Le moyen de spécifier le retard local associé aux différents types d'objets manipulés dans le jeu est détaillé au chapitre 6.

Prédiction d'état adaptable La prédiction d'état (IEEE, 1995) se base sur un seuil d'erreur entre la position réelle d'un objet et la position calculée pour déterminer si un message de notification indiquant la position réelle de l'objet doit être émis ou non. Ce seuil d'erreur doit être dépendant de la position et de l'environnement des objets, c'est-à-dire *quel* objet se trouve à *quelle* position. Ainsi, les contraintes de cohérence peuvent être relâchées selon la position de l'objet dans le jeu. Nous définissons pour ce faire la notion de régions critiques.

Région critique Une région critique est une zone dans le jeu où une cohérence stricte est nécessaire afin que tous les joueurs aient la même vision de cette région. Dans une région critique, une incohérence entre les joueurs peut se traduire par une inéquité du jeu (Zhou and Shen, 2007). Dans une telle région, nous proposons d'augmenter dynamiquement la fréquence d'émission des messages de notification en diminuant le seuil d'erreur de la prédiction d'état. Par conséquent, il devient possible de relâcher la cohérence en dehors des régions critiques.

De manière complémentaire à une adaptation aux régions critiques, la prédiction d'état peut utiliser des seuils d'erreur différents selon les objets en fonction de leur mouvement et de leur importance dans le jeu. De même que pour les valeurs du retard local, les seuils doivent être spécifiés par le programmeur de jeu comme nous le décrivons au chapitre 6.

III.1.2 Réduction des retours en arrière

Lors de la réception d'un message en retard, c'est-à-dire ayant une estampille inférieure à celles des messages déjà traités, il peut être nécessaire d'effectuer un retour en arrière. Nous proposons dans cette section une approche permettant de diminuer le nombre de retours en arrière pour améliorer le confort de jeu. Nous nous basons sur la notion d'*actions critiques* qui sont des événements sensibles impactant le résultat du jeu. Les messages de mise à jour notifiant d'une action critique doivent absolument être pris en compte et ne peuvent être ignorés. Par ailleurs, une annulation de ces messages par un retour en arrière peut avoir un impact négatif sur la jouabilité. Cependant, nous montrons qu'en combinant les notions de *région critique* et d'*action critique*, il devient possible d'ignorer certaines actions critiques afin d'éviter un retour en arrière.

Action critique Une action critique est une action concernant une entité donnée mais qui affecte d'autres entités dans le jeu. Par exemple, une commande de tir peut augmenter le score du joueur ayant tiré s'il a touché la cible, mais peut en même temps affecter l'état de la cible. La bonne livraison d'un message relatant une action critique est primordiale pour l'état du jeu. Cependant, nous considérons que lorsque ces actions ne se produisent pas dans une région critique, et que les conditions réseau deviennent mauvaises (avec une latence très élevée), ignorer une action critique arrivée en retard permet d'éviter un retour en arrière.

Corrélation faible et critique Le concept d'obsolescence (Ferretti and Roccetti, 2004b) énonce qu'un événement arrivant en retard alors qu'un événement émis après lui (i.e. ayant une estampille supérieure) a déjà été traité, est devenu obsolète. Avant de décider si ce message peut être ignoré, il faut alors déterminer s'il est corrélé ou non avec un message le précédant. Dans le cas où une corrélation existe, il devient nécessaire de revenir en arrière à l'instant où ce message aurait dû être traité, puis de prendre en compte le message tardif et enfin de re-traiter les messages annulés. Cependant, (Ferretti and Roccetti, 2004b) ne propose pas de mécanisme pour calculer la corrélation entre deux événements. (Xiang-bin et al., 2007) definit deux événements comme étant corrélés s'ils sont associés avec le même objet.

Nous proposons les deux concepts nouveaux de *corrélation faible* et de *corrélation critique* en prenant en compte les notions de région et d'action critiques. Ainsi, un message en retard ne peut être considéré comme obsolète que s'il ne concerne ni une région critique ni une action critique. Dans le cas contraire, il doit toujours être pris en compte par le jeu.

La décision d'un retour en arrière est ainsi prise pour les événements corrélés de manière critique tandis que les événements faiblement corrélés peuvent être ignorés.

Notre contribution se présente ainsi sous la forme d'un algorithme dynamique combinant les différents mécanismes que nous venons de décrire à savoir : retard local adaptable, prédiction d'état adaptable et prise en compte de l'obsolescence et du niveau de corrélation des événements obsolètes afin d'éviter les retours en arrière. Cet algorithme est détaillé à la section 3.3.5.

III.1.3 Évaluation des Algorithmes de Synchronisation

Nous avons évalué les algorithmes de synchronisation que nous proposons en effectuant des mesures au cours de l'exécution de plusieurs jeux simples que nous avons développés en J2ME sur la plate-forme GASP (Pellerin et al., 2005; Pellerin, 2010). Nous avons exécuté le jeu sur des téléphones mobiles Nokia N93 communiquant sur un réseau Wi-Fi avec le serveur GASP.

Concernant l'approche avec prédiction d'état adaptable (*adaptableDR*), nous avons utilisé un jeu de course de voitures où une région critique est définie au niveau de la ligne d'arrivée. Nous avons mesuré l'écart entre la position réelle d'une voiture pilotée par un joueur A et sa position telle qu'elle est affichée à distance sur l'écran du joueur B. Nous avons comparé les approches *adaptableDR* et *simpleDR* qui utilise un seuil d'erreur fixe. Au cours des expérimentations, nous introduisons des variations dans les délais de communication en retardant les messages à l'émission. Les résultats d'évaluation montrent qu'en adaptant le seuil d'erreur utilisé par la prédiction d'état, il est possible de limiter les écarts les positions réelle et calculée. De plus, nous montrons que le nombre de messages échangés avec l'approche *adaptableDR* est inférieur de plus de 20% à l'approche *simpleDR*.

Pour évaluer l'impact de la prise en compte du niveau de corrélation entre un événement obsolète et un événement déjà traité, nous avons compté le nombre de retours en arrière nécessaire et le nombre de messages en retard. Nous avons pour cela développé un jeu de ballon à 2 joueurs. La règle du jeu consiste soit à mettre un but en lançant le ballon dans une zone prédéfinie, soit à toucher le joueur adverse avec le ballon. Trois régions critiques sont ainsi délimitées : une région rectangulaire pour la zone de but et une région circulaire autour de chaque joueur correspondant à la zone dans laquelle le joueur peut être touché. Nous avons comparé les approches avec corrélation critique et faible et l'approche TimeWarp dans laquelle la corrélation entre les événements n'est pas considérée. Il apparaît que la prise en compte de la corrélation permet de limiter de manière très importante le nombre de retours arrière. Moins de 20% des messages en retard induisent un retour arrière lorsque la corrélation est prise en compte.

III.2 Architecture Système pour Jeux Multijoueurs

Nous avons discuté dans l'état de l'art, les propriétés de différentes architectures systèmes pouvant être envisagées pour les jeux multijoueurs sur mobile. Une architecture client-serveur a l'avantage de faciliter le contrôle et le maintien de la cohérence du jeu, mais elle ne permet pas de passer à l'échelle et de gérer un grand nombre de clients, de même que l'interactivité dans le jeu est limitée par les délais de communication des réseaux sans fil. Dans un système pair-à-pair, l'interactivité entre les joueurs est meilleure mais au détriment de la facilité de contrôle par l'opérateur du jeu pour en particulier éviter la triche.

Nous présentons dans cette section une architecture multi-serveurs (voir Figure 5.2.1) avec plusieurs serveurs de session et un serveur d'administration du jeu pour lever les verrous suivants liés aux jeux multijoueurs sur mobile :

1. Faible contrôlabilité des serveurs de jeux en architecture multi-serveur

2. Déconnexions fréquentes
3. Maintien simultané de la cohérence et du confort de jeu

Le serveur d'administration héberge la gestion des comptes des joueurs et de la facturation ainsi que la partie cliente du code du jeu. La gestion des joueurs étant centralisée, cela facilite le contrôle du jeu. Les serveurs de session, connectés en mode pair-à-pair entre eux, et connectés chacun avec le serveur d'administration, héberge la partie serveur du code du jeu.

Un joueur voulant jouer à un jeu à partir de son mobile se connecte au serveur d'administration et télécharge la partie cliente du jeu. Lorsqu'il souhaite rejoindre une session de jeu, il en demande l'autorisation au serveur d'administration. Celui-ci choisit un serveur de session de manière à minimiser les délais de communication pour le joueur, par exemple en raison de la proximité géographique, et l'avertit de l'arrivée d'un nouveau joueur.

Pour permettre la continuité d'une session de jeu en cas de déconnexion, nous proposons de mettre en place un imitateur (*mimicking engine*) pour reproduire le comportement des joueurs et de le déployer côté client sur le terminal mobile et côté serveur sur le serveur de session alloué au joueur. Lorsqu'une déconnexion est détectée, la logique de jeu interagit avec l'imitateur pour en quelque sorte remplacer les joueurs distants avec lesquels la connexion a été rompue. Le comportement de l'imitateur est fonction, de l'état du jeu au moment de la déconnexion, du comportement précédent des joueurs et de règles pour prédire les état suivants. Ces règles sont dépendantes de l'application de jeu et doivent être définies lors de la conception du jeu. La logique de jeu peut également prendre en compte les événements de déconnexion en les intégrant de manière naturelle dans le jeu, comme, par exemple, de montrer une voiture à l'arrêt aux stands dans une course de voiture lorsque le joueur pilotant cette voiture se retrouve déconnecté.

En ce qui concerne le maintien de la cohérence entre les joueurs, mais sans que cela se fasse au détriment du confort de jeu, nous proposons une approche hybride partagée entre le client et le serveur et adaptable dynamiquement. En effet, la responsabilité de la gestion de la cohérence peut migrer dynamiquement entre le client et le serveur en fonction des conditions réseau et de l'entrée ou non dans une région critique dans le jeu.

Nous avons évalué cette approche hybride avec un jeu de course de voitures et nos résultats montrent qu'une meilleure cohérence, en d'autres termes une estimation de la position d'une voiture par le joueur distant très proche de la position réelle, est obtenue avec des mécanismes sur le client lorsque la latence devient grande, à savoir supérieure à 2s. De plus, l'approche associant des mécanismes de maintien de la cohérence à la fois du côté client et du côté serveur donne de meilleurs résultats que l'approche uniquement sur le serveur dès que la latence devient élevée.

III.3 Un Médium de Synchronisation

Il peut être nécessaire de combiner plusieurs algorithmes de synchronisation pour assurer la cohérence entre les joueurs. Ces algorithmes sont complexes et difficiles à programmer. Le code de maintien de la cohérence étant souvent entremêlé avec le

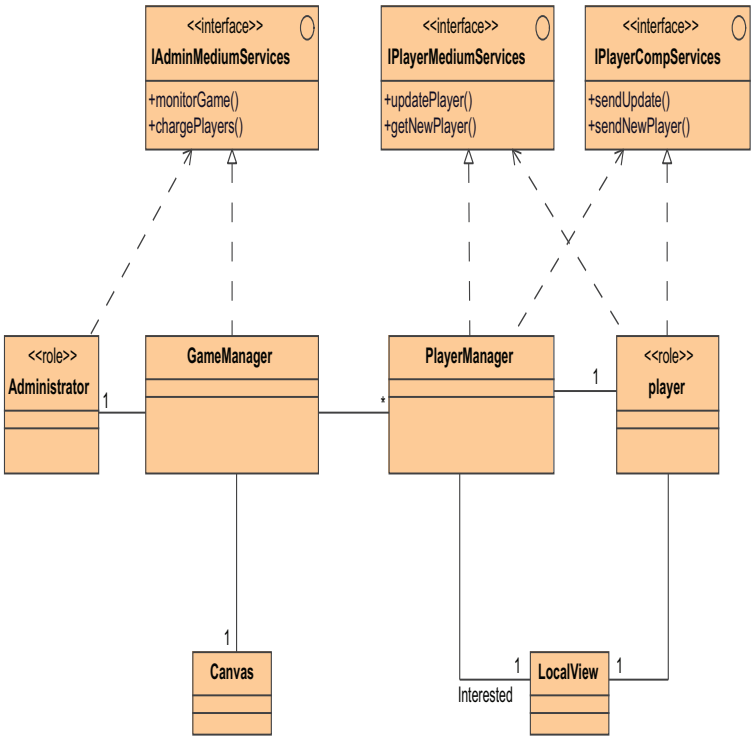


Figure III.1: Modèle du Médium de Synchronisation avec Gestionnaires de Rôle

code de la logique de jeu, ce dernier devient difficile à maintenir et à faire évoluer. Il est donc souhaitable de séparer le code des algorithmes traitant de la cohérence, qui peut être considéré comme extra-fonctionnel, de celui de la logique du jeu et de l'isoler dans un composant spécifique. Nous étendons ainsi le concept de médium de communication (Cariou et al., 2002) en lui intégrant des algorithmes de synchronisation pour en faire un médium de synchronisation.

L'objectif d'un médium de communication est de séparer pour un composant la partie traitant des interactions avec d'autres composants de la partie fonctionnelle. Le médium est ainsi une réification d'un protocole, service ou système d'interaction, de communication ou de coordination. Ce principe évite l'adhérence du médium à une application donnée et facilite donc sa réutilisation pour différentes applications.

Nous présentons sur la figure III.1 les classes et interfaces d'un médium de synchronisation.

Le composant *Player Manager* représente le médium sur chaque client et offre/requiert des services demandés/offerts par le client. Il constitue l'interface avec le client résidant sur le terminal du joueur et utilise les services offerts par le médium via l'interface *IPlayerMediumServices* comme l'arrivée d'un nouveau joueur ou la réception de messages de mise à jour en provenance des autres joueurs. Ces services sont fournis au travers des méthodes *getNewPlayer()* et *updatePlayer()*. Pour

interagir avec le client, le médium utilise les services définis dans l'interface *IPlayerCompServices* dédiée aux échanges de données avec le client.

Game Manager correspond à la partie serveur de médium traduit par le rôle *Administrator*. Il utilise les services offerts par *IAdminMediumServices* comme par exemple la prévention de la fraude, la facturation, etc... La classe *Canvas* représente les données générales du jeu sur le serveur et la classe *LocalView* correspond à une partie des données du jeu dont le joueur a besoin en local sur le client pour le faire fonctionner correctement.

Ce médium de synchronisation est une version générique. Il peut être décliné de différentes manières suivant les algorithmes de synchronisation que l'on veut utiliser. Il est ainsi possible d'ajouter l'interface *IDeadReckoningServices* au composant *PlayerManager* pour incorporer l'algorithme de dead-reckoning ou encore les classes *CalculateLocalLag* et *CalculateLocalVector* pour l'algorithme de retard local ou *local lag*.

L'agrégation de tous les *Role Managers* de tous les clients et serveurs constitue le médium. De plus un *Role Manager* peut être composé de sous-composants communiquant avec le jeu et responsable d'une préoccupation spécifique, telle que la gestion des régions critiques. L'architecture présentant les sous-composants que nous proposons est détaillée à la figure 6.5.7 dans la partie principale de la thèse.

III.3.1 Évaluation de l'Approche avec Médium de Synchronisation

Nous avons évalué l'approche par médium de synchronisation en termes d'une part de facilité de développement et d'autre part de performances en le mettant en œuvre pour le développement de deux jeux, un jeu de guerre de l'espace et un jeu de bataille de tanks.

Il est à noter que le jeu de tanks a été développé par une étudiante, ne connaissant pas au préalable le médium de synchronisation, au cours d'un projet de fin d'études ingénieur dans une démarche de validation externe de nos travaux.

Il est apparu qu'une fois les concepts manipulés par le médium de synchronisation appréhendés par le développeur, les efforts d'implémentation sont réduits au niveau du nombre de classes à définir et du nombre de lignes de code à implémenter. Par ailleurs, l'impact du médium sur les performances du jeu reste négligeable et se limite à un surcoût de moins de 5ms par message.

IV Conclusion et Perspectives

Cette thèse présente un travail sur trois aspects complémentaires pour faciliter la réalisation de jeux multijoueurs sur mobile.

1. Algorithmes de synchronisation : Nous avons proposé une approche adaptative pour le maintien de la cohérence dans les environnements virtuels distribués tels que les jeux multijoueurs tenant compte des conditions du monde virtuel et du réseau de communication, et permettant d'adapter les valeurs

de seuil utilisées pour le retard local et la prédiction d'état. Nous avons introduit les notions de région critique, où une cohérence forte est nécessaire, et de corrélation faible et critique pour la gestion des événements en retard, ce qui permet de moduler le niveau de cohérence au cours du jeu. Nous avons mené des évaluations sur plusieurs jeux sur téléphone mobile montrant que l'interactivité et l'équité dans le jeu étaient améliorées.

2. Architecture système : Nous avons présenté une architecture avec plusieurs serveurs de session connectés en pair-à-pair et interagissant avec un serveur d'administration du jeu répliqué. En cas de déconnexion, un mécanisme d'imitation du comportement d'un joueur est mis en œuvre pour permettre au reste des joueurs de continuer la partie.

3. Architecture logicielle : Nous avons défini un médium de synchronisation qui est un composant réutilisable chargé de la gestion des communications et du maintien de la cohérence. En permettant une séparation des préoccupations entre ce qui est propre au jeu et ce qui est lié à la synchronisation, le développement du jeu devient plus simple et sa maintenabilité et son évolutivité en sont améliorées. Le médium a été mis en œuvre pour le développement de trois jeux attestant de sa réutilisabilité.

Ce travail pourra être poursuivi selon plusieurs directions. À court terme, des évaluations avec un grand nombre de joueurs permettront de mesurer le passage à l'échelle de nos algorithmes de synchronisation et également de vérifier le bon comportement de l'architecture système proposée. Par ailleurs, nous souhaitons déployer plusieurs variantes du médium de synchronisation, chacune offrant une stratégie de synchronisation particulière pouvant être sélectionnée dynamiquement à l'exécution. Nous envisageons également de proposer un médium multi-réseau capable de gérer les communications à travers plusieurs réseaux tels que les réseaux WIFI, Bluetooth et 3G. Finalement, nous souhaitons généraliser notre approche à d'autres types d'applications distribuées et interactives en offrant des interfaces communes de communication et de synchronisation.

Chapter 1

Introduction

Since the introduction of the first computer game, called *OXO*, developed in 1952, computer games have attracted millions of people throughout the years. According to estimates (Carless, 2006), online games market will hit a record $13 billion as compared to 3.4 billion dollars in 2005, surpassing the revenue provided by box-office tickets. The first interactive video game, "tennis for two" by Higgingbotham (Brookhaven National Laboratory, a US nuclear research lab in Upton, New York) was developed in 1958. The interactive experience of that non-commercial game immediately gained a vast popularity and traced the path for the horde of commercial descendants that have invaded arcades and homes till these days. Nowadays, the emerging game market pushes the industry and researchers to provide distributed solutions that give users with advanced online gaming applications able to support them during sophisticated game sessions (Natkin, 2003; Griwodz, 2002; Ferretti, 2005; Pellerin, 2010). With the feeling of immersion required by the players participating to an online game, networked and distributed technologies now come into the game arena. Keeping this in view, many software companies are now interested in developing novel commercial software platforms to support networked multiplayer games (Terraplay, 2000; Unreal,).

With the increasing number, capabilities and innovative use of cell and web phones, the mobile game industry will exploit this multi billion-dollar market to outsell a new generation of multiplayer networked games (Akkawi et al., 2004a; Akkawi et al., 2004b). In fact, the mobile games market is estimated to reach around $6.8 billion by 2013 in the United States only. The mobile communications industry is meeting game developers to devise effective strategies able to engage billions of consumers worldwide. Already a number of new multiplayer games for portable devices have been developed, such as *Pirates!* (Falk et al., 2001), *SpyGame*, *Human Pacman* (Cheok et al., 2003), *AR-Soccer* etc. Now, the trend is that of providing mobile users with complex exciting networked multiplayer games.

On the other hand, as games are becoming more and more simulation oriented, it is easy to envisage that also other types of applications will converge to the use of the new game technologies. Such applications that may take advantage of technical solutions introduced by game developers and researchers include simulations related to medical surgery, military simulations, e-commerce etc.

Many works have been carried out concerning, for example, new projects aimed at developing combat video games to enhance strategic, combat, and decision-making skills of military commanders (Rhyne, 2002a; Kumagai, 2001). Furthermore, collaborative applications' researchers are developing augmented reality interfaces and explore how immersive collaborative virtual environments might support group interaction (Billinghurst and Kato, 2002). Finally, surgery simulations and video games share a quest for realistic object behaviour and high-quality images (Wagner et al., 2002). Summing up, more and more non-game applications adapt and use elements of computer games to enable the creation of compelling user experiences in several domains (Tsang et al., 2003).

The above considerations explain the importance of devising new software solutions that provide support to networked multiplayer games. In essence, this emerging game market is characterized by a growing demand for scalable responsive strategies able to provide players with the feeling of full immersion into the virtual game world during the game evolution. The point is that the development of networked multiplayer games is more complex than classic stand alone games, as they require a significant level of network programming, state maintenance and administration of a network infrastructure to support online gaming sessions (Kawahara et al., 2004). Several advantages may derive from such kind of networked multiplayer games including, for example:

- the possibility to enable a large amount of dispersed players to interact with each other, also by means of a plethora of portable devices, such as laptops, cell phones, PDAs or game consoles;

- the possibility to maintain a persistent state of the game even when users experience high latency and jitters, disconnect because of a temporary network failure or want to perform other activities (Okanda and Blair, 2004); examples of games that already provide persistent virtual worlds are Everquest (Everquest,), Lineage (Lineage,) and Ultima Online (Unreal,);

- the possibility of enabling proactive advertising/advising software mechanisms employed to inviting users to play the game.

Keeping in view the above discussion, we now present the problems dealt with in this thesis.

1.1 Problem Presentation

The main objective to pursue when developing a networked multiplayer game is that of offering the players a more realistic representation of the virtual world. The idea is to have the real player immersed into the virtual world as if this virtual world is not distributed physically. To achieve this objective, a dominant factor is that of ensuring a real-time evolution of the game, so that players may enjoy a game experience which is similar to real-life gaming. This factor provides remote users with

full interactivity. In short, the system must be able to guarantee the satisfaction of real-time requirements, also ensuring that the consistency of the game state is maintained. These two requirements, i.e. interactivity and consistency, are dependent on the underlying network latency. In wired-networks, where latency ranges between 100-300 milliseconds, the issues of consistency and interactivity have been discussed and worked upon for quite some time.

With the arrival of wireless telecommunication data networks such as GPRS and 3G, and with the increase of the use of mobile phones, game users have steadily started playing on their mobile terminals which give them more freedom in terms of mobility and location. In wireless networks, because of the higher latency, the frequent disconnections and the mobility of the user(s), the issues of consistency and interactivity become more crucial as compared to the wired networks. The GPRS network suffers from high latency for data transmission (1000-2700ms round trip time (McCaffery and Finney, 2004)), making timely real-time communications between collaborative gamers close to impossible within the scope of any real-time game. While the emerging 3G networks are much improved (400-800ms (Frécon and Stenius, 1998)), it still remains above the tolerance of typical real-time multiplayer games which is below 250 ms (Pantel and Wolf, 2002). Although, 4G networks promise to provide much more faster communication than the current 2.5 and 3G networks (Balakrishnan and Sadasivan, 2007), it is yet to be world-wide implemented and tested for real-life virtual environments. Even with fast development of communication technologies, the network latency cannot be avoided as the information cannot flow at a speed more than that of light. Thus, the issues of consistency and interactivity still remain a bottleneck for wired multiplayer games in general and mobile multiplayer games in particular.

Keeping in view the high latency of wireless networks, as discussed above, one of the objectives of this thesis is to focus on the following question: How to maintain the consistency among different mobile multiplayer games' players in the face of high network latency and user's mobility when frequent disconnections and network jitters may occur. With this problem, also come the limited resources of portable terminals such as limited memory and processing power. Apart from this, a player, while playing on a mobile phone, may have to pay for each message he/she shares with other remote players in the game. This in mind, in this thesis report, we focus our attention on finding viable ways to maintain consistency among mobile players playing a multiplayer game with limited resources and a costly, high latency and unreliable network. Based upon the knowledge of the requirements of the game and the network conditions, we propose a consistency maintenance approach adaptable to high latency and jitters, to achieve consistency in multiplayer mobile games. A consistency maintenance mechanism is also referred to as a synchronization mechanism; therefore, in this document, we will use the terms *consistency maintenance* and *synchronization* interchangeably.

The network architecture and the infrastructure upon which the game is deployed directly influence the consistency and the interactivity of the game because 1) the latency varies from architecture to architecture as it is dependent upon how many nodes a message has to pass before arriving at its destination, 2) there is a difference whether the consistency is resolved at a central place where this can

be managed more simply or the consistency resolution is distributed and solved in a more complex manner. Hence, the choice of the network architecture directly impacts the consistency maintenance within a multiplayer game. Also it also conditions the way the game companies control the game. Game companies prefer a platform that gives them more control over the game. We, therefore, in this report also discuss different architectures for multiplayer games and how they are related to consistency maintenance approaches. We propose an architecture for multiplayer games on mobile phones using cellular networks which could provide state consistency as well as control over the game.

The mechanisms for consistency maintenance use algorithms that are very complex and are, therefore, hard to program. Also because of their complexity, the evolution of the game code becomes difficult during the course of time. It is thus desirable to separate the code of these algorithms from the game logic. The objectives are to be able to re-use these algorithms when developing a new game and to provide a middleware platform which can offer many different algorithms suitable to different situations. In separating the game logic from the consistency maintenance algorithms, the focus is to have an architecture where the developer(s) of the game concentrate(s) only on the game logic, and is/are not bothered by the consistency and interactivity issues. The point here is that latency is basically a network problem and is not related to the game logic and hence should be handled separately. As a third contribution, we propose an architecture for separating the game logic from the consistency algorithms.

Hence our contributions cover three different but interrelated points of views of the game.

1. From the point of view of a multiplayer game user, we contribute to lessen the efforts of maintaining a consistent multiplayer game state for a fair gameplay so that players can really immerse in the virtual world.
2. From the game provider's point of view, we propose a multiplayer platform that is more profitable, controllable and attractive.
3. From the game developer's angle, we propose a software architecture that lets the developer(s) to focus on the core logic of the game only and leaves the communication and synchronization aspects to a third party.

1.2 Organization of the Thesis

The first part of the thesis is dedicated to a study of the state of the art on the synchronization aspect in multiplayer games. In the chapter 2, we present a literature survey related to the above three requirements. In section 2.1, we discuss different synchronization algorithms used in distributed virtual environments, especially multiplayer games. Then, in section 2.2, we discuss different system architectures used for multiplayer games. In section 2.3 we present a literature study on candidate software architectures for the separation of synchronization algorithms from the core application logic.

The part 2 of this thesis report presents our contribution.

Chapter 3 describes our first contribution. We discuss our observations for synchronization in multiplayer games. Based upon these observations, we propose a dynamic approach suitable for multiplayer games in high latency networks. We then build on this dynamic approach and propose an algorithm to reduce the number of rollbacks in case of a high and varying network latency.

In chapter 4, we present the results of our experimental evaluation of the approaches we propose in chapter 3.

Chapter 5 is related to our second contribution. In this chapter we present our study of the network architectures used for multiplayer games. Based upon our observations, we propose a distributed multi-servers architecture for mobile multiplayer games. We call this architecture session server architecture. Because of the high latency of wireless networks, the application of synchronization mechanisms only at the game server is not sufficient (as is done by many approaches), so in this chapter we present an adaptable consistency mechanism which switches between client and server sides depending upon the need of the situation. We present our evaluation of this approach in the same chapter.

In chapter 6, we present our third and final contribution. We propose an approach for the separation of synchronization algorithms from the core application logic and their insertion into a reusable communication component. We call this component the *synchronization Medium*.

Chapter 7 presents our evaluation of the synchronization medium from performance as well as re-usability points of view.

In the last part of the thesis, the chapter 8 concludes this report and presents an overview of our future perspectives.

Part I

State of the art

Chapter 2

State of the Art: Synchronization In Multiplayer Games

In this chapter we first discuss the state of the art for the research on synchronization algorithms used in interactive distributed applications such as multiplayer games and military simulations. Then in section 2, we make a study of different system architectures used by networked multiplayer games. In section 3, we discuss works related to the separation and re-usability of synchronization algorithms in multiplayer games.

2.1 Consistency Maintenance Algorithms in Distributed Virtual Environments

In this section, we discuss a literature study related to different consistency maintenance algorithms used in distributed applications. From the consistency maintenance point of view, distributed systems can be divided into discrete and continuous systems. 1) Discrete systems are where the state of a system changes in discrete steps after an event (such as a user action) has occurred and where the time has no impact upon the state of the system. Distributed databases is an example of discrete systems. In this type of systems, conservative approaches for consistency maintenance can be used in which the system advances only after ensuring a global consistent state. 2) Continuous systems, such as multiplayer games, are those where the state of a system changes not only with the users' actions, but also evolves with the passage time. For maintaining consistency in such type of systems, an optimistic approach can be used where the system does not have to wait for achieving a consistent global state, but is evolving with time.

As in this report we are dealing with continuous applications, particularly multiplayer games, in the next section we give just a very brief overview of the con-

servative approach and then in section 2.1.2 we discuss, in detail, the optimistic approach which can be used in distributed continuous virtual environments.

2.1.1 Conservative Approach

The conservative approach is a first attempt to solve the consistency problem in distributed systems. In conservative algorithms such as the Lockstep Synchronization algorithm (Funkhouser, 1995), no member is allowed to advance its simulation clock until all the other members have acknowledged that they are done with the computation for the current time period. This prevents out of order events from being generated. It is impossible for inconsistencies to occur since no member performs calculations until it has the exact same information as everyone else. Unfortunately, this scheme also means that it is impossible to guarantee any relationship between simulation time (game time) and wall-clock time. Also if one node in the system fails, then all the others may wait indefinitely before receiving updates from that node and impeding any progress in the game evolution (Palazzi, 2006).

This type of approach is suitable for discrete applications but not for distributed continuous applications such as multiplayer games because this approach does not guarantee that the application will advance at a regular rate, an essential requirement for continuous applications. In continuous distributed applications, the state of the application not only changes with users' interactions, but also with time. Hence conservative algorithms are very rarely used for game state synchronization.

2.1.2 Optimistic Approach

2.1.2.1 Prediction Based Approach

In continuous distributed applications such as distributed virtual environments and multiplayer games, remote sites interact with each other via a network. Because the update messages from remote sites may reach the other sites after some significant delay, a mechanism is needed to hide this delay so that messages that arrive from a remote user to a local user are not in the "past" but are updated to the "present" clock time.

A very first approach to hide this latency and make the data consistent on remote terminals is the prediction of the positions of remote objects. This is called Dead-Reckoning named after the naval approach of finding the position of a ship in the sea at a certain time. Dead-Reckoning (IEEE, 1995) is used to reduce bandwidth consumption and hide network latencies. It consists in sending update messages less frequently and estimating the state information between the updates using the already received information such as position, velocity and/or acceleration of the object. The predicted value can be different from the actual one, which is received through the update message. In this case, some convergence method can be used to reach the actual value (Palant et al., 2006). The importance of dead-reckoning in mobile games is that it permits a mobile terminal not to be blocked and to continue even if it is not receiving the data in case of a short term disconnection, for instance. The convergence method must fuse the actual value and the predicted value in a smooth way, so that there are no abrupt effects on the game user. (Singhal and

Cheriton, 1994) proposes a dead-reckoning protocol based on the position history of the object being dead-reckoned. It makes sense to use the simple dead-reckoning algorithm when the path is smooth and straight such as in car race games, and to use the position-based history protocol when the path is not smooth but follows a certain pattern during its motion.

The pre-reckoning algorithm (Duncan and Gračanin, 2003) is proposed as a "variation of the standard Dead Reckoning algorithm to decrease prediction errors without significantly increasing network traffic". This algorithm anticipates the possibility of reaching the error threshold during the prediction and sends a status update message immediately without waiting for the threshold to be violated. Although it can result in the increase of update packets being sent unnecessarily, the objective of this algorithm is to eliminate foreseeable errors with a negligible increase in update packets. Different experiments on the pre-reckoning algorithm demonstrate its potential to outperform the standard Dead-Reckoning algorithm as shown in (Zhang et al., 2004).

To synchronize data between remote players, prediction is not enough in the face of high delays because prediction for such a long time is error prone and also because the users continuously interact with the application and change the trajectory of the objects. Although update messages can be predicted to a certain degree, other types of *stand alone* command messages, for example a *shoot* message, if lost, cannot be predicted. The problem of prediction becomes worse in wireless networks because of their higher delays as compared to wired networks. Therefore, other consistency mechanisms are required to synchronize the data among different participants of a distributed and mobile continuous application.

We now discuss such consistency maintenance approaches that are proposed in the literature for distributed continuous applications.

2.1.2.2 Bucket Synchronization

The Time Bucket Synchronization Mechanism (Steinman, 1990; Steinman, 1991) uses cycles of times to allow different nodes to synchronize with each other. At the end of every cycle, each node increments its simulation time in unison by a certain amount T. A variation of the Time Bucket Synchronization was implemented in a distributed game called Mimaze (Gautier and Diot, 1998). In this synchronization mechanism, the game time is divided into fixed time intervals with associated buffers called buckets. Messages received from remote players with their time stamps are stored in their corresponding buckets, i.e. each time-stamp has an associated bucket. To deliver a global state, the messages in the current bucket are processed to compute the current global state.

To synchronize the remote messages with the local messages, the local messages are delayed and stored in later buckets (in terms of time) so that remote messages issued at that time can arrive. All messages arriving later than their bucket time are simply discarded. The advantage of this approach is that in case of small delays, we could buffer the local message to wait for remote messages without compromising interactivity. However, in case of higher network delays, this approach can result in

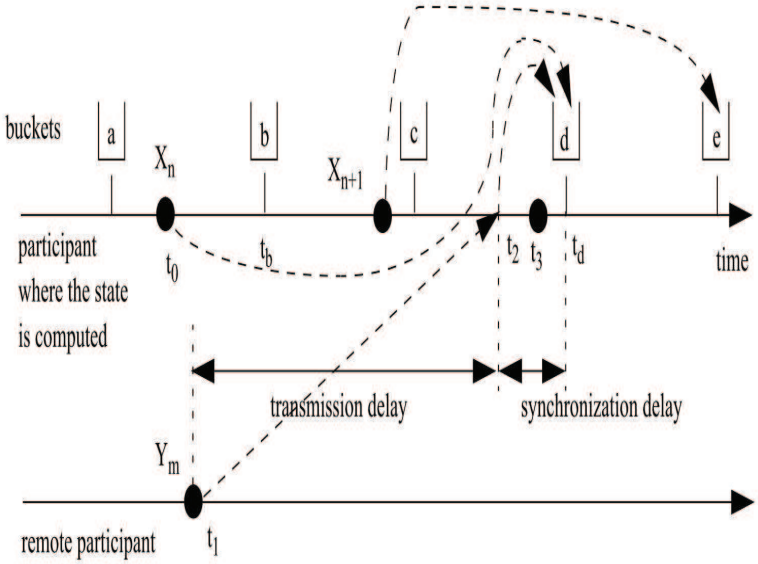


Figure 2.1.1: Bucket Synchronization Approach, taken from (Gautier and Diot, 1998)

discarding lots of messages thus producing inconsistencies. Also for mobile multiplayer games, a complex problem is that of fitting the bucket size with the unstable conditions of wireless networks having large and variable jitters.

The bucket synchronization approach is shown in figure 2.1.1. Without any synchronization, the local ADU (Application Data Unit or message) issued at time t_3 would be processed together with the ADU issued at t_1 from another location (but received at t_2, which is in the same state processing interval than t_3). Bucket synchronization allows information received at t_0 to be delayed in the bucket d (that will be processed at t_d) in order to be synchronized with the ADU issued at t_1 by another participant.

A variation of bucket synchronization, called Locked Bucket-Synchronization Algorithm (LBSA), is proposed in (Moon et al., 2006). LBSA adopts the method of the lockstep algorithm which is a send-and-wait mechanism. In this algorithm, each site moves forward only when the frame corresponding bucket is filled with packets of all the players. "When the delayed packet arrives, it is stored in the corresponding bucket" and the player's game process moves forward again. The problem with this approach is its pessimism as the game cannot move forward unless a player's bucket is filled. This type of approach is not suitable to fast paced games such as car racing games or first person shooter games.

For fast paced games, we cannot wait for the messages that are delayed more than the players reaction time, and therefore we need optimistic techniques where the game application moves forward without waiting too much for the delayed messages. The definition of "too much" is game dependent and can be in the range of 100 to 300 milliseconds (Pantel and Wolf, 2002; Khan et al., 2007).

2.1.3 Roll Back Approach

Time Warp Time Warp (Jefferson, 1985; Lin and Lazowska, 1991; Mauve et al., 2004) is a synchronization mechanism for parallel/distributed simulations. It allows logical processes to execute events without the guarantee of a causally consistent execution. Upon the detection of a causality violation, rollback procedures recover the state of the simulation to a correct value. Two problems occur with this method when using it in networked multiplayer games. First, it requires to store previous states, for which memory is needed. Second, rollbacks demand processing power which can be a costly resource on some limited power terminals. (Mauve et al., 2004) proposes to combine the use of local lag and time warp to improve consistency and decrease the number of rollbacks in a mirrored server architecture.

(Liang and Boustead, 2006) combines local lag and time warp in the Quake 3 Arena game [1] to demonstrate how the two approaches can be used to optimize the playability of real life network games. With the combination of these two approaches and with a network lag of 200 milliseconds, their results showed that performance doubled as compared to just using local lag to achieve consistency. When using Time Warp on mobile terminals connected via a wireless network, the number of rollbacks can significantly increase because of the late arriving messages thus requiring additional processing power and memory space for the rollbacks. As computing and memory resources can be very limited on mobile terminals, some other techniques that reduce the number of rollbacks are required to be able to play a multiplayer game on mobile terminals in a consistent way without being irritated by the network latency.

Trailing State Synchronization (TSS) TSS (Cronin et al., 2002) also executes a rollback when an inconsistency is detected. However, it implements rollback intelligently to avoid high memory and processor overheads demanded by Time Warp Synchronization. As shown in figure 2.1.2, instead of keeping snapshots of every command as in Time Warp, TSS keeps two copies of the same game world, each at a different simulation time separated by some synchronization delay. The most recent one in the time domain is called the leading state. The other is called the trailing state. When an inconsistency is detected in the leading state and rollback is required, instead of re-processing all the states for each snapshot as TW does, TSS just simply copies the game status from the trailing state to the leading state, and then performs all commands between the inconsistency point and the present point again. TSS does not actually solve the rollback problem originated from TW, but it will have better performance when the following two situations are present. First, the game state is large and it is expensive to store the snapshots. Second, the delay between states is small. In order to rollback, copies of the past checkpoints are required. It is still a challenge to do it with less memory and processing speed in case of playing a game on mobile phones.

[1] http://www.idsoftware.com/games/quake/quake3-arena/

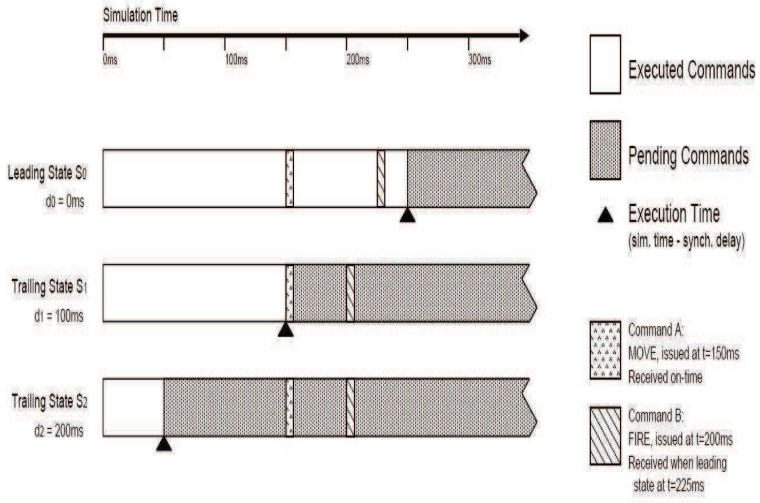


Figure 2.1.2: Trailing State Synchronization Algorithm, taken from (Cronin et al., 2002)

2.1.3.1 Obsolescence-based Synchronization

Before discussing the obsolescence-based synchronization scheme, we discuss the properties of correlation and obsolescence. *Correlation* (Ferretti and Roccetti, 2004b) among game events may be characterized as a property which states that two game events e_i and e_j are correlated if different execution orders of the two game events lead to different game states. Examples of correlated game events typically involve game events, possibly generated by different players, which act on the same game elements. Independent movements of different virtual characters, instead, are examples of non-correlated game events. (Ferretti and Roccetti, 2004b), however, does not define any mechanism to calculate the correlation between any two events. (Xiang-bin et al., 2007) defines two events to be correlated if they are associated with a single object. Hence, to provide players with a uniform evolution of the game, it is enough that correlated game events are processed by all players respecting their correct timestamps order. Instead, no ordering guarantee is needed to process different players' events that do not alter the game state. Such correlation-based order has the main advantage of reducing the synchronization overheads by reducing the number of rollbacks.

We now discuss the notion of *obsolescence* (Ferretti and Roccetti, 2004b). Given two subsequent game events e_i, e_j with time stamps $T(e_i), T(e_j)$ respectively such that $(T(e_j) > T(e_i))$ it may be the case when processing e_j without e_i leads to the same final state that would be reached if both events were processed in the correct order (i.e. e_i becomes obsolete). Recent studies demonstrated that by exploiting the semantics of the game, there exist many situations where fresher game events cancel the importance of previous events. For example, knowing the position of a character at a given time may be no longer important after a certain

time period, if the position of the character has changed. It is also worth noticing that the notion of obsolescence cannot be applied to e_j and e_i when other events correlated to e_i have been generated within the time interval $[T(e_j), T(e_i)\ ]$. These game events may alter the evolution of the game state thus making inapplicable the notion of obsolescence.

Waiting Obsolescence-based Scheme In this approach (Ferretti and Roccetti, 2004b), the notions of correlation and obsolescence are used. This scheme has two requirements. 1) Correlated events must be processed in the same order by all the participants and 2) Persistent events must always be processed. Non correlated, non-persistent late arriving messages can be discarded. Persistent events are those events that must be eventually delivered irrespectively of their delivery time, such as the shooting of another object. The problem with this scheme is that for correlated events to be processed in the same order, the system must wait for all of them before processing them. This is somewhat similar to the Bucket synchronization scheme (Gautier and Diot, 1998). The problem with the bucket synchronization scheme is that it discards late arriving events irrespective of their importance in the application. Instead, this approach only discards obsolete events. Note that no roll-backs are required for this scheme as opposed to Time Warp because of its wait for correlated events.

Optimistic Obsolescence-based Synchronization As in Waiting obsolescence-based scheme, the optimistic approach (Ferretti and Roccetti, 2005a) also uses the concept of correlation and obsolescence. According to Optimistic Obsolescence-based scheme, each receiving player verifies if a given event e may be already identified as obsolete. In this case, e is dropped. Otherwise, a check is carried out to control whether any game events e_i, correlated to e and generated after e, have been already processed. If this check succeeds, then a rollback procedure is performed (which is based on a standard incremental state saving technique such as (Fujimoto, 1999) where all events e_i are rolled back). At this point, e is processed followed by the execution of all those rolled back events which are not obsolete. In fact, obsolete events are discarded during the rollback process. If the check fail that determines that a rollback procedure is needed, e is directly processed. It is easy to observe that this OOS scheme respects the correlaction-based order and guarantees that only useless (obsolete) game events are eventually dropped by some player. An important point is that OOS guarantees the game state consistency among all players. The final game state computed by OOS scheme is not altered while an augmented interactivity is achieved by dropping obsolete events and permitting different processing orders for non-correlated events.
(Ferretti and Roccetti, 2004b) also presents a scheme similar to the above, but without the notion of obsolescence and it processes only correlated events in the same order.

2.1.3.2 Perceptive Consistency

Perceptive Consistency (PC) (Bouillot, 2005) provides an ordering of updates and avoids potential conflicts. Before discussing Perceptive Consistency, two terms need

to be defined. The property of *legality* requires that the latency for a given media instance between two remote processes must be kept constant. For example, in the case of a car racing game, the legality property is respected if the time between two successive positions of a given car is the same for the two users, and hence the speed is also respected. The *simultaneity* property states that the physical time between the playouts of two updates is the same for all users. For the car racing game, the simultaneity property is hold if in the case of a collision between two cars, the two cars are considered to be at the same place at the same given time by all the users. For a system to be perceptive consistent (Bouillot, 2005), it must satisfy both properties of simultaneity and legality.

The algorithm implementing PC has three phases. In the first phase, the algorithm calculates, for a given player, the maximum communication delay between this local player and all the remote players. Then, in the second phase, the algorithm calculates the local lag to be introduced locally to order the events before the playout of a media instance. In the third phase, the message is played out. In the case of mobile games, as network delays can be quite longer, the local lag introduced by PC can have bad effects on the players. Hence, dead-reckoning can be combined with PC to hide the effect of local lag by adding predicted intermediate states.

2.1.3.3 Rendez-Vous

The Rendez-vous platform (Chandler et al., 2004; Chandler and Finney, 2005a; Chandler and Finney, 2005b; Chandler and Finney, 2005c) is intended as an alternative to rollback based consistency methods. This is particularly true in high latency networks where under normal circumstances rollback mechanisms would prove insufficient.

Unlike rollback techniques, the Rendezvous solution copes with inconsistency by accepting a degree of inconsistency as inevitable before gradually trying to converge the inconsistent states forward to a single consistent state in the future. "When using the Rendezvous system, each independent player will see a slightly different view of the game action, reflected by when they received certain state information and what actions they performed themselves". The separate game universes observed by each of the players to form the Rendezvous multi-verse, slowly become increasingly similar over time as Rendezvous calculates the means for the game states to converge by manipulating game rules and begins to act on those calculations, providing a managed inconsistency. Although, inevitably, the players will continue to develop new inconsistencies. However, the system efforts to keep the inconsistencies to a minimum, without significantly affecting the game play or the fairness of the results will provide enough consistency so that separated players can play a multiplayer games across the network.

In order to successfully recover from an inconsistency, it is essential that the Rendezvous mechanism maintains a degree of semantic understanding of the underlying application. As the game continues around the convergence process, Rendezvous makes use of a series of game rules, provided explicitly by the game, to manipulate each independent game reality from their inconsistent states back to an

acceptable convergent reality called a target.

2.1.3.4 Adaptive Schemes

In adaptive schemes, the synchronization approach is adapted according to the situation such as network latency. In (Ikedo and Ishibashi, 2006), the authors propose an adaptive scheme for consistency among players in networked racing games. In this scheme each terminal inputs the positional information about the player's car at regular intervals (every 33 ms in their experimental system) and sends the information with its time stamp, which denotes the generation time of the information, as a computer data media unit (MU). In the proposed scheme, the adaptive Δ-Causality control is exerted for conservation of causality. Under the adaptive Δ-causality control, when each terminal receives an MU, the terminal saves the MU in the terminal's buffer until the generation time of the MU plus Δ seconds ($\Delta \geq 0$). If MU is received after the time limit, it is discarded. However, owing to discarding MUs, the positions of cars may be output incorrectly. To suppress the influence of network latency (i.e. to output the positions of cars as correctly as possible), the proposed scheme uses a prediction technique when the network load is light. In the prediction technique, when no MU for output exists, the current position of the car is predicted by using the latest two output MUs. However, when the network load is heavy, a dead-reckoning technique instead of the prediction technique is used to suppress the amount of computer data traffic.

In the dead-reckoning technique, at each terminal, the current position of each of the other players' cars is predicted by the latest two received (transmitted) MUs. Then the predicted position is compared with the actual position. If the difference between the predicted position and the actual position (i.e. the prediction error) is larger than a threshold value T_{dr} (> 0), the information about the actual position is transmitted as an MU. Otherwise, no MU is transmitted. When a MU is received, it corrects the position over several times in order to correct it gradually until the difference becomes less than T_{dr}.

"When the dead-reckoning technique is used in the proposed scheme, if an MU is received within the time limit (i.e. its generation time plus Δ seconds) of the MU at each terminal, the MU is saved in the terminal's buffer until the time limit; then, the position is predicted and corrected. Otherwise, the MU can be used only to predict the position of the car at the next output time."

In the proposed scheme, the network load is estimated from the value of Δ. The reason is that the value of Δ is dynamically changed according to the network load under the adaptive Δ-causality control. The value of Δ is changed so as to satisfy the following relation: $\Delta_L \leq \Delta \leq \Delta_H$, where $\Delta_L(> 0)$ and $\Delta_H(> 0)$ are the minimum and maximum values, respectively, of Δ. The value of Δ is increased by $\Delta_I(> 0)$ when the number of MUs which are received continuously after their time limits reaches $N_a(\geq 1)$. On the other hand, when $N_b(\geq 1)$ MUs arrive continuously before their time limits, the value of Δ is decreased by $\Delta_D(> 0)$. The initial value of Δ is set to Δ_L. In the proposed scheme, the prediction technique is used when Δ is smaller than Δ_H, and when Δ is equal to Δ_H, the dead-reckoning technique

is employed.

2.1.3.5 Relaxed Consistency

(Li et al., 2004) proposes relaxed consistency by allowing inconsistencies to occur for a short period of time. In their approach, the game runs on a server while a simulation of the player's avatars run on the clients. During the time of inconsistencies, the clients gradually synchronize themselves to the server to achieve consistency. They use a parameter ϵ to denote the limit of inconsistency between two different sites, and when ϵ becomes zero, the relaxed consistency becomes strict consistency.

(Zhou and Shen, 2007) presents a consistency model for Multiplayer Online Games (MPOGs). Based on their observations, they argue that the first consistency requirement for MPOGs is the *expectation presentation* requirement which is delay-induced. For example, a player shoots another player at one site, but this 'shoot' command does not arrive at the other site, and the remote site avatar sends an update message showing that its avatar is still alive. The second consistency requirement is that all sites maintain the same positions of the moving entities at the same wall-clock time. If this requirement is satisfied, the game state as perceived by a player will be consistent with other players' expectations, thus the game is fair for all players - this is called fairness requirement.

Hence an MPOG is said to be consistent if it satisfies the fairness requirement. The fairness requirement is not achievable given the high network latency. Therefore some relaxation is needed. But how much and how ? Again based on observations, a minimum time is required to see the visual effect of an entity. This time is denoted by σ and is around 0.1 to 0.2 seconds. Based on this observation, if a spatial representation X is produced at a site i at time t, and is reached at a site j at time t', if $t < t' < t + \sigma$, then the game is consistent. If the transmission delay is greater than σ, then this requirement may not be achievable and consistency needs to be further relaxed.

As a further observation, if the spatial distance between two objects is too small, it is hard for a human to clearly discern the relative position of the two objects. This minimum distance is denoted by ϵ. Let $X_i(t)$ be the visual representation of an entity at site i at time t, and $|X_i(t) - X_j(t)| < \epsilon$, then the two players at sites i and j will not have different perceptions because this difference is too small. Only when $|X_i(t) - X_j(t)| > \epsilon$ and when this situation has been lasting for a period longer than σ, we would expect that two players do not have a consistent view of the game. Based on the above observation, the consistency criterion is defined as follows:
Let v and T_d denote the moving speed of an object and the average network delay respectively. If $T_d - \epsilon/(v) < \sigma$ then the game is said to be consistent. ϵ/v means that it takes ϵ/v time for visual effect at two different sites to exceed ϵ and $T_d - \epsilon/(v)$ is the time that this situation lasts. Note that the consistency criteria may be satisfied even when $T_d > \sigma$ depending on the value of ϵ and v.

Also the consistency requirement may further be relaxed if the moving objects are not in some critical region e.g. near the base line in a tennis game. Note that

according to the above requirement $v < \epsilon/(T_d - \sigma)$, it means that we have put an upper limit on the speed of an object. However, this can be relaxed if the game is not in a critical region. There must be some mechanism to slow down the speed of the object when it is entering the critical region.

(Zhou and Shen, 2007) relies on loose consistency when strict consistency requirements are difficult to achieve in high latency networks, especially wireless networks. It does not define a consistency maintenance algorithm but just a consistency model based on relaxed consistency.

(Schloss et al., 2008) gives a model of elastic consistency where the consistency requirements vary between a strong consistency and a loose consistency for different types of distributed virtual applications. This work, however, does not describe any specific algorithm for consistency maintenance.

2.1.4 Analysis of Consistency Maintenance Algorithms

Table 2.1.1 presents an analysis of consistency maintenance algorithms. TW and correlation-based algorithms are highly optimistic and they apply rollbacks when needed. They are mainly used for mirrored server architectures. Bucket synchronization is similar to the correlation approach in dropping events, but does not have any rollback mechanism and is implemented on client side in peer-to-peer architecture. Adaptive and relaxed consistency approaches also drop events and provide a loose consistency. Rendezvous is mainly targeted at mobile multiplayer games and exploits the game semantics to provide a loose consistency. This approach is, however, difficult to implement. We can conclude from the table that no single consistency maintenance approach provides a comprehensive mechanism for different games.

2.1.5 Use of Synchronization Algorithms in Other Domains

Distributed Cyber Drama: Distributed Cyber Drama, also known as distributed story telling, is a type of literature in which different participants (through the use of their virtual characters) remotely participate and play different roles in the story. This kind of distributed computer-based entertainment is making significant advances in the entertainment world thereby "becoming an interesting alternative to the tradition linear narratives of TV, radio, movies and magazines" (Mateas and Sengers, 1999; Cavazza et al., 2002; Ferretti et al., 2004; Ferretti and Roccetti, 2004a). (Ferretti et al., 2004; Ferretti and Roccetti, 2004a), through the implementation of game synchronization algorithms, show that consistency maintenance algorithms used in multiplayer games can also be applied to Distributed Cyber Drama application. In essence, the interaction between different participants from the point of view of the exchange of update messages is the same for both multiplayer games and Distributed Cyber Drama, except for the fact, that in a multiplayer games, the participants want to compete out the others, while in cyber dramas, they cooperate with each other in completing the story.

Distributed Virtual Concerts: In a distributed virtual concert, different actors of the concert such as the musicians, the sound engineers and the audience

Table 2.1.1: Comparison of different Synchronization Algorithms

Approach	Architecture	Message Discarding	Rollbacks	Location	Suitability for mobiles	comments
DR	any	no	Convergence	Client	yes	—
TW and TSS	Mirrored Server	no	yes	game server	for wired networks	strong consistency
OOS	Mirrored Server	yes	yes	game server	yes	loose consistency
Bucket Synchronization	P2P	yes	no	Client	for wired networks	—
Adaptive Approach	any	yes	NR	Client	–	loose consistency
Relaxed Consistency	any	yes	NR	–	–	loose consistency
RendezVous	any	NR	no	Client	specifically for mobile games	—

are geographically distributed and interact through the Internet by communicating using real-audio streams. However, *auditory inconsistencies* can arise among musicians due to delays in real-time streams, which can hinder the collective music practice. Consistency algorithms are needed to solve these auditory inconsistencies. It has been shown that such algorithms can also be applied to multiplayer games (Bouillot and Gressier-Soudan, 2004; Bouillot, 2004; Bouillot, 2005).

Distributed Military Simulations: Distributed Military Simulations such as fighter planes simulations or battle tank simulations have similar consistency requirements to multiplayer games. In fact, the dead-reckoning algorithm, a fundamental feature of the IEEE DIS (Distributed Interactive Simulation) standard (IEEE, 1995) was first developed for military simulations. Because of the close similarity of distributed military simulations and networked multiplayer games, the synchronization approaches used in one domain can also benefit to the other domain (Ferretti, 2005; Rhyne, 2002b).

Multimedia whiteboards: Another potential application for consistency maintenance algorithms is Multimedia Lecture Boards (MLB), a sort of shared whiteboards, where participants interact simultaneously to achieve a good approximation of regular face-to-face lectures (Vogel and Mauve, 2001). MLB has a replicated architecture where each user runs an instance of the application and hence consistency control mechanisms are required to keep the replicated application data synchro-

nized. (Vogel and Mauve, 2001) has shown the combine use of "local lag" and "timewarp" to maintain the consistency between distributed users of a multimedia lecture board.

2.2 System Architecture For multiplayer Games

The main obstacle to real-time interactions in distributed applications is the Internet's inability to provide low-latency guarantees. Multiplayer player games have strong consistency requirements, but the combination of consistency and low-latency is difficult to achieve as messages may be delayed indefinitely in the network. The choice of an architecture can greatly affect delays in update messages and also the way, consistency maintenance is performed.

In this section, we briefly discuss different network system architectures used in multiplayer games. We also discuss the bandwidth requirements for each type of architecture and give an overview of the consistency maintenance mechanisms that are used while a game is deployed on a given architecture. First we discuss the centralized client server architecture and then in the next section, we discuss a distributed approach where game state can be distributed at more than one nodes.

2.2.1 Client-Server Architecture

In a Client-Server Architecture, clients input key presses from the users and send them to the server. This architecture is shown in the figure 2.2.3(a).

The server collects commands from all of the clients, computes the new game state, and sends state updates to the clients. The clients then render the new state. In a Client-Server Architecture, the game state consistency is simple to maintain, since there is only one copy of the game state maintained by the server. But a Client-Server architecture suffers from high latency because the messages travel from the clients to a potentially distant server, and then the resulting game state updates propagate back from the server to the clients adding up to the travel time.

Another problem with this architecture is the single point failure represented by the server. Also the scalability of the architecture becomes an issue in case a large number of clients are connected to the server since the bandwidth at the server side is limited and may result in a bottleneck. However there are some advantages of this architecture that is why it is still being used by many game service providers. One of the main advantages of the Client-Server approach, and one of the reasons why most gaming companies prefer it, is its administrative control such as players' payment for their participation in the game, as the data is centralized on one server. Also cheating becomes difficult since all the data passes through the server and can be centrally monitored. As far the mobile multiplayer games are concerned, a potential disadvantage of client-server approach in this case is the high latency of wireless network and frequent disconnections. Also it is difficult to use the interest management technique, which ensures that game entity information is only made available to other entities that are capable of perceiving that information within the game. In this technique, the client side processing is required to calculate the

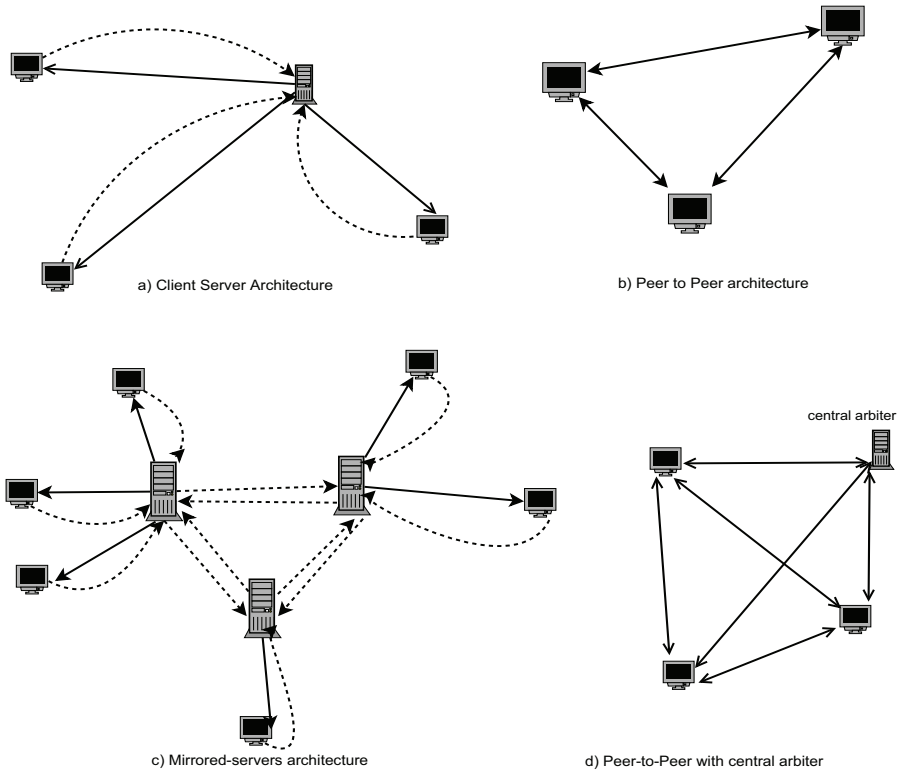


Figure 2.2.3: Different architectures for multiplayer games

information to be made available to the interest zone. In a mobile environment, the clients may not have sufficient processing capability (McCaffery and Finney, 2004).

Bandwidth requirement for CS architecture (Pellegrino and Dovrolis, 2003a) compares the bandwidth requirements of different architectures. Let T_U be the duration of the execution of the game loop including reading the inputs of the local player, receiving updates from remote players, computing the new local state, and rendering the graphical view of the player's world. Let the size of the update be L_U bytes. A client sends a player update to the server in every game loop period, and so the client bandwidth requirement for sending data is L_U/T_U. The server replies to each player with a global state message, which represents the new state of each of the N players. The size of the global state message is $L_G = NL_U$ bytes. So the client bandwidth requirement for receiving data is L_G/T_U. Hence, the aggregate bandwidth requirement for a client in a CS architecture is:
$(L_G + L_U)/T_U = ((N + 1)L_U)/T_U$
which scales linearly with N.

Similarly if we calculate the aggregate bandwidth requirement at the server side, it turns out to be
$(N(N + 1)L_U)/T_U$
which scales quadratically with the number of players N.

2.2.2 Distributed Architectures

In this section, we discuss different architectures where the game state is distributed across the network.

2.2.2.1 Peer-to-Peer Architecture

In a Peer-to-Peer architecture, there is no central repository of the game state. Instead, each client maintains its own copy of the game state based on the received messages from all the other clients. This architecture is shown in figure 2.2.3(b). The primary advantage of the P2P architecture is its scalability. The problem of bandwidth bottleneck at the server in the CS architecture is solved, although the bandwidth requirement at the clients increases. Moreover, message latency can be possibly reduced when there is a direct connection between peers as messages travel directly from one client to another (Cronin et al., 2002).

In a P2P architecture, there are multiple copies of game state. If messages are lost, or arrive at different times, inconsistencies can arise. Due to packet replication (McCaffery and Finney, 2004), it is more difficult in P2P to minimize bandwidth utilisation. Bandwidth utilisation can be minimized by using interest management and relevance filtering techniques. Administrative control in P2P is difficult to achieve as most of the code and data is distributed among many clients, where they are vulnerable to hackers. As far mobile multiplayer games are concerned, IP multicast for mobile phones is yet to be widely established, although in future it may become quite common (Bakhuizen and Horn, 2005). Also, as there is no centralized control of the game, the game service providers generally do not choose this kind of architecture.

Bandwidth requirement for P2P architecture In a P2P architecture, all the players send update messages to each player and receive update messages from each player, so sending and receiving bandwidth requirements are the same i.e. $(N-1)L_U/T_U$ (Pellegrino and Dovrolis, 2003a). The aggregate bandwidth requirement is $2(N-1)L_U/T_U \approx 2NL_U/T_U$ which is twice the bandwidth requirement for a player in CS architecture. Anyhow, it scales linearly with N thus removing the server's bandwidth bottleneck of a CS architecture. Interest filtering can also be used to reduce the bandwidth requirement further.

2.2.2.2 Mirrored-Server Architecture

(Cronin et al., 2002) proposed a Mirror-Server architecture for multiplayer games on the Internet. In this architecture (figure 2.2.3(c)), game server mirrors are topologically distributed across the Internet and clients connect to the closest mirror. It is a tradeoff between Client-Server and P2P architectures. "As in Client-Server, Mirrored Servers can be placed under a central administrative control. Like P2P, messages do not have to pay the latency cost of travelling to a centralized server. Mirrored-Server systems must still cope with multiple copies of the game state". This problem can be solved with the trailing state synchronisation mechanism. The bandwidth requirement of the servers is reduced as they receive messages from

only a subset of all the clients and only a few servers, rather than all players when connected in a P2P manner.

2.2.2.3 Peer-to-Peer with Central Arbiter architecture (PP-CA)

In PP-CA (Pellegrino and Dovrolis, 2003b) (figure 2.2.3(d)), players exchange updates communicating directly with each other, just as in the Peer-to-Peer model. This minimizes the communication delays between players. Each player sends its updates not only to all other players, but also to the central arbiter. The central arbiter listens to all player updates, simulate the global state of the game, and detect inconsistencies. In the absence of inconsistency, the central arbiter remains silent, without sending any message to the players. When an inconsistency is detected however, the central arbiter will resolve it, create a corrected update, and transmit that update to all the players. The corrected players will then be rollbacked to the previous accepted state.

Bandwidth Requirements for PP-CA In PP-CA (Pellegrino and Dovrolis, 2003a), each player sends updates to all other players $(N-1)$ as well as to the single central arbiter. So the sending data bandwidth requirement of each player is NL_U/T_U. Similarly each player receives updates messages from all other players (N-1 players) and from the central arbiter in case of inconsistency (worst case scenario). The receiving data bandwidth requirement for each player is therefore NL_U/T_U. Hence the total bandwidth requirement for a player in case of PP-CA is $2NL_U/T_U$ which is linear.

2.2.2.4 Clustered-Server Architecture

(Hsu et al., 2003) proposes a clustered-server architecture to make the client-server approach more scalable. By contrast to a mirrored-server architecture, each server in the clustered-server has a partial game world and provides a consistent world-view for that part of the system. The game world is divided into regions that have their own multicast group. All players related to a given region send updates to the corresponding server. "When an object crosses the boundary of a region", a common region is defined along the boundary of the two concerned regions. All servers adjacent to the common regions keep the copies of the object. When the object leaves a common region and enters a region solely owned by a server, the other server(s) disown(s) the object thereby reducing the network traffic. The cluster-server is more scalable and synchronization can be achieved efficiently because there are no multiple copies to be synchronized as in mirrored-server architecture.

Table 2.2.2 presents a comparison of different architectures for multiplayer games.

Table 2.2.2: Comparison of different architectures used for multiplayer games

Architecture	Consistency Algorithms	Game Control	Cheating	Delay
Client Server	algorithms are simple	Central	difficult to cheat	long (via server)
Peer to Peer	non-trivial because peers are distributed	Distributed	easy to cheat	relatively small
Mirrored Server	relatively simple	Distributed over Private Network	relatively difficult	relatively small
PP-CA	relatively simple	Distributed	relatively difficult	same as P2P

2.3 Separation of Consistency Issues from the Game logic

In section 1.1, we discussed different consistency maintenance algorithms for the state synchronization in distributed virtual environments such as multiplayer games. We saw that a combination of these algorithms can be used to reach a possible global consistency. These algorithms are very complex and are, therefore, hard to program. It is thus desirable to separate the code of these algorithms from the game logic. To our knowledge, very little work has been carried out for the separation of the synchronization concern from the game logic. In this section, we discuss a communication component, called Medium that has been earlier proposed in (Cariou et al., 2002), for the separation of communication concerns from the core logic of distributed applications.

2.3.1 Medium: A Communication Component

In this section we discuss a communication component called medium. We also discuss our inspiration from this approach to apply it in the multiplayer games domain. When different components of a system communicate between them directly, they have to handle two different things. 1) Their own functionality and 2) interaction details. These two aspects are completely different from one another and mixing the code for them in a single component can lead to problems, such as the difficulty in the evolution of a component's code over time. Presented in (Cariou et al., 2002), the concept of *communication component* or *medium* is to separate interactional details from the functional details of a component. These interactional details can be handled separately by the medium. Hence a medium is

the reification of an interaction, communication or coordination system, protocol or service in a software component. This architecture has the advantage that the same medium or interaction component can be reused for different types of application. An interaction component specifies the services it offers and those it requires.
A medium, like any software component, can take different shapes according to the level at which it is considered. It exists as a specification describing the communication abstraction that it reifies, but also at implementation and deployment levels. It is indeed possible to manipulate a high level communication abstraction at all stages of the software development cycle.

Methodology of specification of the Medium in UML At the specification level, a medium is specified by a collaboration using a UML class diagram. All this information can be described in a contract. This contract must include the list of services offered and required by the medium and also specify the semantics and dynamic behaviour of each of these services and the medium. A UML collaboration can be reified into a medium at the implementation level, ensuring a good traceability from the interaction specification to its implementation.

A medium is specified with the help of three views of UML: 1) Collaboration Diagram, 2) OCL (Object Constraint Language) constraints which allows to specify the properties of the medium such as semantics of the services and 3) State diagrams to manage the temporal constraints

Reification Process of the Medium A refinement process transforms this specification into a low-level implementation design. This refinement process is carried out in three phases. In the first phase, for each component interacting with the medium a class called *Role Manager* is produced. This class is responsible for all the interactions with its corresponding component. A medium is an aggregate of these role managers. This phase corresponds to the choice of platform dependency. In the second phase, the class representing the medium is removed, and only the role managers are left interacting with their corresponding components and with each other. Depending on the non-functional constraints, this phase can lead to many design choices (Cariou et al., 2002). The third and final phase defines, for each design specification in the preceding phase, one or more deployment diagrams, describing how different role managers and components are distributed and grouped at the time of the deployment of a medium. The reification process of the medium is shown in the diagram of figure 2.3.4.

Deployment of interaction Component In an architecture, where communication is represented by a medium, the components do not interact directly, but through the medium. Therefore a component does not need to know the whereabouts of the other components to which it wants to communicate. This localisation issue is handled by the interaction component (medium). To be capable to offer a local service to a component, the medium must consist of many elements (called Role Manager fulfilling a role). Components are categorised by Roles they play from the point of views of the services of a medium) each one locally present associated with a component connected with the medium. These role managers must be connected together through a network infrastructure. The set of these role managers

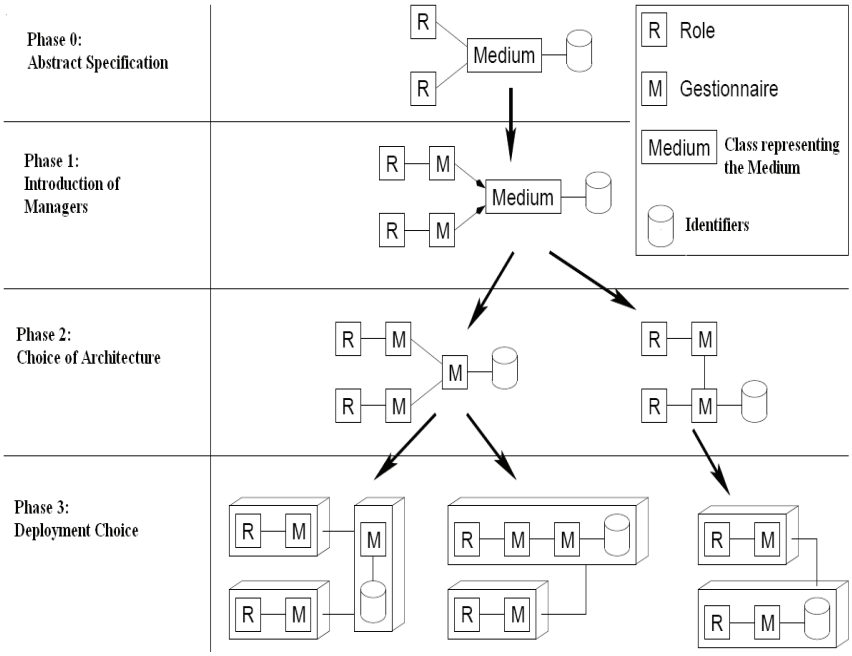


Figure 2.3.4: Reification Process of the medium: diagram taken from (Cariou et al., 2002)

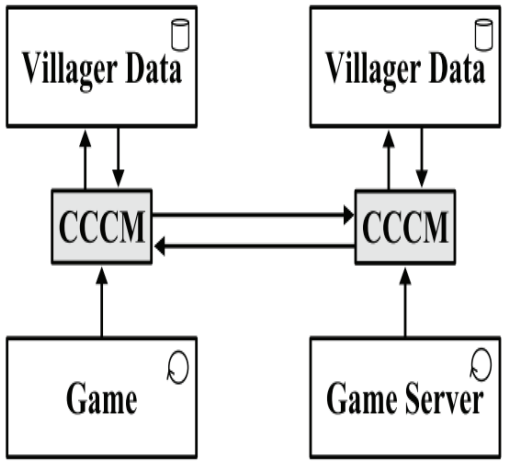


Figure 2.3.5: Plug Replaceable Consistency maintenance: figure taken from (Fletcher et al., 2006)

constitute a medium. Thus from the point of view of deployment, each medium is deployed on many sides fulfilling a role there and offering/requiring a service locally from/to a medium.

2.3.2 Plug-Replaceable Consistency Management

(Fletcher et al., 2006) presents an approach which separates game logic from consistency maintenance code through the use of reusable, plug-replaceable concurrency control and consistency maintenance (CCCM) modules. Using plug replaceable consistency maintenance strategies also permits rapid comparisons of multiple approaches, which facilitates experimentation. This approach is based on Workspace Architectural Model (Phillips et al., 2005). This model provides three advantages over hand-coding of replica consistency maintenance. First, the programming of consistency maintenance is separated from the game application itself, simplifying the main application code. Second, multiple plug-replaceable consistency maintenance schemes can be used in the same application, facilitating experimentation. Finally, consistency maintenance algorithms can be collected and reused in future applications. The idea is shown in figure 2.3.5. In the figure, the CCCM components are inserted into the game client and server, and they synchronize the data between the two. The Villagers Data represent the players' state on the server and on the client side.

2.3.3 Matrix Middleware

The Matrix middleware (Krishna Balan et al., 2005) is a distributed middleware for massively multiplayer games. Matrix provides multiplayer games' developers with an infrastructure offering low latency and consistency maintenance. The two main criteria of the Matrix middleware are 1) to provide an attractive and easy to use game middleware so as to allow games developers to concentrate on the game core logic, and delegate the issue of scalability and communication infrastructure to Matrix and 2) to support the performance requirements of massively multiplayer games. To fulfill the first criterion, Matrix provides games companies preferred client-server architecture allowing developers to use the existing security mechanisms. It also handles "the distributed aspects of the game such as consistency, scalability, resource provisioning and fault-tolerance, leaving the multiplayer games' developer to focus on the core game logic". To meet the second criterion, Matrix offers localized consistency by dynamically handling an area of interest (AOI) approach, and provides low latency by avoiding unnecessary buffering of the packets. One of the disadvantages of Matrix is that the games developers must use the Matrix fixed APIs with the specified infrastructure. They, for example, cannot choose a point-to-point connection to reduce latency so as to achieve strong consistency when required. Also they assume that the game's virtual world can be divided into equal fixed static regions, which is not always possible.

2.4 Concluding Remarks

In this chapter, we presented a literature review related to three different aspects of multiplayer games i.e. consistency maintenance, system architectures and the separation of synchronization concerns from the game logic. We saw that the majority of these algorithms are proposed for wired networks and very little work has been carried out on multiplayer games on mobile terminals, where the network latency is comparatively high and frequent disconnections may occur. Also very little work has been carried out related to the system architecture for mobile multiplayer games. Besides, we believe that the medium approach i.e. separating the functional aspects of a component from the communication aspects can be enriched by separating the consistency aspects from the game logic and making them become part of the medium.

In the next chapters, we discuss our contribution to these three issues. We first start with the issue of consistency maintenance in the chapter 4. The system architecture and the separation of the synchronization concerns are then discussed in chapters 5 and 6 respectively.

Part II

Contribution

Chapter 3

Synchronization Algorithms for Multiplayer Games on mobile phones

3.1 Introduction

In the previous chapter, we discussed different synchronization algorithms used for consistency maintenance in multiplayer games. The approaches proposed in the literature mainly target the networked multiplayer games for wired networks. A very little comprehensive work has been carried out on consistency maintenance in mobile multiplayer games. In this chapter, we propose different mechanisms to handle consistency maintenance in the face of unreliable and high latency wireless networks while playing a game on low memory and processing power terminals. In section 2, we discuss our observation of the consistency requirements in multiplayer games. Then in the remaining parts of the section, we discuss our approach and an algorithm. In section 3, we incorporate the proposal of section 2 to synchronize events from different players by maintaining the causality relationship between them.

3.2 An Adaptable Approach to State Consistency in Mobile Multiplayer Games

Because of the rollbacks and re-processing of commands, TSS (Cronin et al., 2002) and Time Warp (Mauve et al., 2004) are very costly in terms of memory and processor usage and hence are not very suitable to mobile games. Also, the important jitters of the communication delays in wireless networks can cause a large number of messages to arrive late, thereby increasing the number of rollbacks and decreasing the playability of the game. The local lag approach, combined with dead-reckoning is suitable for high latency networks. But because of changing delays and jitters in wireless networks, a fixed local lag can also cause inconsistencies in the game

state across different nodes. Furthermore, we believe that the message discarding approach presented in (Ishibashi et al., 2001) can be interesting to combine with local-lag. This approach allows to discard the messages which arrive late due to the network jitter as they may cause inconsistencies.

In this section, we present a dynamic approach in which the consistency maintenance algorithm changes its parameters according to different factors of the environment, e.g. network load, type of object in the virtual environment, location of an object in the virtual world etc. We first discuss the conditions under which these different parameters are dynamically changed and then we combine these different approaches in the form of an algorithm.

3.2.1 Observations

Observation 1 While playing a multiplayer game, the inconsistencies occur due to communication delays across the network. The game programmers estimate these delays in order to compensate for the late arriving messages from remote users. However, because of the jitters in the communication delays, especially in wireless networks, these delays may vary greatly. Changes in the network latency have a direct impact upon the consistency maintenance approach being used, and consistency requirements can be violated if these dynamic changes in the network latency during the game session are not handled. Therefore, it is necessary to observe the network load during the game session and compensate for these delays accordingly on the fly.

Observation 2 In a rich multiplayer game, there are many types of objects in the virtual environment. The velocities and directions of these objects vary according to their nature. For example, in a tennis game, the speed of the tennis ball is greater than the speed of the players. Also a player has to react sharply to the movement of a ball. Therefore, these different types of objects in the game world have different types of consistency requirement. The algorithms responsible for consistency management have to react not only to the varying latencies of the underlying network infrastructure, but also have to deal differently with different types of objects of the game.

Observation 3 We also observe that the consistency requirement of an object not only depends upon its speed, but also upon its position/location in the virtual world. For example, in a car racing game, we need strict consistency management when two cars are very close to each other and that they both are very near to the finishing line. In a football game, a strict consistency maintenance is needed when a player along with the ball enters the goal-area. However, if the play is in the middle of the field with no opponent players near him/her, then the consistency requirement can be relaxed. Hence the location of an object must be taken into account while maintaining its global state.

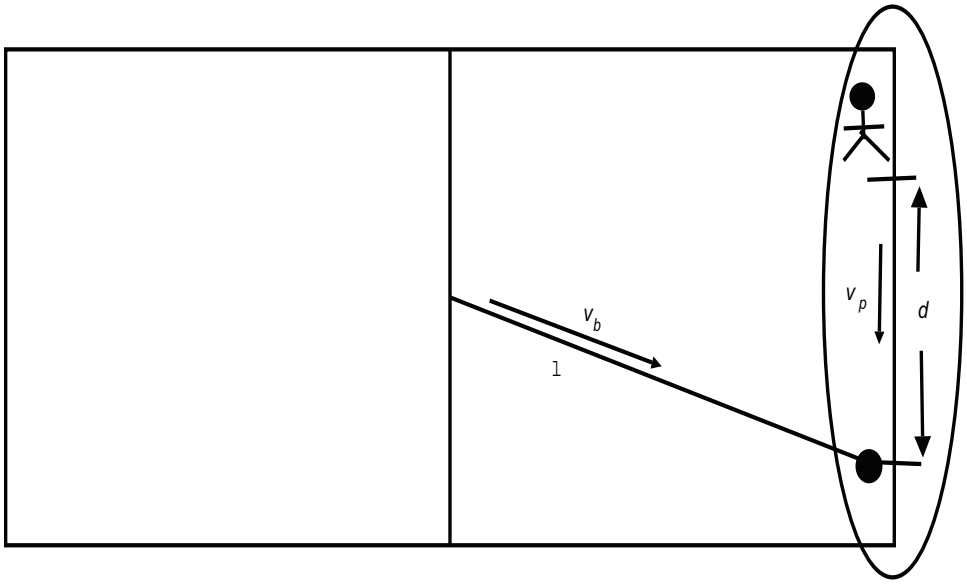
With respect to these observations, we believe that a consistency maintenance algorithm must take into account the context of the game along with the network latency. In the next section, we discuss how we propose to dynamically vary the values of different parameters for consistency maintenance.

3.2.2 An Adaptable Local Lag

When a message (containing information about the object such as its speed, direction, position, and the time stamp etc.) about an object is received from a remote user, this object has a certain distance, possibly zero, with another object, destination, or any other important entity in the game world, called pivot. This distance between the objet and the pivot is essential in determining the consistency and interactivity requirements at that point in time. For example, in a cricket or baseball match, when the ball is approaching the bat, the need for high responsiveness from the system increases.

We change the value of the local lag according to three factors:

1. If the object whose message we have received from the remote user is coming closer towards the pivot, we reduce the value of the local lag. If the object is going away from the pivot, we increase the value of the local lag up to a certain limit called $Local\text{-}lag_u$. This increase can be continuous with respect to the motion of the object, or can be discrete based on zones as in (Santos et al., 2007). The rate of the change of the value of the local lag is application dependent, and the programmer must specify it during the development of the game. In chapter 6, we discuss how a programmer can specify these values to a communication component responsible for consistency management.
2. The value of the local lag also changes according to the load on the network. When the number of messages arriving later than a certain wait time increases of a certain amount N_d, we increase the value of the local lag. This increase in the number of messages arriving late can be due to jitters in the network communication delays. Note that the value of the local lag is proportional to the network latency, but we cannot increase this value more than a certain limit because it will have a bad effect on the responsiveness. This upper limit is dependent on the pace of the game and can be specified by the game developer.
3. We can set different local lag values for different types of objects according to their importance in the game. For example, we can have a smaller value of local lag for the ball and a higher value for the players in a tennis game because the responsiveness of the ball must be high to satisfy the expectation requirement as in (Zhou and Shen, 2007). Again, we need an interface provided by the component implementing the algorithm so that the game developer can specify the relative values of local lags for different objects in the virtual environment.



The area around the base line constitutes a critical region in a Tennis game.

Figure 3.2.1: Critical region example

3.2.3 Message Discarding

As proposed in (Ishibashi et al., 1999), messages arriving later than a certain time limit, because of the network jitters, can cause inconsistencies in the game and hence it is better to discard them. In (Ishibashi et al., 2001), the authors propose to *vary* the value of the causality interval, i.e. 'the time threshold for discarding a message', according to the network load so as to keep the number of discarded messages under a certain limit. As in a virtual environment all the objects do not behave in the same way, we believe that different classes of objects can be assigned a different value of *delta* according to their properties. For example, in a tennis game, the *position* of the ball coming towards a player is more important than the position of his opponent player, and thus the value of *delta* must not be very large so that the position of the ball becomes more dependent on the prediction than on the real value. We believe that combining the local lag approach with message discarding can yield better results.

3.2.4 Adaptable Dead Reckoning

The value of the dead-reckoning threshold should be dependent on the situation and environment of the objects, that is, *which* object is positioned at *which* place. Sometimes, the consistency requirements for an object must be high, but if that object is not at a certain position at a given point in time, then its consistency requirements can be relaxed. Keeping this idea in mind, we propose to rely on critical regions for adaptable consistency maintenance.

3.2.4.1 Critical Regions

We define a Critical Region as a region in the game, where we need strict consistency so that all the players have a consistent view of the game in that region. A critical region is the one in which inconsistencies can violate the fairness requirement (Zhou and Shen, 2007) of the game or can annoy the player because his expectations are violated. Therefore we propose to increase the transmission of real update messages, i.e. change the dead-reckoning threshold on the fly, so that prediction errors, while calculating the future positions, remain very low in these regions.

For example in the case of a tennis match, if the ball hit by one player is touching the ground near the base line on the other side of the court, and the opponent player is quite far from the ball, we increase the message sending frequency and stop the dead-reckoning so as to increase the fairness between players. The ball will touch the ground only when the original message is received and hence the result of the score will be correct. The calculation of the decision whether to use dead-reckoning or not can normally be done through some easy arithmetics. For example, in figure 3.2.1, let v_b and v_p be the current speed of the ball and the maximum speed of the opponent player respectively. Let l be the distance covered by the ball from the centre of the court to the point where it touches the ground and d be the distance of the player from that point. The ball will reach the ground in l/v_b time units and during that time the player can cover a distance of $v_p * l/v_b$ distance units. So, if $d > v_p * l/v_b$, then the player cannot reach the ball, and we stop dead-reckoning from near the net where we did the calculation. Of course, the ball will stop and jump in the air for a short period of time near the centre of the court but it will not affect the playability and the fairness of the game because we know that the opponent ,ould not reach the ball in any case. Note that, if we do not use the idea of critical region and continue with dead-reckoning, we may predict a wrong position with the ball touching the ground near the base line. Hence, that wrong decision may cause one player to loose a point which he otherwise would have won. However, the consistency requirements can be relaxed outside the critical region.

As we will show in the next section, by relaxing the consistency outside of the critical region, we can reduce the number of rollbacks required in case of late arriving messages thus improving the playability and the responsiveness of the game. Also, with this relaxation, we can increase the dead-reckoning threshold while sending the messages, thereby decreasing the total number of messages sent. This will not only help in reducing the congestion in the network, but also the cost of playing the game on wireless networks where each message sent has a cost associated to it. Figure 3.2.2 gives an overview of different critical regions in a game.

Figures 3.2.2.a) and 3.2.2.b) represent critical regions around an object which is moving.Figure 3.2.2.c) represents one object entering the critical region of another object thus asking for a stronger consistency approach. Figure 3.2.2.d) represents a fixed critical region near the goal post in a football game. It must be noted that the concepts of area of interest and sphere of visibility are different from the notion of critical regions with respect to state consistency maintenance. The former concepts are primarily used for message filtering so as to send messages only to those players in which one is interested. By contrast, the critical region concept is used for consistency maintenance and for calculating a fair final state.

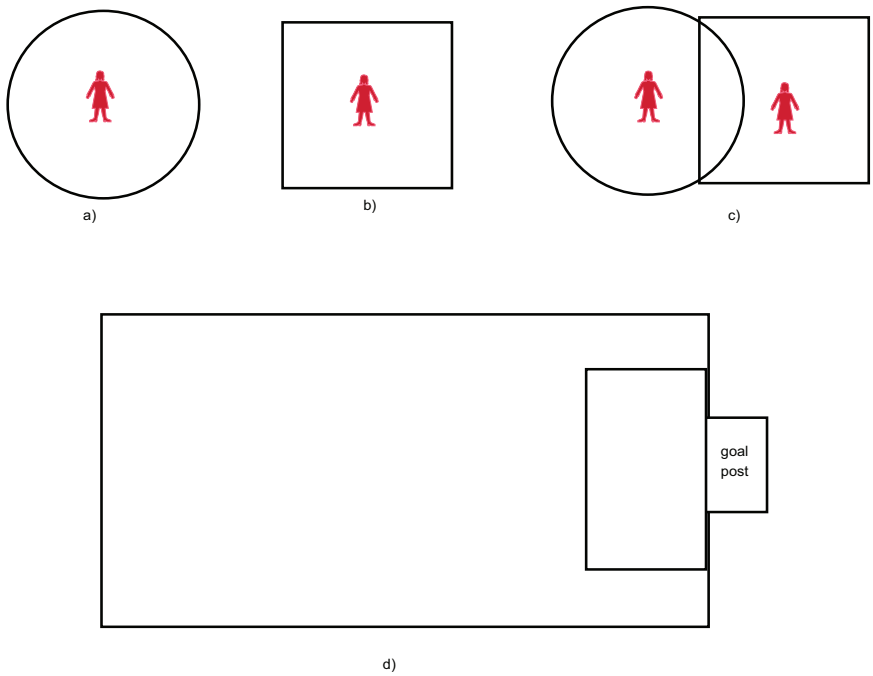


Figure 3.2.2: Different critical regions in a game

(Zhou and Shen, 2007) discusses the critical region concept and proposes to slow down the movement of the object while entering the critical region. The authors put an upper limit on the velocity of an object due to which it does not look like a real world one. Instead, in our approach, we use the critical region just to calculate whether we need to perform dead-reckoning or not. Later in this chapter, we use this concept to reduce the number of rollbacks by exploiting the consistency requirements of different regions in the game, by determining whether we need a strong consistency or a relaxed one in a given region.

(Santos et al., 2007) discusses about the distance between objects and an object's distance from a pivot, which can be an object, destination or any other important entity in the game world. This paper proposes to set different degrees of consistency for different objects according to their distance from the pivot. However, they do not deal with changing the value of the parameters dynamically during the execution of the game. Apart from critical regions, dead-reckoning is also dependent on the network load and on the nature of the object. For example, we can have different dead-reckoning thresholds for different objects according to their movement and importance in the game world. These thresholds must be specified by the game developer to the component responsible for consistency maintenance, as we discuss later in this report.

3.2.5 The Algorithm

In this section, we combine the ideas discussed above in the form of an algorithm. The algorithm is shown in Algorithm 1. The game program assigns local lag values at the start of the game (line 1). This is done separately for different classes of objects and according to the network conditions. Whenever a message arrives from a remote player or is transmitted to the network, the network conditions and the location of the object in the virtual game are checked, and the program decides if there is a need to change the value of the local lag (line 2). The message is stored for a duration corresponding to the local lag value (line 3). Any message which has taken more than its local lag value time is discarded (line 4). Note that we suppose that all the clocks are synchronized, and hence it becomes possible to measure how much late a message has arrived. If the number of discarded messages (line 5) increases more than a certain threshold for that class of objects, then the value of the local lag is adjusted so that the number of discarded messages is decreased (lines 6, 7 and 8). The dead reckoning algorithm is applied on the stored message after its local lag time (during which it is stored) expires to predict its current position and is displayed (line 9). At line 10, the value of the dead-reckoning threshold is adjusted according to the network conditions and the objects position and/or velocity. The algorithm from line 2 onward is repeated throughout the game session.

Algorithm 1 Adaptive Consistency Algorithm

1: Calculate the *local lag* at the beginning of the game for each class of objects according to network latency and responsiveness requirement of the object(s).
2: Change the *local lag* if the network load has changed or the location of an object is changed
3: Buffer the messages according to their *local lag* value before playing out
4: Discard the messages if they are arriving late
5: Calculate the number of discarded messages
6: **if** the number of discarded messages exceed a certain threshold for that class of objects **then**
7: re-adjust the value of *local lag*.
8: **end if**
9: apply DR
10: Change the threshold value for DR if

- the object has entered the critical region
- the network load has changed

Although dynamic dead reckoning and local lag can help us hide the varying network latency, it is still possible that messages arrive out of order arising the need for rollbacks. When the number of out of order messages increases, it also increases the number of rollbacks which can disturb the playability of the game. Also for

rolling back the out of order messages, we need to store a large number of already processed messages, which demands a lot of memory space, which is always limited in handheld terminals. In the next section, we discuss how our dynamic approach combined with the notion of critical region can reduce the number of rollbacks.

3.3 Incorporating Obsolescence and Correlation in Dynamic Algorithms

In networked multiplayer games, all the messages from remote players should arrive at a local player in the same causal order so that the calculated state at a certain player is consistent with the global state of the game (Cheriton and Skeen, 1993). However, satisfying the causal order for all the messages in the game would degrade the performance of the game (Défago et al., 2004). Because of the of unordered delivery of events through unreliable protocols, the game state, because of network delays, at different players can be inconsistent at different points in time. To address this issue, synchronization algorithms are used to reach a consistent state at all the players. In an optimistic synchronization approach, the participants process events without waiting for the arrival of late events and repair any potential inconsistency when it actually occurs. This approach is suitable for continuous interactive applications. Different solutions (Mauve et al., 2004; Cronin et al., 2002; Ferretti and Roccetti, 2004b; Ferretti and Roccetti, 2005a; Xiang-bin et al., 2007) can be used to repair inconsistencies resulting from relying on an optimistic approach. A rollback mechanism allows repair the inconsistencies that occur because of the late arriving messages due to an unreliable network protocol. These rollback mechanisms rely on fixed parameters to solve inconsistencies without keeping in view the changes in the network and game environment. In wireless networks, the delays are higher as compared to wired networks and also jitter can occur varying the network delays. Also, we believe that in a multiplayer game, different objects and regions in the game have different consistency requirements that vary during the game session as discussed in (Khan et al., 2008).

In this section, we propose an optimistic approach based on rollback mechanisms which adapts its behaviour according to the changes in the network conditions and the consistency requirements of the game at a particular time. The aim is to considerably reduce the number of rollbacks and improve the playability of the game while using a wireless network. Furthermore, we introduce the idea of *critical actions* to denote highly sensitive events in the game which impact the outcome of the game. The update messages representing these actions cannot be discarded and must be eventually delivered. Also a rollback of these messages can have an adverse effect on the result and the usability of game. However, combined with the notion of *critical region*, there can be situations where we can discard even *critical actions* to reduce the number of rollbacks.

3.3.1 Critical Regions and Critical Actions

We have already defined the concept of critical regions in multiplayer games. We now introduce the concept of *critical actions*.

We define *critical actions* or *events* as commands in the game, that unlike position update messages, affect the result of the game for other objects. For example, a shoot command can increase the score of a player and can kill or reduce the moving ability of another player. Hence, we consider the shooting command as a critical action. The delivery of critical actions is highly essential. However, according to our observations, there are times when dropping such events does not violate the outcome of the game, as we will discuss shortly. Also, the rollback of a critical action can cause a user to quit the game because of the violation of their expectation (Zhou and Shen, 2007) as there is difference between state that he/she could achieve because of his/her own actions and the resultant state after the rollback. Therefore, efforts must be made to deliver the critical actions before users' expectations are violated and rollbacks of critical actions should be avoided whenever possible. We argue that in certain cases, such as very high latency and when the action is not taking place in a critical region, the critical action message can eventually be dropped if arriving *very* late. The definition of *very* depends on the nature of the game, the value of the network latency and the local lag at that time.

3.3.2 Weak and Critical Correlation

Introduced in (Ferretti and Roccetti, 2004b), the concept of obsolescence states that an event that arrives late at a recipient and some event that was issued after this event (i.e. having greater time stamp) has already been processed, is obsolete and must be discarded. Additionally, the concept of correlation states that if this obsolete message is correlated to an earlier processed event, then all the events till that correlated event must be rollbacked and reprocessed along with this new arriving message. (Ferretti and Roccetti, 2004b), however, does not define any mechanism to calculate the correlation between any two events. (Xiang-bin et al., 2007) defines two events to be correlated if they are associated with a single object. We introduce here a new concept of *weak correlation* and *critical correlation* by incorporating the notions of *critical regions* and *critical actions*. We propose that any late arriving event should be considered obsolete if it is neither in a critical region nor is a critical action. A critical action in a critical region should never be considered obsolete and must be eventually delivered.

We now give the definitions [1] of weak and critical correlations.

Property1 : *weak correlation* χ_w,
Given two critical events e_{ci} and $e_{cj} \in E_c$, where E_c is the set of all critical events that are related to o_i and $o_j \epsilon O$, the set of all objects, with time stamps T_i and

[1] We adopt a simplified version of a syntax based on a Plotkin-style operational semantics (Plotkin 1981 (Plotkin, 1981)). In particular, (ei; ej, s) $\rightarrow$ s* denotes an initial game state s from which final game state s* is reached through two subsequent events, namely ei and ej.

T_j, such that $T_i < T_j$, then,
$e_{ci}\ \chi_w\ e_{cj}$ iff
$(e_{ci};\ e_{cj},\ \mathrm{s}) \rightarrow s1 \wedge (e_{cj};\ e_{ci},\ \mathrm{s}) \rightarrow s2 \wedge s1 \neq s2 \wedge o_i$ and o_j lie outside any critical region. $s1$,$s2 \in S$, the set of all states.
All non-critical events are weakly correlated.

Property2 : *critical correlation* χ_c,
Given two critical events e_{ci} and $e_{cj} \in E_c$, where E_c is the set of all critical events that are related to o_i and $o_j\ \epsilon\ O$, the set of all objects, with time stamps T_i and T_j, such that $T_i < T_j$, then,
$e_{ci}\ \chi_c\ e_{cj}$ iff
$(e_{ci};\ e_{cj},\ \mathrm{s}\ ) \rightarrow s1 \wedge (e_{cj};\ e_{ci},\ \mathrm{s}) \rightarrow s2 \wedge s1 \neq s2 \wedge o_i$ and o_j lie in any critical region. $s1$,$s2 \in S$, the set of all states

3.3.3 Relaxed Consistency

In our definition of relaxed consistency, when the number of late arriving messages increases above a certain threshold, thereby indicating a high network latency, we roll back only those events that are critically correlated and we discard weakly correlated events. We also increase the local lag value for critical events so as to further decrease the number of rollbacks and increase the user expectations of the game. Whenever the number of late arriving messages decreases below a certain threshold, then we stop the discarding of weakly correlated events. This is because the number of rollbacks has already decreased with the decrease in the network latency, so processing weakly correlated events will not increase the number of rollbacks beyond a certain limit. Also we decrease the value of local lag to increase the interactivity for the users. For all other events that are not correlated and are not critical, we discard them when arriving late, because it will not affect the playability of the game and will unnecessarily increase the number of rollbacks thereby wasting mobile terminals limited resources such as processing power and memory space.

3.3.4 An Example Scenario

In figure 3.3.3 we show a game scenario example in which a character hits a moving target (an animal in the figure) with its gun shots. To hit the target, the shooter first sends a warning (or ready) sign before shooting it. Without a warning sign, it cannot shoot the target. In the figure, a circle around the animal character denotes a critical region where a gun shot can hit the character. Suppose that the warning sign $w2$ arrives later than the actual shot $s2$. Since $s2$ hits an area that is outside of the critical region, we do not need to perform any rollback when $w2$ arrives since it will not affect the outcome of the game. We say that $s2$ and $w2$ are weakly correlated. In the case that $w1$ arrives later than $s1$, then we have to apply a rollback. Indeed we cannot discard $w1$ as this would violate the game rule requiring to send a warn sign before shooting. We say that $w1$ and $s1$ are critically correlated. If we have already experienced a large number of rollbacks thus suggesting a high network latency, we increase the value of the local lag so as

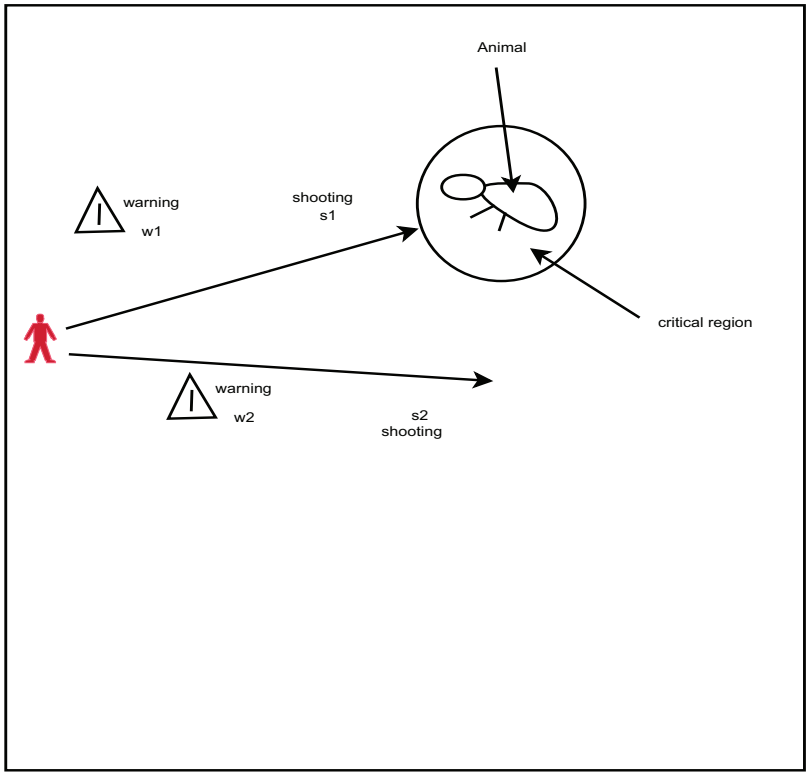


Figure 3.3.3: Correlation of events in a game with two players

to give more time to late arriving messages and hence to decrease the number of rollbacks in the critical region.

3.4 Dynamic Rollbacks Reduction Algorithm

In this section we present a dynamically adaptable synchronization algorithm based upon the concept of obsolescence and critical correlation using the optimistic approach as already discussed in the previous sections. In our algorithm, we propose to minimize the number of rollbacks using a dynamic approach. If the number of rollbacks reaches a certain limit, we increase the local lag value, so that more and more messages could arrive on time. However, when a player enters a critical region and the latency is not very high, we reduce the value of the local lag and increase the frequency of message sending to achieve high interactivity. When the number of required rollbacks increases in the critical region, then, instead of doing rollbacks for all the arriving events, we discard weakly correlated events and increase the frequency of messages to minimize the dependency, i.e. correlation, on a single late arriving message.

The pseudo code for the algorithm is given in Algorithm 2. Lines 10 to 15 are the most important ones as far as the concept of critical correlation and obsolescence and the avoidance of rollbacks are concerned.

We discard any message that arrives late (becoming obsolete) and is not correlated with any previous message(s) (lines 7 and 8). Otherwise, if a message arrives late and is correlated to any previous message e_c and if the number of rollbacks is less than a certain threshold, we apply a rollback on all the messages processed before e_c, including e_c itself, and re-process them in the correct order (line 11). In line 14, we rollback only critically correlated events as now the latency is very high and the number of rollbacks has reached the threshold. In lines 17 to 19, we increase the value of the local lag if the number of rollbacks increases over the limit so as to avoid processing related messages in an incorrect order. This will also decrease the number of obsolete discarded messages which could be related to any future message(s).

There is no doubt that the interactivity will be lessened at the benefit of the consistency and the correctness of the results. We discuss the issue of the trade-off between interactivity and consistency in the next section. We can have an interactivity threshold which would avoid increasing the value of the local lag more than a certain level. We keep a different value for the rollback threshold in critical regions and change the values of the local lag and dead-reckoning thresholds in these regions if the number of rollbacks reaches a certain limit (lines 24 and 25). Lines 2 through 25 are repeated in a loop throughout the execution of a game session.

3.5 Responsiveness vs Consistency

By waiting for the late arriving messages through an increase of the local lag value, the consistency is improved, but it means that even local actions (and those remote events that arrived earlier because of the difference of network latency) must be delayed before being played out. Thus it lessens the responsiveness (or interactivity) of the game. If a game requires high responsiveness, then we need to reduce the value of the local lag which can disturb the causal order of events (and increase the need for rollbacks), thus compromising the consistency of the system. Hence there is a trade-off between these two properties of a distributed interactive application.

We propose to apply different degrees of interactivity and responsiveness for different situations and/or regions of a game. For example, if a player has to shoot a static target, we need high consistency but low responsiveness. Since the target is static and as long as the bullet hits (or misses) the target, we do not need high interactivity from the system but rather a fair result. By applying a suitable value of the local lag, the shooting player will observe that his shooting action has taken place at a slow pace, but will get the true results. However, we need high responsiveness from the system in some other cases, such as hitting a ball coming towards a player in a baseball game.

The trade-off between responsiveness and consistency is shown in figure 3.5.4. Although we have shown consistency (and responsiveness) on a scale from 0 to 1, where 1 means absolute consistency, absolute consistency is never achieved in distributed virtual environments with non-zero communication delays and optimistic synchronization and hence consistency should be compromised for the sake of high responsiveness.

Algorithm 2 Correlation and Obsolescence based adaptive Algorithm

1: Calculate the *local lag* at the beginning of the game for each class of objects according to the network latency and responsiveness requirement of the object(s).
2: Change the *local lag* if the network load has changed or the location of an object has changed
3: **if** the message arrives during its local-lag specified time **then**
4: Buffer the message according to its *local lag* value before playing out
5: GOTO line 20 (apply DR)
6: **end if**
7: **if** the message is obsolete (not arriving in its specified local-lag time) and not correlated with events already processed during time of local lag **then**
8: Discard the message
9: **else**
10: **if** the number of rollbacks is less than a certain threshold **then**
11: rollback the messages and process this message then all others
12: **end if**
13: **else**
14: rollback only critically correlated events and discard all others (we reach when the number of rollbacks is greater than a certain threshold)
15: **end if**
16: Calculate the number of rollbacks
17: **if** the number of rollbacks reaches a certain limit **then**
18: increase the value of *local lag*.
19: **end if**
20: apply DR
21: **if** the object has entered the critical region and/or the network load has changed **then**
22: Change the threshold value for DR for that object
23: **end if**
24: **if** the rollback reaches a certain limit in the critical region **then**
25: decrease the value of DR threshold and increase local lag value
26: **end if**

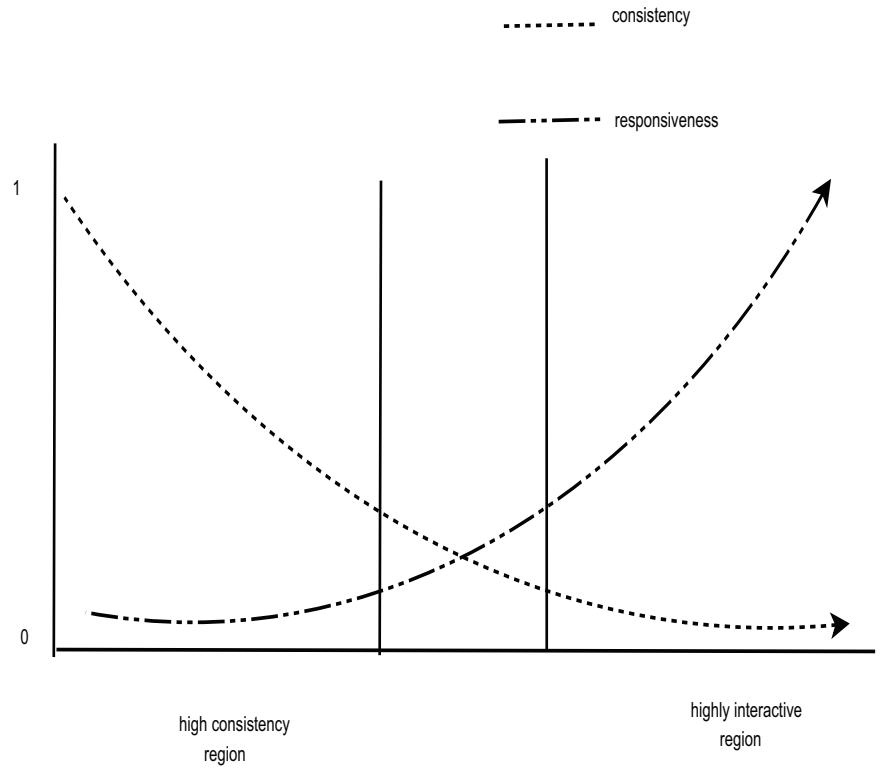


Figure 3.5.4: Trade-off between consistency and responsiveness in different game regions

3.6 Concluding Remarks

The reason for proposing a dynamic and adaptable approach for synchronization in mobile multiplayer games is that the latency of wireless networks is highly unpredictable impeding to apply a static approach. Apart from this, in First Person Shooter games, there are objects in the game world whose pace is fast as compared to other objects in the same game. By having different values of the thresholds for local lag and dead reckoning for different objects, we can reduce the number of update messages for slow moving objects. This creates a space for an increase of the number of messages for fast moving objects in order to increase consistency and responsiveness. Also, the consistency requirement varies according to the position of the game objects in the virtual world. There are areas in the game where we can relax the consistency without having a bad effect on responsiveness and fairness, and there are other *critical regions* where strict consistency needs to be maintained. We believe that identifying these regions during the game development and specifying them to the synchronization medium (chapter 6) will greatly simplify the problem of consistency maintenance.

Chapter 4

Evaluation of the Proposed Synchronization Algorithms

In this chapter, we present the experimental results of the consistency maintenance approach we have proposed in the previous chapter.

4.1 Evaluation of Adaptable Synchronization Approach

We evaluated our proposed approach using a car racing game that we developed for this purpose. The game was developed in J2ME, on the top of the GASP platform (Pellerin et al., 2005; Pellerin, 2010) and can be played on all mobile phones supporting Java. We have already tested it on Nokia N93 mobile devices connected to a GASP server via a Wi-Fi network. The game can also be played over a cellular network. A picture of the game being played on Nokia N93 is shown in figure 4.1.1. The GASP server was deployed in a Tomcat servlet running on an intel machine with 2.5 Ghz processor and 3.5 Giga Byte RAM. The operating system used was Microsoft Windows XP. Each experience was run at least ten times and the average values were calculated which were used to show the results of our algorithms.

4.1.1 Adaptable Dead-Reckoning

Figure 4.1.2 shows the results of our evaluation comparing dynamic and static dead-reckoning. The positions of a remotely displayed car at different times during the game session for the static and dynamic approaches are compared to the original position of the car. From the figure, it is clear that adaptable dead-reckoning prediction errors are smaller than with the simple dead-reckoning. Note that during the experiments, we varied the latency of the network (by keeping the messages waiting at the server side before being executed by the receiving player's terminal). Because of these changes in the network delays during the game session, our

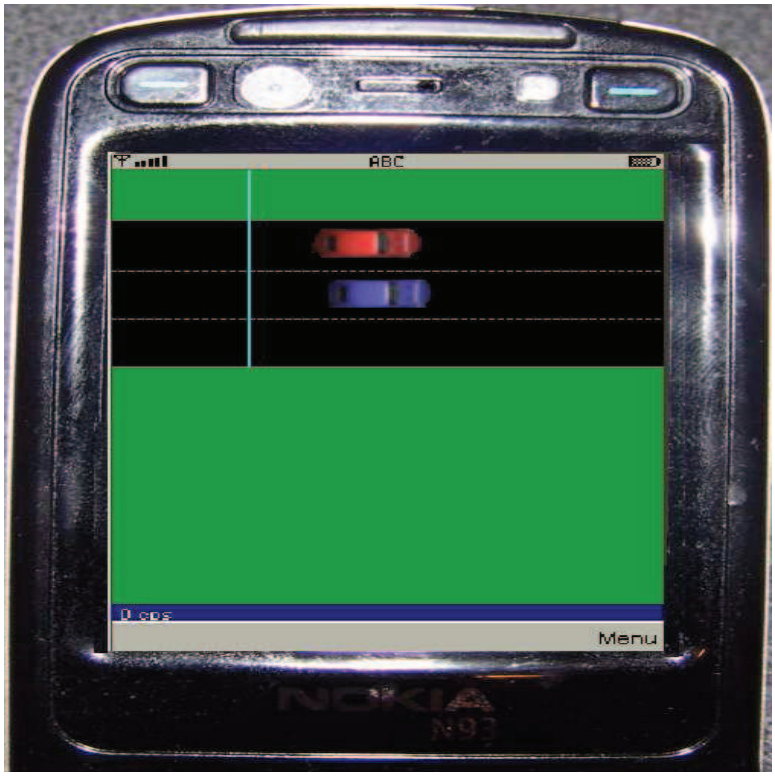


Figure 4.1.1: A simple car racing game played on Nokia N93

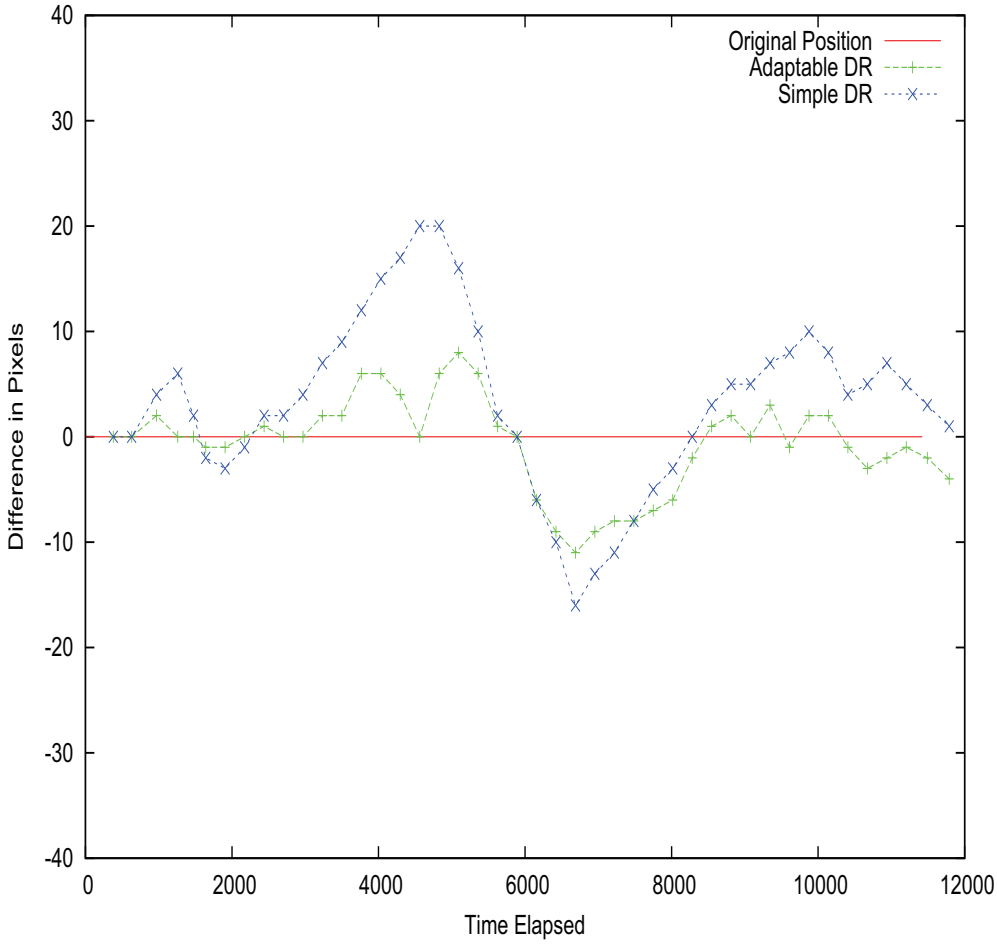


Figure 4.1.2: Adaptable vs Simple Dead-Reckoning

adaptable approach performs better as it adjusts itself on the fly to variations in the network latency. On the other hand, the simple dead-reckoning approach uses the same threshold for sending messages irrespective of changes in the network latency during the execution of the game and hence suffers from greater prediction errors. Had the network latency not changed during the game session, the results with both simple dead-reckoning and adaptable dead-reckoning would have been the same.

In the figure 4.1.3, we compare the static and adaptable dead-reckoning approaches in terms of the number of update messages sent to the server during the course of the game execution. In case of playing games on mobile phones using a cellular network, the users may have to pay for each message that they send. Therefore, it is very important to reduce the number of update messages from the client's mobile to be sent to remote participating players. In the figure, it appears that our adaptable approach allows to decrease the number of exchanged messages for more than 20% compared to the static approach.

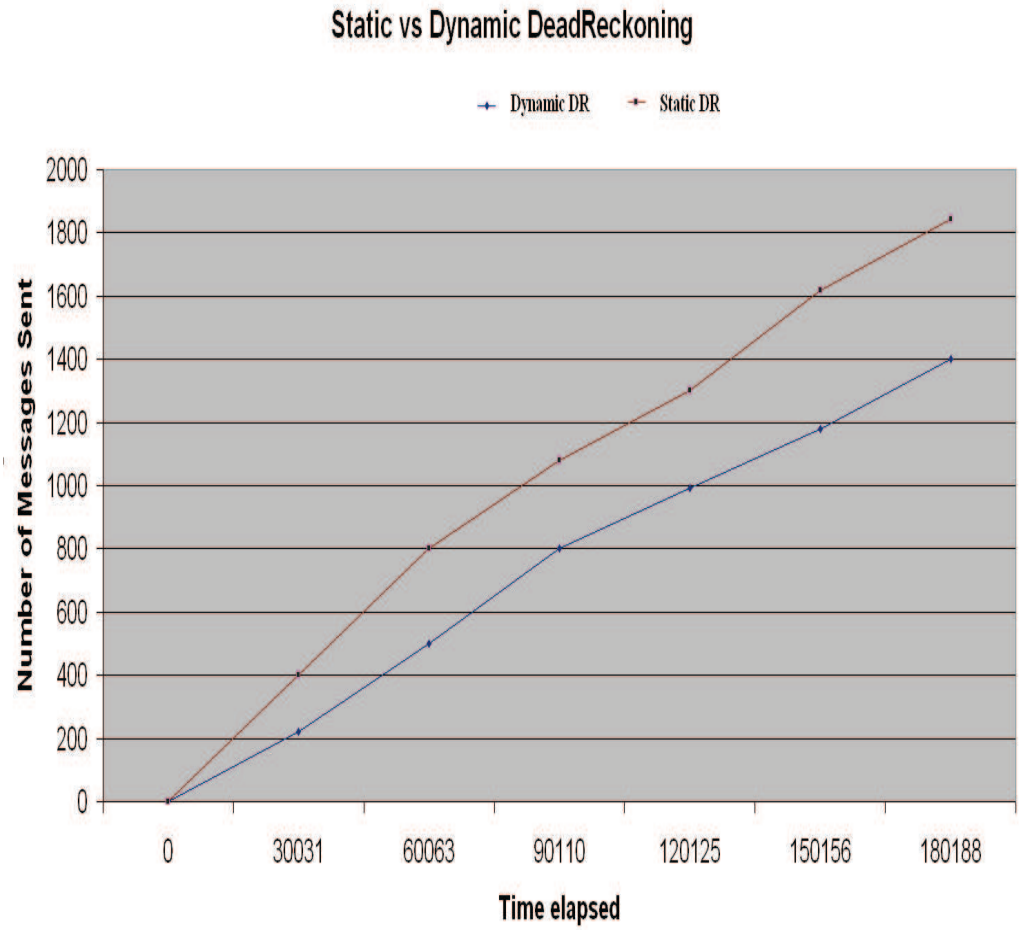


Figure 4.1.3: Number of messages sent in dynamic and static dead-reckoning

This is because in the static approach, which uses at the local player a fixed error threshold to decide when to send update messages, the chances of a prediction error to occur increase. Indeed for each prediction error that passes the error threshold, the local player has to send an update message. In the adaptable approach, the error threshold also varies according to the network and game conditions. It is then less likely that the prediction error will surpass the error threshold and hence the number of messages sent is lesser than in the case of simple static dead-reckoning.

4.1.2 Critical Region Approach

When we apply the critical region approach in those regions where the result of the game matters, we observe an improvement in the game state consistency. The result is shown in 4.1.4. As we can see from the diagram, when the remote car is approaching the finishing line (corresponding to an elapsed time of 9000 milliseconds), our result shows that the position of the car displayed on a remote terminal

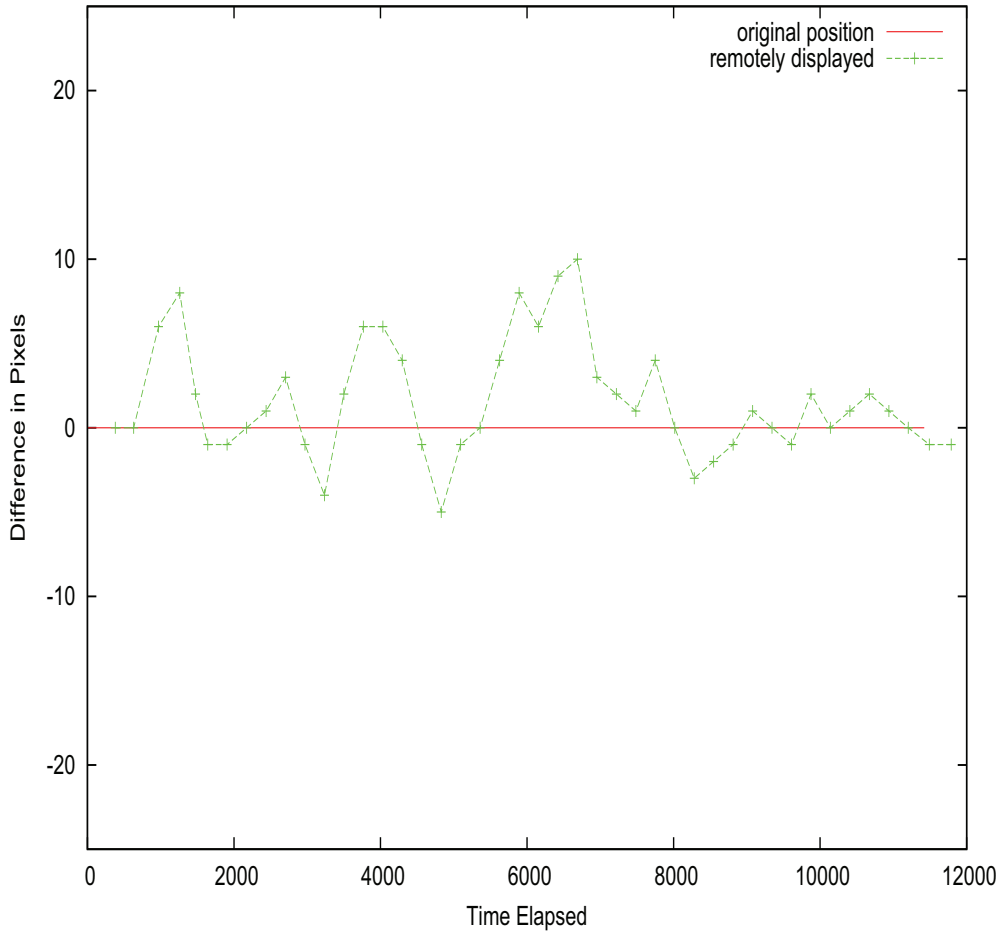


Figure 4.1.4: Consistency of the game using the Critical Region approach

is quite close to the original position of the car. This is because we increase the message sending frequency of the sender and also increase the local lag value for the sender. This is intended to leave some time to messages to reach their destination before displaying them locally.

4.2 Evaluation of the Critical Correlation-based Approach

In the evaluation of the critical correlation-based approach, we are interested in the measurement of two items.

1. the number of rollbacks;
2. the amount of dropped events;

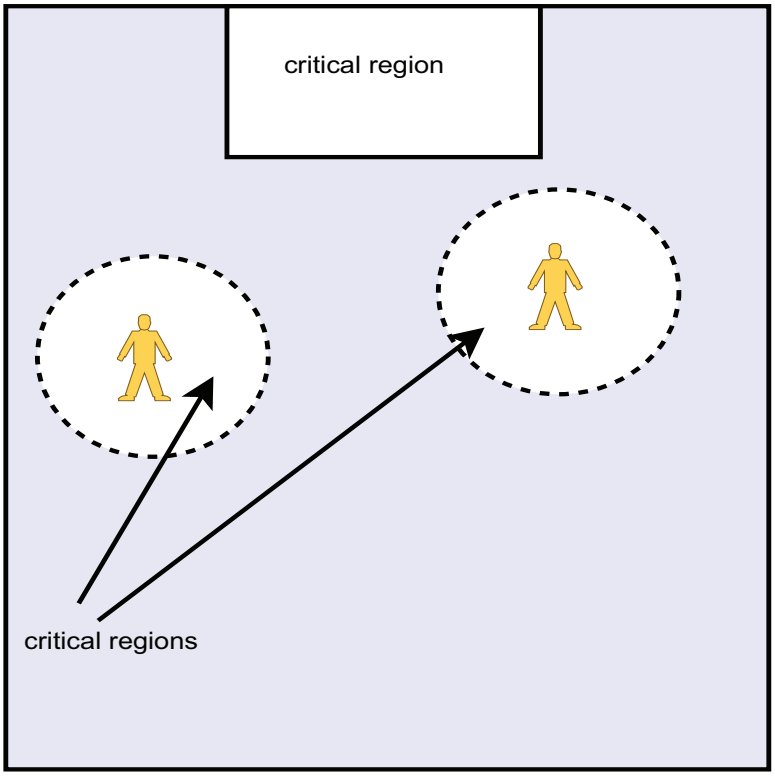


Figure 4.2.5: A game with two players

For our evaluation, we have developed a simple mobile game using J2ME and the Java Servlet technology. The game logic resides on a mobile phone and different players interact with each other via a server which is a simple message queue Servlet. In the game, we have characters representing different players and a goal post, which represents a critical region. Each player has a circle of specific radius representing the critical region around them. A player can earn points by either hitting another player or the goal post with a ball. The rules of the game allow the player to be hit only in the critical region shown by the circular shape. The game is shown in the figure 4.2.5.

On the server side, we randomly select messages to be delayed and deliberately delay them by storing them for some time before the clients can receive these messages. When a delayed message arrives at the client, the client calculates whether this message is correlated with another message or not. In our game semantics, a message is correlated with another message if it belongs to a ball (critical action) or if a player entered a critical region. In case of a very high consistency, we can drop even the *hit* message because it will have no effect on the result since our game rules allow a player to be hit only in a critical region. If the message is not correlated, we simply drop it, otherwise we apply a roll-back mechanism and reprocess all the already processed messages. We continuously calculate the number of roll-backs and apply our dynamically adaptable algorithms by changing the Dead-Reckoning and Local Lag thresholds to control the number of roll-backs.

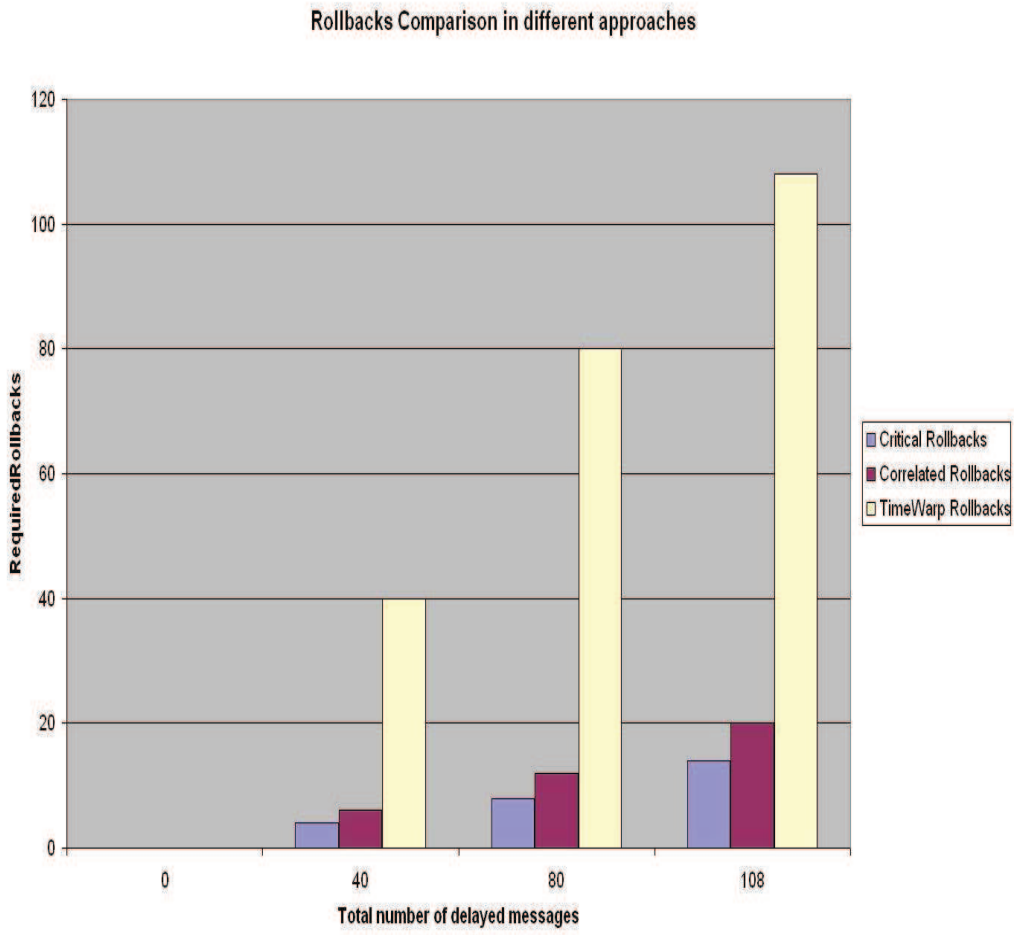


Figure 4.2.6: Rollbacks comparison in three different approaches

In essence, we measure the number of roll-backs in the critical regions and outside of the critical region, the number of messages discarded while the players are in a critical region and outside of it. We also compare these results with the fixed-correlation based approach and the time warp algorithm, where there is no concept of obsolescence and correlation. The results are shown in figure 4.2.6.

From the figure, it is clear that the number of rollbacks required for time warp is higher than for the other two approaches. This was expected, since time warp does not drop any message and applies rollbacks for all late arriving messages. Less than 20% of the late messages imply a rollback when correlation is taken into account. The result of our approach is better than for the fixed-correlation based approach, because we rollback only those messages which are critically important, and discard non-critical messages in case of high delays.

Figure 4.2.7 compares the number of messages re-processed per rollback as a function of their correlation probability. In case of low probability of correlation (high No-Correlation probability on the x-axis), our Critical Correlation based ap-

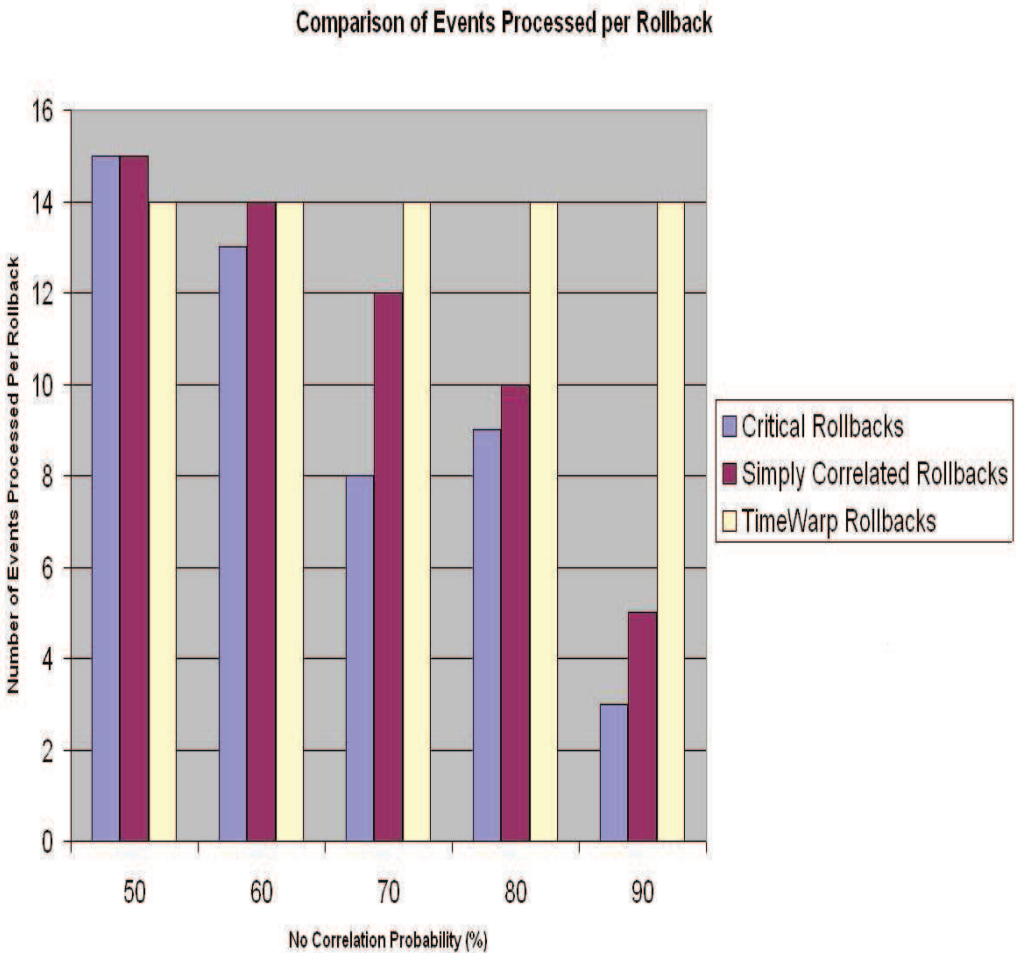


Figure 4.2.7: Comparison of events processed per rollback

proach re-processes a smaller number of messages as compared to the Time Warp and Simple Correlation based approaches. This is because in case of lesser correlation probability, there are even lesser chances that the messages will be critically correlated and hence they are discarded without affecting the outcome of the game. However, in case of high correlation (low No-Correlation probability percentage), our result approaches that of TW and CRL. This happens only when almost all messages are critically correlated. This is very rare in real game scenario, where players enter and exit the critical regions and only a few of their actions are critical.

Figure 4.2.8 compares the increase in the total number of rollbacks as a function of the time elapsed during the game session. Note that on the Y-axis, we show the number of rollbacks required as a whole i.e. when the rollback mechanism has to be applied, and not the number of messages to be rollbacked which can be manifold higher than these numbers. For obvious reasons as discussed in the previous paragraph, our approach performs better than simple correlation. The increase

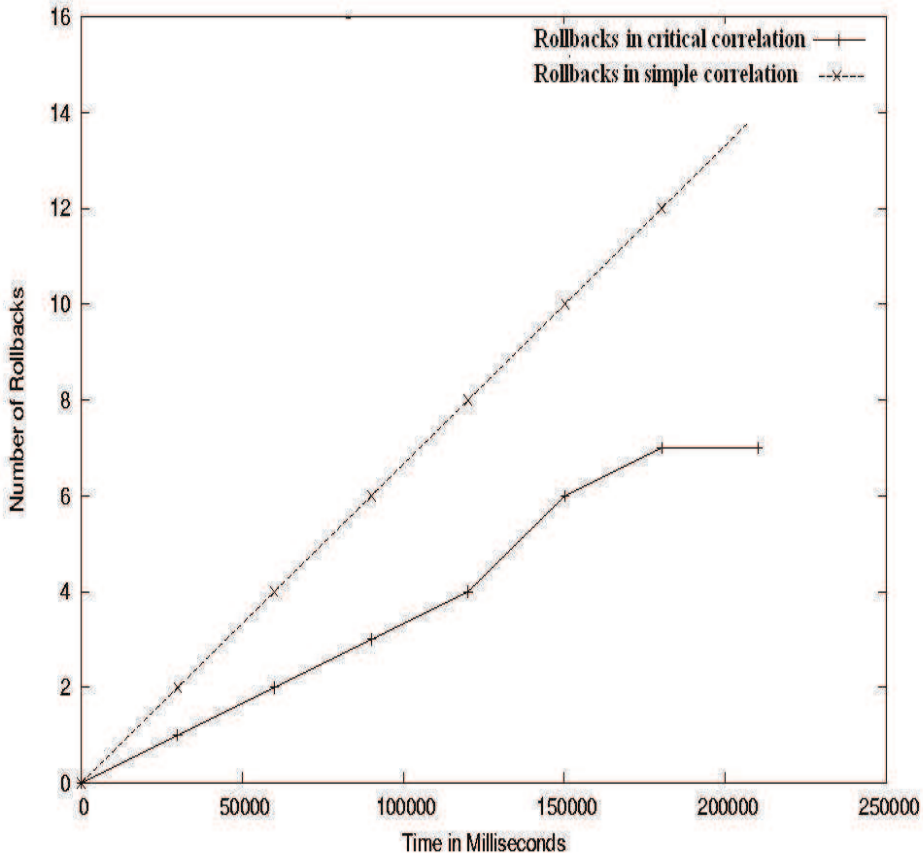


Figure 4.2.8: Rollbacks comparison as a function of time elapsed

in the number of required rollbacks is linear in case of simple correlation because we periodically delay only correlated update messages at regular intervals. In the figure, for the critical correlation-based approach, the increase in the number of rollbacks is not uniform (the graph is not straight). This is because even if we delay messages periodically at regular intervals, they may not be critically correlated at that junction of time and hence are discarded without the need for applying the rollback mechanism. It must be noted here that our approach is dependent upon the strategy of the players that play the game as when and where they send critically correlated messages. In the worst case scenario, when all the messages received are critically correlated, our mechanism is equal to the simple correlation mechanism.

4.3 Concluding Remarks

In this chapter, we have shown our experimental results of the approaches presented in chapter 3. We have seen that adapting the consistency mechanism according to the situation helps in improving the overall consistency of the system. We have also shown that in some areas of the game, we can relax the consistency requirement without disturbing the playability and the fairness of the game.

Chapter 5

System Architecture for multiplayer games

5.1 Introduction

In chapter 2, we discussed different system architectures used for multiplayer games. The client-server architecture is the preferred one because of its central control and simplicity of consistency maintenance. But it suffers from the issues of scalability and high latency between two clients communicating via a server. In P2P systems, the communication can be point to point with possibly low latency (Cronin et al., 2002), however there is no centralized control and a risk exists of cheating from player's side. Also, game companies do not prefer this architecture because they cannot oblige players to pay for the game because of the lack of a central administrative control over the game. P2P also increases the bandwidth requirement at the client side.

(Pellegrino and Dovrolis, 2003b) tries to solve the bandwidth problem by applying a central arbiter, responsible only for consistency maintenance. In this case, the bandwidth requirement at the client side is reduced in case of absence of inconsistencies. (Yang and Sutinrerk, 2007) gives the idea of mirrored-arbiters in place of a single arbiter. Both approaches do not solve the issues of the complexity of consistency maintenance and of the loss of central command in case of P2P. Mirrored-server architecture lies between Peer-to-Peer and client-server, and is both scalable and providing low consistency between players. However, it still suffers from consistency issues in case players are geographically distributed over long distances across the globe. These different types of architecture target mainly the distributed virtual environment (DVE) applications executed on wired networks. However, there is very little work in terms of architecture for multiplayer games on wireless networks. Wireless networks have some special characteristics which require them to be treated differently. These include frequent disconnections, mobility, limited processor, memory and power resources. (Cacciaguerra and D'Angelo, 2008) proposes a model for running mobile multiplayer games with enhanced playability and capable of controlling communications between mobile devices and the game infrastructure. In the case of network failures, the game

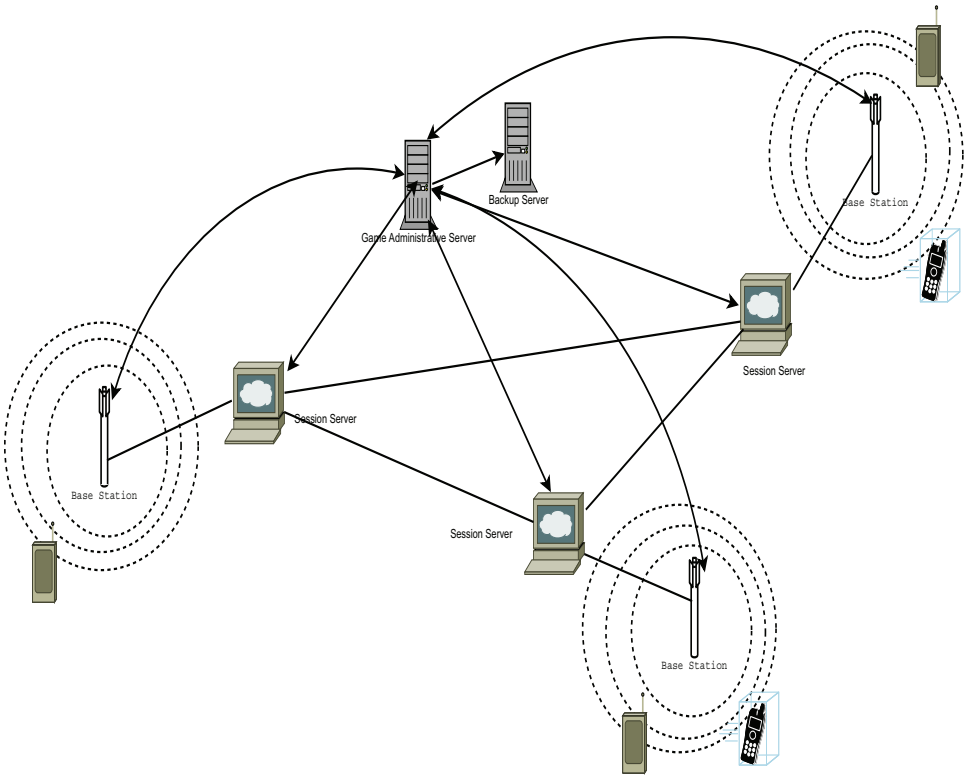


Figure 5.2.1: Session Server based Architecture for 3G mobile Gaming

behaviour is produced locally based on the previous behaviours until the communication channel is restored. This work, however, does not suggest neither central nor distributed control over the game, does not consider the scalability issue and does not provide any synchronization mechanism. In the next section, we propose a game infrastructure capable of handling issues that are specific to wireless networks such as varying latency, mobility and frequent disconnections.

5.2 Session Servers Architecture for 3-G mobile gaming

In this section, we propose a new architecture for 3-G mobile gaming keeping in view the following problems faced while playing a multiplayer game on wireless networks.

1. Frequent Disconnection
2. Loose control over the game servers in case of many Mirrored Servers

Our new architecture is shown in figure 5.2.1. The architecture consists of a single game administrator server. This server contains the management of the

accounts of the players and their billing issue as well as the executable code for the game client's terminals. This server is backed by another secondary server in case this server is down because of some problem(s). The real game server related to the game's logic is executed by session servers which are connected to the administrator server and also between them in a peer-to-peer manner.

When a player wants to play a game through his/her PDA or mobile phone, he/she connects to the administrative server through a 3G network to download the executable files of the game. When the player wants to join a session, he/she requests the administrator server to grant him/her a session. The administrator server grants a session server to this player and informs the granted session server that a new player is going to join it. The player is granted the nearest geographical session server. Depending upon the logic of the game and/or the underlying middleware, a session server can handle many sessions each having a limited number of players. The idea of sessions is important because with the limited display screen, the number of players playing a multiplayer game on mobile phone is limited for a given session of the game, although the total number of players playing the game in different sessions may be quite large. Note that a single session can be replicated in two or more session servers in case players from different geographical areas are playing the same session. In this case it resembles the Mirrored-Server architecture except that a game session is not replicated on all the servers, but only on those whose players are participating in that session.

The session servers, periodically (but not too frequently), send information about the disconnected players to the administrator server so that the later can delete its corresponding data from the database. Also, in case of long distance mobility, the administrator server can disconnect the player from one session server and migrate his/her state to another session server near to this player.

5.2.1 Handling Disconnections

Because of frequent disconnections, a player can loose a game session which can make this player as well as his/her opponent quitting the game in the middle of the session. To avoid this, we propose to have a mimicking engine on the game client as shown in the figure 5.2.2. The client module contains a game's client logic which receives update message from remote players in the session. Based upon of the game logic, this component informs the rendering engine to display the local player as well as the remote players. The network manager is responsible for communication as well as anticipating the disconnections. Whenever the network manager senses a disconnection, the game logic component requests the mimicking engine to mimic the remote players. This can be based upon the current game state as well as the previous behaviour of the game stored by the game logic component. For example in case of a car racing game, a reasonable expectation from the mimicking engine can be to display the remote cars with the same old speeds and directions. This way, the local player is unaware about the disconnection and plays the game as if the remote players are still connected.

On the server side, as shown in figure 5.2.3, whenever the session server sees a disconnection, it initializes a client mimicking controller component to mimic that

disconnected player and shares the updated messages with other players in the same session based upon the logic in the mimicking controller. However when the connection resumes, the state of the remote players on the session server must be copied to the previously disconnected client to replace the states of remote players as predicted by the mimicking engine. This synchronization must be smooth so that the player is not annoyed due to abrupt changes in the remote player's behaviour and positions. The same must be done in copying the state of local player to the server. In case of permanent disconnection (after a certain time has passed and the player has not been reconnected), the mimicking controller at the server should remove this player from the game session in a smooth way. For example, in a car racing game, the mimicking controller can show the disconnected player as stopped at the road side and signalling the other players that he/she cannot continue the play because of some problem in his/her car.

5.2.2 Other Advantages of this Architecture

In the previous section, we saw that the main advantage of our architecture is handling disconnections. In this section, we discuss other advantages of our architecture.

- Single Point of game management: With a single point of administrative control, the game company can have full control over the game in terms of players' accounts and billing. With a backup administrative server, the issue of 'single point of failure' can be easily solved. The scalability is not an issue, as it may appear at first, since the administrator server does not send/received data frequently, but only in case players join/leave the game.
- Because of the clock time difference across the globe, it is possible that players from a certain geographical area will play at the same time while the other will not. For example, at the afternoon time in Europe, there are lesser chances that Australian people will participate in a session. In case the session is limited to a single geographical area, the session will not be replicated at other session servers and hence all the messages received by the session server will be sent to players in that geographical area playing that session without the data being synchronized with other session servers. This can greatly reduce the delay time between two clients. In this case, to avoid single point failure for a session server, a cluster of session servers can be used in a single geographical area or the session can be replicated on at least two geographically distinct session servers even if the second session server does not have local players connected to it.
- Also with each session being handled differently, the area of interest filtering (Morse et al., 2000) can easily be implemented at the session servers.

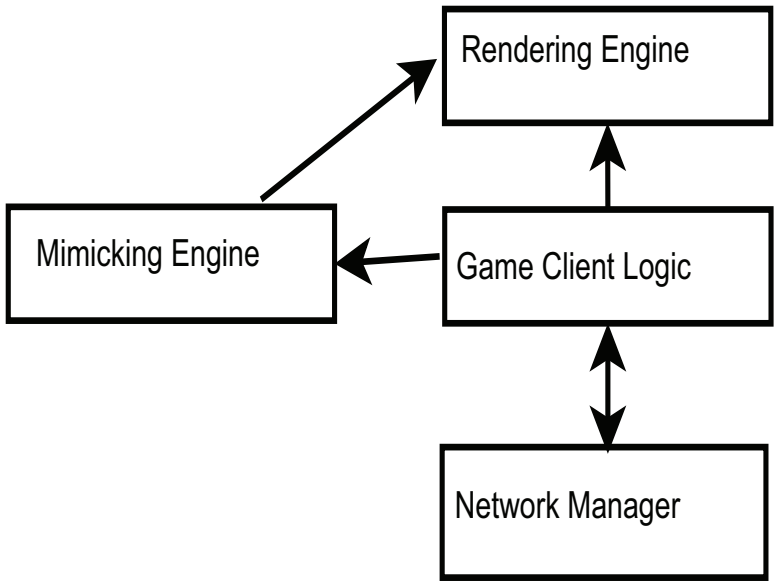


Figure 5.2.2: Client Side Modules for the Game

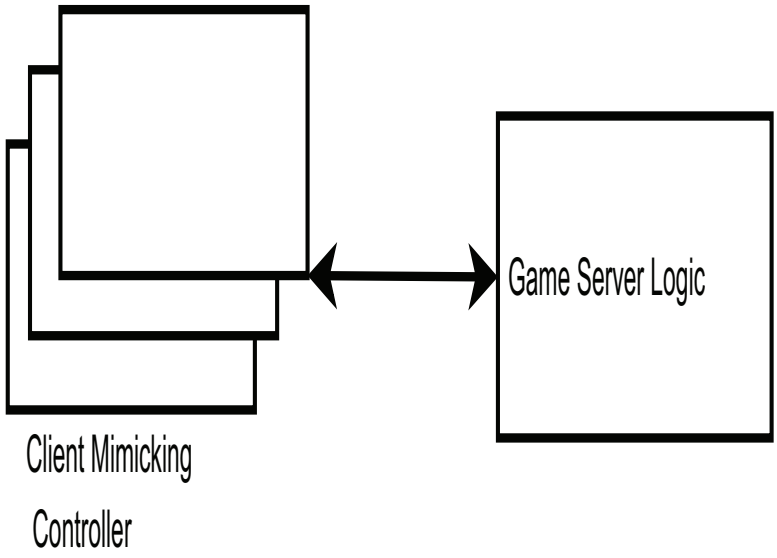


Figure 5.2.3: Session Server Modules

5.3 Consistency Mechanisms and System Architectures

We have discussed different consistency mechanisms in chapter 3 and, in the previous sections, we discussed different architectures. We have seen that the game logic resides on a server in both client-server and mirrored server architectures. Although different proposals given in the literature discuss consistency mechanisms, there is very little work on *where* to apply these mechanisms at the implementation level in the game infrastructure. (Cronin et al., 2002; Ferretti and Roccetti, 2005b) and many others suggest to synchronize data at the game server. However, in a wireless network, there is still a delay between an access point and the mobile device, leading to the possibility that messages arrive in a different order. Hence, we need a consistency mechanism both at the server and client sides. In this section, we first discuss consistency maintenance from the point of view of network architectures. Then, in the next section, we discuss server centric and client centric consistency maintenance approaches. We then propose a protocol to dynamically switch between client and server side consistency maintenance depending upon the conditions and the needs of the game environment.

In a Client-Server Architecture, consistency management is comparatively simple as the data is centralized and the messages from different clients can be ordered using different mechanisms (Bouillot, 2005). Also, interest management techniques (Morse, 1996; Morse et al., 2000) are easier to implement as compared to what should be done in a P2P architecture. A P2P architecture suffers from bandwidth limitations at the client side as well as from the large number of disordered messages arriving from so many different clients each having a different delay with this local client. (McCaffery and Finney, 2004; Hsu et al., 2003; Smed et al., 2002) discuss consistency management in P2P using interest management. Consistency support is based on two components. "Firstly, it relies on the interest management mechanism to identify which state needs to be kept consistent among the number of nodes that share a subset of the same interest. Secondly, it relies on partitioning (the game arena is partitioned among different clients) to allow one authoritative owner of that state who may actively manipulate that state or delegate control of it to another player". The interest management mechanism can be used to detect a player's ability to interact with other entities, and this information can be used to determine if consistency of state between particular nodes needs to be managed. (McCaffery and Finney, 2004) does not detail any particular methods of consistency mechanisms. In Mirrored-Server Architecture, we have both a client-server approach (between a mirrored server and players connected to it) and a peer-to-peer approach (between different mirrored servers). Trailing State Synchronization (Cronin et al., 2002) and Time Warp with local lag (Mauve et al., 2004) are the proposed consistency management in mirrored-server architecture.

When mobile clients are connected to a server through a wireless network, the network delays can vary considerably and jitters can occur. As the game logic executes on the server which normally has some game consistency maintenance and event ordering mechanisms, the update events can reach different players at different points in time because of the variation of the network latency. This can cause inconsistencies between different clients. Also, the server must send the update messages frequently enough so that a consistent virtual world can be displayed on

the client's mobile terminal. This can increase the bandwidth requirement of the server considerably.
In this part of our thesis, we propose an adaptive protocol for hybrid consistency maintenance (i.e. both at server and client), which adapts itself to the varying network latency and to the changing game consistency requirements to achieve consistency among different players. This adaptation takes place at run time and decides whether a consistency mechanism is required on the mobile client's side or not.

5.3.1 Server Centric Approach

As discussed before, two main categories of requirements driving the developments in proposing the client-server architecture are:

- For game creators, the deployment of a game on a large category of terminals should not conduct to additional development costs,
- For players, the game experience (mainly measured in the game reactivity and loading times) should be similar or better compared with a locally installed and executed game.

The main idea is mainly to execute all the game logic on the server while the clients should only execute graphical interface related components.
In the case of multiplayer gaming, synchronization between different players is directly ensured by the server by controlling at each step the scene graph of each terminal. It means that all players will always see the game in the same state. Therefore, the use of techniques for synchronization between the clients is not needed. The main drawback of the proposed method is the sensibility to the network latency. Big latencies can cause different players to view the game in different states at the same time. This can cause disadvantages (in terms of fairness) for the players that see the state later than the other players (Zander et al., 2005). To solve this synchronization problem, it is necessary to have a thin layer of the game logic on the client side itself for consistency maintenance. In the next section, we discuss the client side consistency maintenance approach.

5.3.2 Client Centric Approach

In this section, we discuss how consistency is maintained in a client-server architecture when the logic of the game resides on the client side. In this case, the game logic is totally on the client side and the server performs only message passing and some administrative works such as maintaining a database of players, their accounts etc.
We consider the clients to be playing on mobile devices such as mobile phones, PDAs, and laptops. The heavy-client approach, as we call it, gives the player more control over the game and reduces the bandwidth requirements on the server side.

This approach has the following advantages over a thin-client approach:

- With the deployment of all the game logic on the client side, we do not need high capacity servers or mirrored servers dedicated to execute the game logic (Cronin et al., 2002).

- Because all the logic is on the client side, the clients now need to send messages to other clients through the server less frequently and only when required, using dead-reckoning algorithms (IEEE, 1995). This resolves the bandwidth issue which could be a bottleneck in case the server has to send messages for displaying on the clients at the frame rate.

This approach has some disadvantages:

- Because of the heterogeneity of mobile devices, it is difficult to develop a game that runs on so many different devices which have different memory capacities, different screen resolutions and are connected to the server through different networks such as WIFI, Bluetooth and/or GPRS.

- Because of the player's complete control over the game, he can add cheating mechanism thereby changing the end game results.

- As the game logic resides on the client side, the total delay is equivalent to client-to-client delay which could be approximately double that of server-to-client delay. In case of high network latency, the resolution of state inconsistencies becomes even more difficult.

In chapter 3, we discussed a dynamic consistency maintenance mechanism for high latency mobile multiplayer games (Khan et al., 2008). In this approach, we combine two different synchronization schemes namely dead-reckoning (IEEE, 1995) and local-lag (Mauve et al., 2004). In case of dead-reckoning, there is a prediction model at the sender's side to predict the position of this local player as displayed by the remote player. When the error between the predicted model and the real position exceeds some threshold error value, the player sends an update message to the remote players. Normally this threshold error is fixed for the entire duration of the game and for all objects. In the local-lag approach, we delay the display of local messages for a certain time, hoping that during this lag time the update message from a local player will have reached a remote player and hence consistency will be maintained. This local-lag value is fixed and is the same for all objects and for the whole duration of the game. A more flexible and adaptive approach is therefore needed.

5.3.3 A Hybrid Client-Server Approach

From the above discussions, it becomes clear that both server-centric and client-centric approaches have advantages and disadvantages. For this reason, we propose a hybrid approach combining thin and heavy client architectures (Khan et al., 2010). This combination is made adaptive in that the system can decide at runtime according to the game and context requirements what part of the game logic is executed on the client side. In the case of a high latency and when the game requires strong consistency, a significant part of the game logic has to be executed on the client side in order to apply consistency mechanisms to reach the same state at different terminals. With low network latency, and when the virtual world objects are at a position/speed where small inconsistencies can be tolerated and/or when the capacities of the client's terminal are limited, the client terminal only uses some display mechanism to show the messages directly arriving from the game server.
This hybrid and adaptive approach works well when we have a complex game with a variety of objects with different paces and a network where delays can vary considerably.
A game can be divided into different regions which will have different consistency requirements. In some regions, for example when a player is far away from other players and his movement is considerably slow, it is not necessary to have tight consistency maintenance and only server updates can suffice. On other complex regions where we have many objects near each other with different speed vectors, a stronger consistency maintenance is needed, which can only be done on the client side.
We recommend initializing and dynamically changing the values of dead-reckoning threshold and local lag according to three criteria:

1. When the network conditions change. For example, when the delay increases because of a sudden heavy traffic in the network or when some jitters occur.

2. When a player enters a critical region (Khan et al., 2008) in the game's virtual world, we change the values of dead-reckoning threshold and local lag so that we send messages frequently and do not delay messages for a long time to achieve strong consistency in these critical regions.

3. To fix the values of these two parameters according to the requirements of different objects. For example, to have different values for fast moving objects and slow moving objects in the game world.

We now present an adaptive communication protocol to allow client-server mechanisms to change dynamically their behaviour according to the above mentioned criteria.

5.3.3.1 Server Side Protocol

As shown in Figure 5.3.4, a server has two main components: a Game Logic component and a Communication Controller. The game logic is further divided into differ-

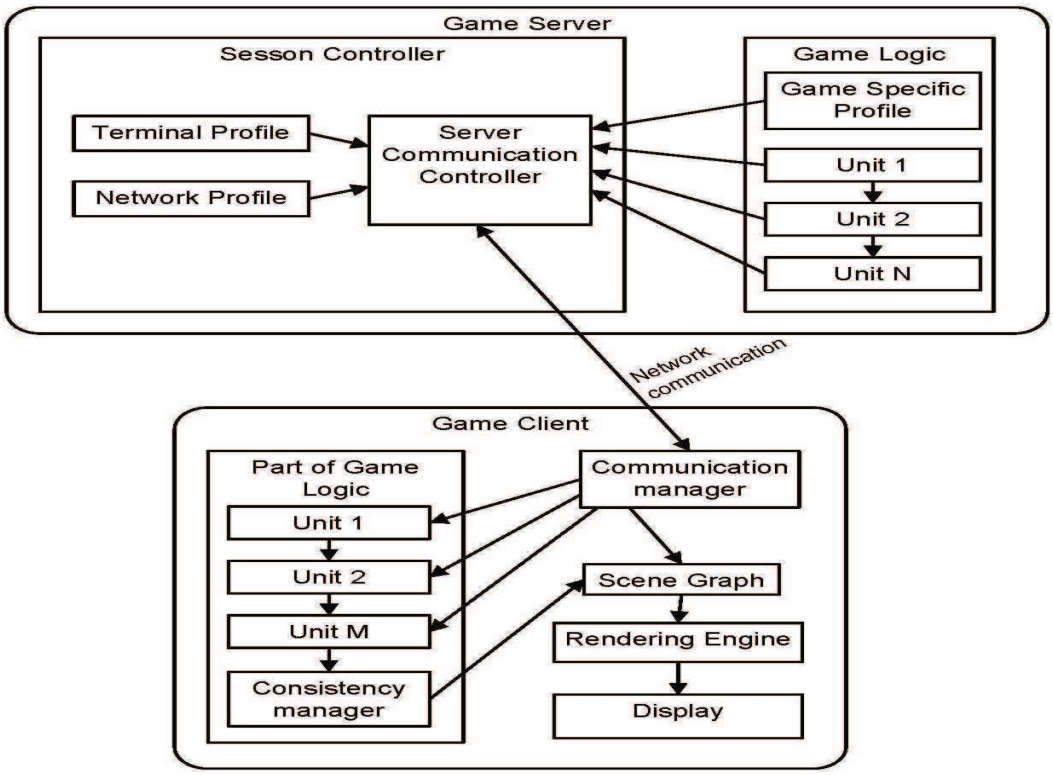


Figure 5.3.4: Hybrid architecture for client-server mobile multiplayer games

ent units necessary for the successful execution of the game world. The Server Communication Controller is responsible for the communication with different clients (players) playing this game. While communicating with different clients, it sends the messages to the clients according to three criteria:

1. If a client, to whom the server is sending the message, is having limited memory, processing speed or screen resolution, it sends only scene graphs to be displayed directly by the limited capacity device. In the message, it signals to the client that no game logic processing is required on the client side. This information is stored in the terminal profile which registers the capacities of different clients before the game starts.
2. In case the network latency is very high, or there are jitters on the communication delay with a certain client, the server informs the corresponding client about the high latency/jitters. Because of high latency, inconsistencies can occur at different clients. These clients, with the feedback from the server, have to apply the necessary consistency mechanisms which we explain in the next section. The information about the network conditions is stored in the network profile and the server checks it before sending messages to the clients.
3. The consistency requirements of a game depend upon different parameters of a player or object such as position, direction and speed in the virtual world. For example, if an object is solitary in the virtual world and its speed is not very high, it may not need a strongly consistent view of others. The information about different game zones, e.g. a circle around a player, is stored in the game profile for each player. If a player has no other player in their critical zone, there is no need for strong consistency on the client side. Hence, in this case, the server should signal the corresponding client that it does not need to apply any consistency maintenance mechanism and that only the scene graph should be updated and be displayed by the rendering engine.

5.3.3.2 Client Side Protocol

On the client side, there is a Communication Manager component, a Game Logic component, and a Rendering Engine for displaying the game world. The game logic on the server side is only a subset of the game, that is $M < N$ in the diagram. The game logic contains only those components which are necessary for consistency maintenance. For example, to use dead-reckoning algorithm for consistency maintenance on the client side in case of a car racing game, it is necessary for the client to have a car track component and some other components to do the necessary predictions on the client's side. The communication manager is the component which receives messages from the server and decides according to the signals/information for the server whether to do some necessary consistency maintenance or not. As mentioned above, the client needs consistency maintenance on its side according to three criteria: Terminal capacity, network conditions and the player's position and speed in the game world.
If consistency maintenance is required on the client side, the communication manager first sends the message to the game logic which processes the message and

applies the necessary consistency maintenance algorithms. After that, the message is used to update the scene graph to be displayed by the rendering engine. If no consistency is required, the communication manager sends the message directly to update the scene graph and renders it.

5.4 Evaluating our Adaptable Approach

We have done a first evaluation of the hybrid and adaptive approach we propose. We evaluated our approach using a car racing game that we developed for this purpose. The game was developed in J2ME, on the top of the GASP platform (Pellerin et al., 2005) and can be played on all mobile phones supporting Java. We have already tested it on Nokia N93 mobile devices connected to a GASP server via a Wi-Fi network. The game can also be played over a cellular network. The GASP server was deployed in tomcat running on an intel machine with 2.5 GHz processor and 3.5 GB RAM. The operating system used was Microsoft Windows XP. Each experience was run at least ten times and average values were calculated which were used to show the results of our algorithms.

We have evaluated the client and server side consistencies in case of high and low latencies. Figure 5.4.5 shows the dynamic switching of the game architecture from thin-client to heavy-client mode and vice versa. In the Figure, at time $t0$, when the latency is low, the game logic and consistency maintenance algorithms run on the server side as represented by the empty wheel and the client only displays the messages through its rendering engine. At time $t1$, when the latency is high, the client executes some part of the game logic necessary for consistency maintenance, represented by the crossed wheel, to hide the high latency from the user.

At time $t2$ the latency is again low and the system switches to thin-client mode. At time $t3$, the player has entered the critical region e.g. the player's car is approaching the finishing line. Hence, although the latency is low, the system switches to the heavy-client mode to achieve strong consistency in this critical region. Figure 5.4.6 shows our evaluation for three different scenarios for a car racing game using only two cars.

The 'Original Position' curve shows the actual positions of a local player on the system on which the game logic is running. The positions values, in pixels, are drawn on the Y-axis against the time shown on the X-axis. Time value 0 denotes the start of the game. As the car starts moving towards the left of the screen on a mobile phone, the values of Y (position in pixels) decreases until it reaches zero which denotes the finishing line. The "RemoteWithlowLatency" curve shows the positions of the car on the remote client, when the latency between the client and the server is very low and all the messages coming from the server are shown directly on the client screen, i.e. without applying any algorithm and without utilising any game logic on the client side. In this case, the difference between the actual car positions and the displayed car is minimal. In the case of a high network latency (between 1000 and 2000 milliseconds in our implementation), the difference between the actual car positions and the displayed car on the client side becomes quite visible when no client side mechanism is applied, as shown by the "HighLatencyWithNoClientConsistency" curve. In case of high latency and/or

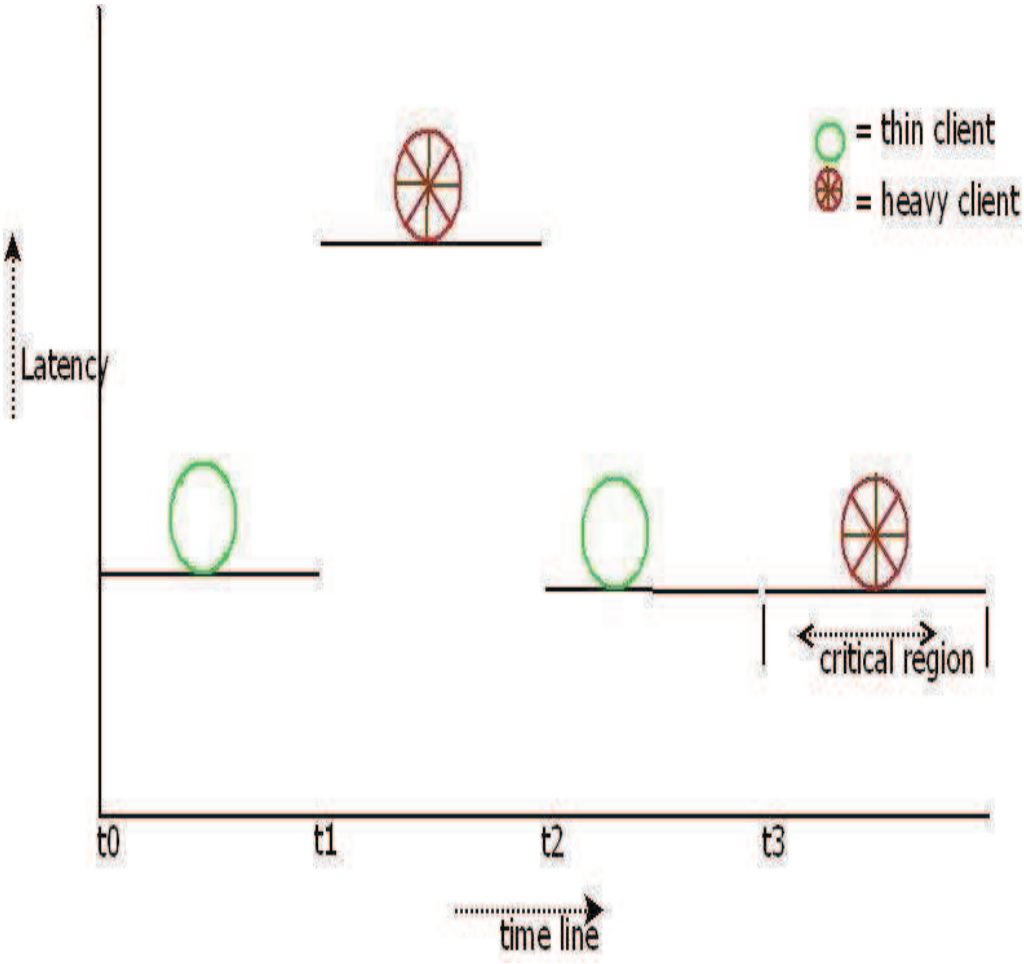


Figure 5.4.5: Dynamic adaptation of the game architecture during the runtime

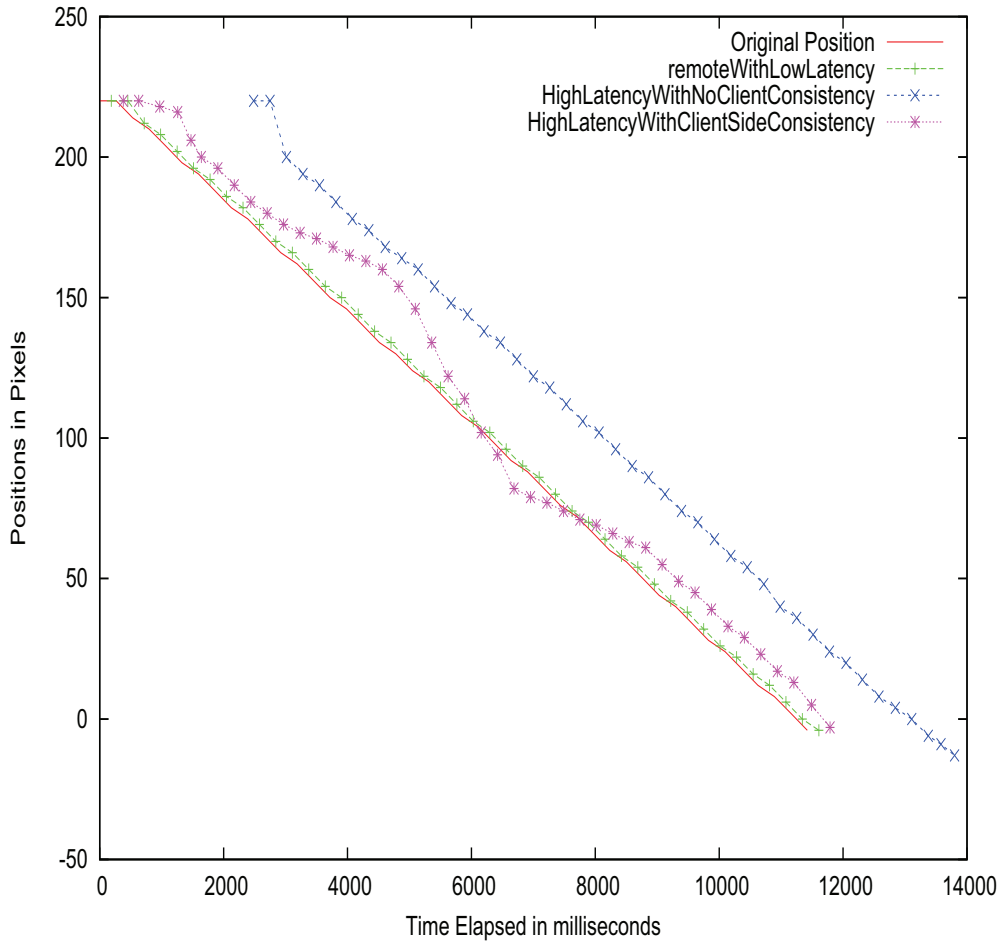


Figure 5.4.6: Comparison of client and server inconsistencies with high and low latency

high speed of the car, we need to apply some client side consistency maintenance to synchronize the data between the server and the client. This result is shown by the "highLatencyWithClientSideConsistency" curve. This curve is closer to the 'Original Position' curve than the one in which the latency is high and no client-server consistency maintenance algorithm is used. This curve is not as straight as the others because when applying prediction algorithms, prediction errors can occur, which need some recovery time to arrive at the correct position.

From the figure, we can safely argue that in case of very low latencies and when the objects move at a slow pace, messages coming from the server can be displayed directly on the client screen without the fear of high inconsistencies. On the contrary, in case of fast-moving objects and high network latency, we need some mechanism on the client side to do the necessary consistency maintenance for which some part of the game logic must reside on the client terminal.

5.5 Concluding Remarks

In this chapter, we first discussed different architectures and proposed a scalable system architecture for mobile multiplayer games. We then proposed a protocol for maintaining consistency between game states on clients and servers by dynamically switching to client side consistency when necessary. We evaluated our results with a multiplayer game and showed that better consistency can be achieved by applying mechanisms at clients' terminals in case of high latency and/or high game consistency requirement at that point in time. We showed that both the client and server side consistency maintenance approaches result in better consistency than the simple server side consistency mechanism in case of high network latencies.

In the future, we would like to experiment with our session server architecture and compare it against other architectures such as the mirrored server architecture used for distributed virtual environments.

Chapter 6

Synchronization Medium Architecture

6.1 Introduction

In chapters 2 and 3, we discussed different consistency maintenance algorithms for state synchronization in distributed virtual environments such as multiplayer games. We saw that a combination of these algorithms can be used to possibly reach global consistency. These algorithms are very complex and are, therefore, hard to program. The intermixing of the synchronization code with the game logic code makes the evolution of the game code difficult over time. It is thus desirable to separate the code of these algorithms from the game logic and to isolate it in a specific module. In the section 3 of chapter 2, we discussed the concept of Medium, which is a communication component, recently proposed (Cariou et al., 2002) to deal with interaction and distribution as non-functional aspects. In this chapter, we extend this concept by inserting prediction and synchronization algorithms into this communication component and we call this component a *Synchronization Medium*.

By using a Synchronization Medium, a game programmer can concentrate on the game logic, and is relieved from having to deal with the synchronization and communication concerns. We also observe that synchronization is an off-shoot of the communication delays and hence it is better to deal with it as a communication concern rather than as a game problem.

Apart from handling consistency management, another advantage of a Synchronization Medium is its reusability. It offers generic interfaces that do not change when replacing a synchronization algorithm by another one inside the medium. A same medium can thus be used by different game applications; the application code is not impacted even if a different synchronization algorithm that best suits the applications needs has to be plugged into the medium. This is an important property of the medium that can be used for dynamically adapting applications. With mechanisms for dynamic adaptability inserted inside the medium, it can adapt itself by using different algorithms according to the context of the game. As we saw in the previous section, there are different types of network infrastructure employed by

the game companies, each having different types of communication protocols and consistency maintenance requirements. We believe that by having different Synchronization Mediums for different types of architectures can facilitate the reuse of the same game code over different platforms.

In this chapter, we discuss the separation of the code of the consistency algorithms from the game logic and their insertion into Synchronization Medium, a distributed communication component responsible for non-functional, non-game issues. We present the design of such a communication component integrating different synchronization algorithms. We first give an overview of the medium as a communication component.

6.2 Overview of the Medium: A Communication Component

Presented in (Cariou et al., 2002), the concept of communication component or *medium* is to separate interactional details from the functional details of a component. These interactional details can be handled separately by the medium. Hence a medium is the reification of an interaction, communication or coordination system, protocol or service in a software component. This architecture has the advantage that the medium can be reused for different types of application.

A medium, like any software component, can take different shapes according to the level at which it is considered. It exists as a specification describing the communication abstraction that it reifies, but also at implementation and deployment levels. It is indeed possible to manipulate a high level communication abstraction at all the stages of the software development cycle. At the specification level, a medium is specified using a UML class diagram. Then a refinement process transforms this specification into a low-level implementation design. This refinement process is carried out in three phases. In the first phase, for each component interacting with the medium a class called *Role Manager* is produced. This class is responsible for all the interactions with its corresponding component. A medium is an aggregate of these role managers. In the second phase, the class representing the medium is removed, and only the role managers are left interacting with their corresponding components and with each other. Depending on the non-functional constraints, this phase can lead to many design choices (Cariou et al., 2002). The third and final phase defines, for each design specification in the preceding phase, one or more deployment diagrams, describing how different role managers and components are distributed and grouped at the time of the deployment of a medium.

In the next section, we discuss the insertion of synchronization algorithms in the medium.

6.3 Synchronization Medium

We consider that synchronization is a non-functional issue which arises due to communication delays, and hence it would be better to deal with it separately from the game logic. We present in this section the design of the medium which handles

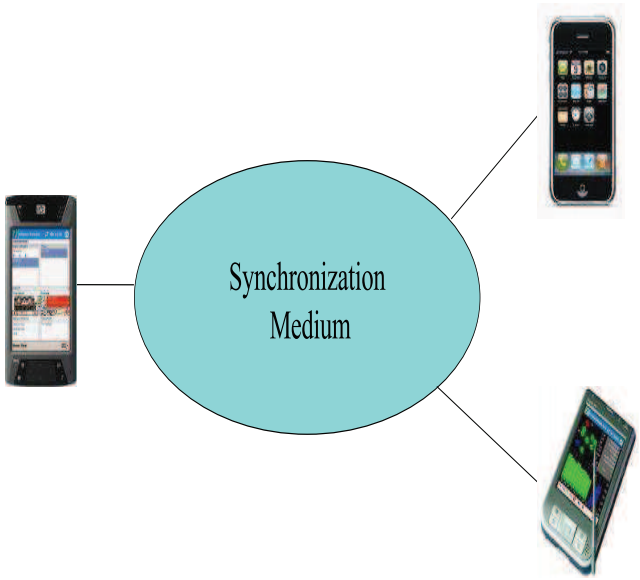


Figure 6.3.1: Synchronization Medium

synchronization as well as communication aspects of the distributed virtual environment. We call this medium a *Synchronization Medium* (Khan et al., 2007). It allows to hide from the game clients, the latency compensation and synchronization mechanisms. Thus a Synchronization Medium is an abstraction of the communication for data consistency across a network. This has the advantage to offer game developers a synchronization tool that can be picked up and used directly. Handling latency hiding and synchronization in a medium has the advantage that the client will see as if the remote clients are available locally through interfaces unaware of the network latency and its variation in the form of jitters, and of the complex issue of consistency maintenance.

This idea is shown in Figure 6.3.1 in the case of a deployment on heterogeneous devices. The Synchronization Medium is logically a single component which is distributed physically across the network offering services required by the components interacting with it and requiring services offered by the interacting components. A two-way interaction actually takes place between the Synchronization Medium and application components.

An abstract specification of a Synchronization Medium is given in figure 6.3.2. Two components with the *Player* and *Administrator* roles interact with the medium. The Player role corresponds to the game client residing on the mobile terminal and uses the "IplayerMediumServices" services offered by the medium. The "IPlayer-MediumServices" interface offers services like provision of information regarding a new player as soon as it joins the game session and the reception of update messages from the remote players. These services are implemented through functions such as *getNewPlayer()* and *updatePlayer()*. The *Administrator* role is the game server which uses the "IAdminMediumServices" services offered by the medium. These may include administrative services such as monitoring the game to prevent cheating and to charge fees from the users. The medium may need some services to

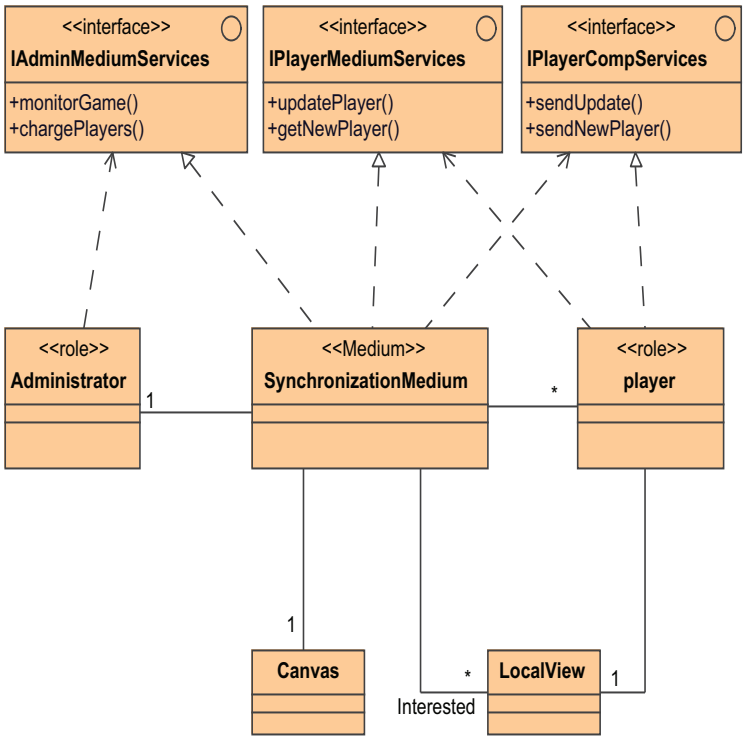


Figure 6.3.2: Abstract specification of Synchronization Medium

interact with the player. These services are offered by the "IplayerComponentServices" interface of the *Player* role. The *Canvas* class represents the overall game data, while the *LocalView* class corresponds to the part of the game data that the player is supposed to receive. A player may, indeed, not be interested in all the game data but only in a subset of the canvas such as through the use of interest management.

As mentioned in the previous section, in the second phase of the reification process, the medium is represented as an aggregate of role managers. This is shown in figure 6.3.3. There are two role managers namely "Player Manager" and "Game Manager". The "Player Manager" is the representative of the medium on each game client and offers/requires services offered/required by the game client. In the case of a client-server game, the two role managers may communicate through a middleware. The set of all role managers on all nodes constitute the *Synchronization Medium*

Synchronization algorithms are then integrated into the medium as internal services which will be invoked by the medium transparently to the player.

We now discuss the design of a Synchronization Medium with different choices of synchronization algorithms.

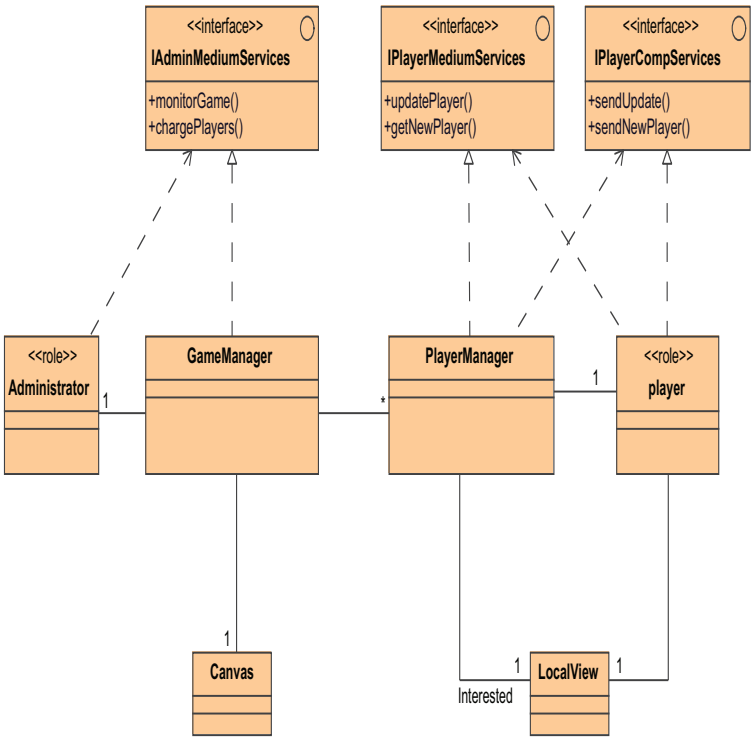


Figure 6.3.3: Introduction of role managers

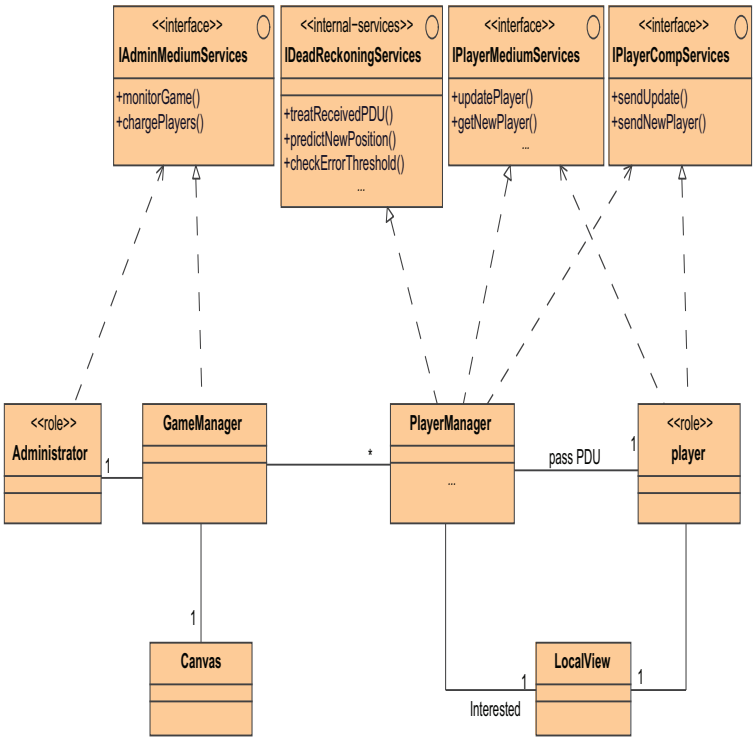


Figure 6.3.4: Synchronization Medium using dead-reckoning algorithm

Figure 6.3.4 shows a class diagram for a medium using a dead-reckoning algorithm. The medium receives from a remote player a Protocol Data Unit (PDU) concerning a remote object. The PDU contains information that uniquely identifies an entity, such as its position and velocity. The PDU may contain an identifier telling which dead-reckoning algorithm to use. The Player Manager passes this PDU to the "IDeadReckoningServices" interface of the medium. This interface is an internal interface of the medium because it is not used directly by any component interacting with the medium. This service predicts the new position of the entity using its *predictNewPosition* function and passes it back to the Player Manager. This updated position is then passed to the player via the "IPlayerMediumServices" interface.

In the case of a PDU emitted by the local player, it is first passed to the local Player Manager which, in turn, sends it to the remote players. Before passing a PDU to the remote Player Manager, the *checkErrorThreshold()* function is called to check whether the predicted value is different from the actual value by a certain margin (this is called dead reckoning threshold). A PDU is passed to a remote Player Manager only when the *checkErrorThreshold* returns true.

Note that a PDU passed by a player is received by a remote Player Manager and not the remote player itself. Hence, the process is hidden from the player component in the game logic. A collaboration diagram showing the dynamic view of how the messages are passed during the dead-reckoning process between a player and the medium is shown in figure 6.3.5. The interface connecting a role manager and a player is an external interface represented by a small black rectangle, while the interface between the local and the remote role managers is an internal interface of the medium represented by a small white rectangle. Similiarly, DeadReckoningServices is an internal service of the synchronization medium and hence its interface is internal as indicated by the two small white rectangles.

Another choice could be to use the Perceptive Consistency algorithm (Bouillot, 2005) within the medium for maintaining the consistency between different players. A diagram showing the Synchronization Medium using the PC algorithm is presented in figure 6.3.6. A player passes an artificial message (u, t(u)), u being an artificial update issued at time t. The Player Manager receives such an artificial message from the remote players and passes it to the "ICalculateLocalVector" services of the medium. As discussed in "Perceptive Consistency" part of chapter 2, this process calculates the latency between the local player and all remote players. This service returns a vector containing these delays. The player manager then passes this vector to the "ICalculateLocalLag" service, which calculates, through its *CalculateLocalLag* function, the local lag to be introduced locally, and returns it to the "PlayerManager" via its *returnAdjust* function. The *adjust* factor is passed to the player through the "IPlayerMediumServices" interface of the medium.

The basic difference between the two mediums, one using dead-reckoning and the other one using PC, lies in their internal services. The interfaces with the outside components remain the same. Thus, for example, the code for the game client almost remains the same except for the small changes for the type of message to be passed to the Synchronization Medium.

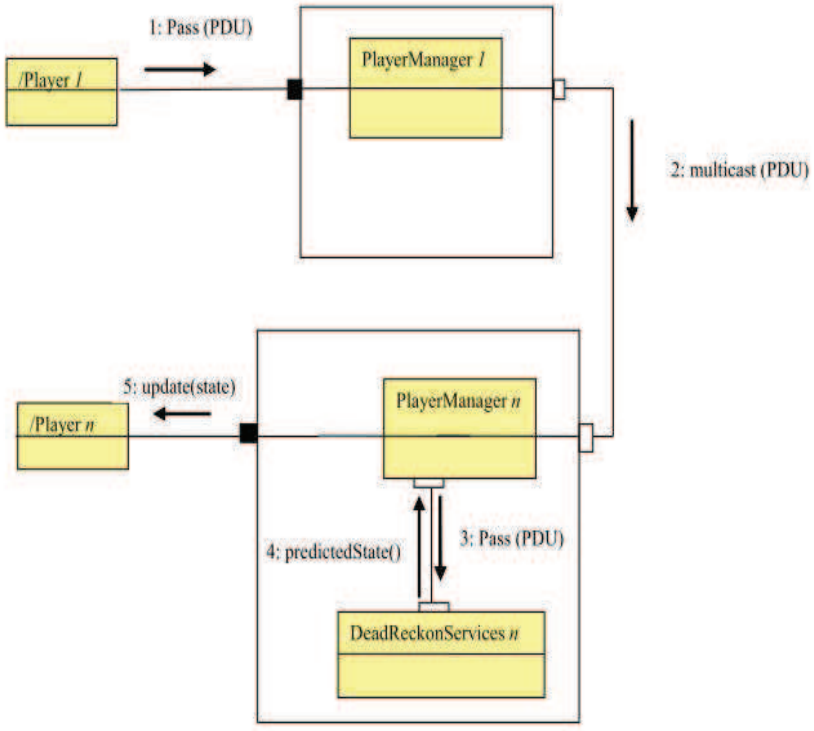


Figure 6.3.5: Dynamic view of message passing in case of Dead-reckoning algorithm

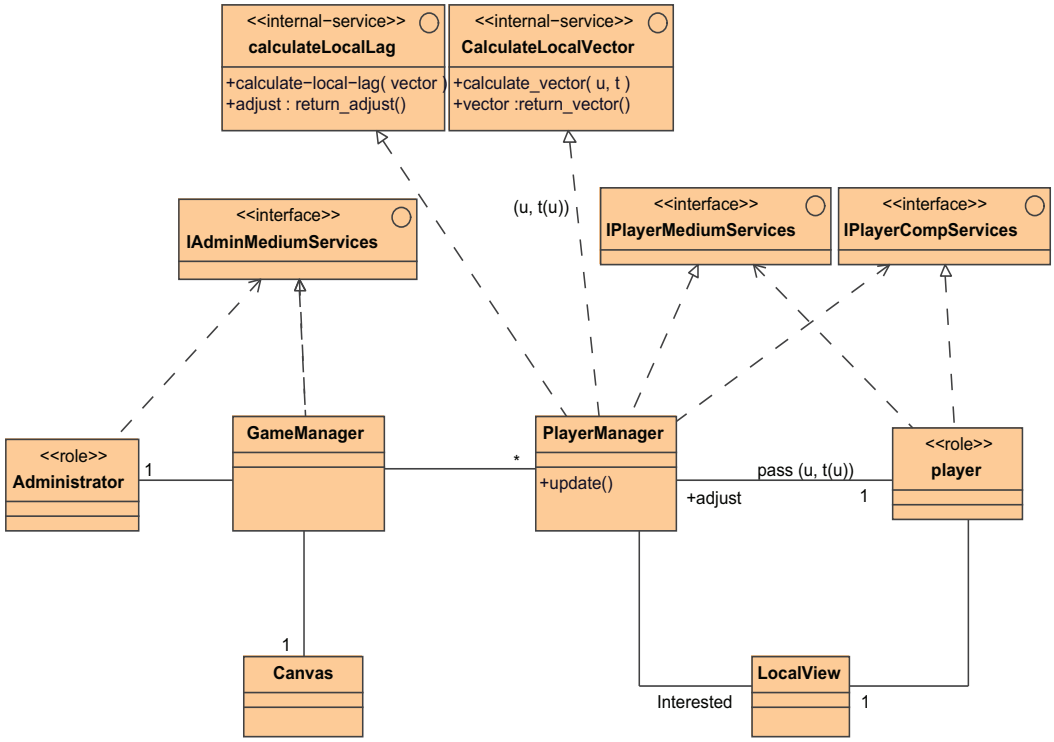


Figure 6.3.6: Synchronization Medium using PC

Note that it is possible for a medium to use a combination of synchronization algorithms for the same game. For example, Perceptive Consistency can be combined with dead-reckoning to compensate for high latencies in mobile networks. Dead-reckoning can help to provide predicted intermediate states during the local lag period introduced in PC.

Also variations of the same algorithm can be used for the same application depending on the context. For example, a dead-reckoning algorithm using position-based history can be used when the motion of an object is not smooth, but predictable over a certain pattern, and a single update based dead-reckoning can be used when the motion of the object being predicted is smooth.

A Synchronization Medium has different responsibilities as we saw in this section. We believe that a Synchronization Medium can be a composite component each having distinct responsibilities. Before going into the detailed architecture of a Synchronization Medium, we first give a classification of multiplayer games according to their consistency requirements.

6.4 Classification of Game Applications

In this section, we classify different types of games according to their consistency requirements. We believe that this classification of game types is necessary so as to have different re-usable Synchronization Mediums for different types of games. A Synchronization Medium has interfaces (offered and required) which are used by the game application. As each class of games share common synchronization approaches, this classification can help us define common interfaces for Synchronization Mediums used by each category. Hence the Synchronization Medium for a given category becomes re-usable for all the applications offering general interfaces irrespective of the internal synchronization mechanisms. Also by this classification, a game developer will be able to better select a Synchronization Medium for their game plugged with a particular consistency approach better suited to the game he/she is developing and configure it to his/her own consistency requirements. This classification is also necessary because, as we will see in the next section, we have to inform the medium about different regions in the game, and the coordinates & orientation of these games depend upon the type of the game. This classification does not include certain interactive applications such as turn based games, shared white boards etc, because their synchronization techniques are different from continuous applications such as multiplayer games and military warfare simulation and hence they are out of the scope of this report.

6.4.1 Field Applications

In Field Applications, the players play on a specific field. This field is fixed and does not change its location. Instead, players move from one position to another in the field in order to play the game. Examples are Football, Hockey, Tennis games etc. In Field Applications, there are some regions in the field where we need strong consistency because an error of judgement during prediction in these areas can result in an incorrect state from the point of view of different players and can

affect the result of the game. We categorize these regions as critical regions that we proposed earlier (Khan et al., 2008). For example, in tennis game, the area near the base line is a critical region. When a ball coming from the opposite side, lands near the base line, an error in prediction can cause display a ball as landing outside that could have actually landed inside and vice versa. The two players can disagree about the results and hence the game play will not be fair.

6.4.2 Racing Games

Racing games form a subclass of Field Applications. In this type of game, there is a fixed track and the players' avatars or other objects move on the track to reach to the end of the track. In racing games, the end of the track is a critical region because it is here that decision about the winner of the game will be taken.

6.4.3 First Person Shooters

In first person shooter games, there are different objects and targets which can be fixed or movable. Because of the complexity of this type of games, there are many fixed and movable critical regions. Because of the high variety in this type of games, it is difficult for a Synchronization Medium designer to provide a medium with already defined critical regions. Hence through a Synchronization Medium interface, the game developer has to inform the medium about the coordinates of critical region(s) in his/her game.

6.5 Components of the Synchronization Medium

A Synchronization Medium is a composite component containing subcomponents. It interacts with the game application by offering some services through its interfaces, and also requests services from the game by using the game services through the game interface. The part of the Synchronization Medium residing on a client side is called the Player Manager. The set of all the player managers on all the clients playing a game forms the Synchronization Medium.

Each player manager is a composite component containing different Managers. Figure 6.5.7 shows the internal structure of the medium and its interaction with the game application. In this section we discuss different managers of the Synchronization Medium, how they interact with each other, and how they communicate with the game application and the underlying network infrastructure.

6.5.1 Critical Area Manager

The Critical Area Manager (CAM) is responsible for the processing of all information related to the critical area(s) in a game. As we discussed in (Khan et al., 2008), a critical region is a region in the virtual environment of the game where strict consistency mechanisms are needed to achieve better results. The CAM takes

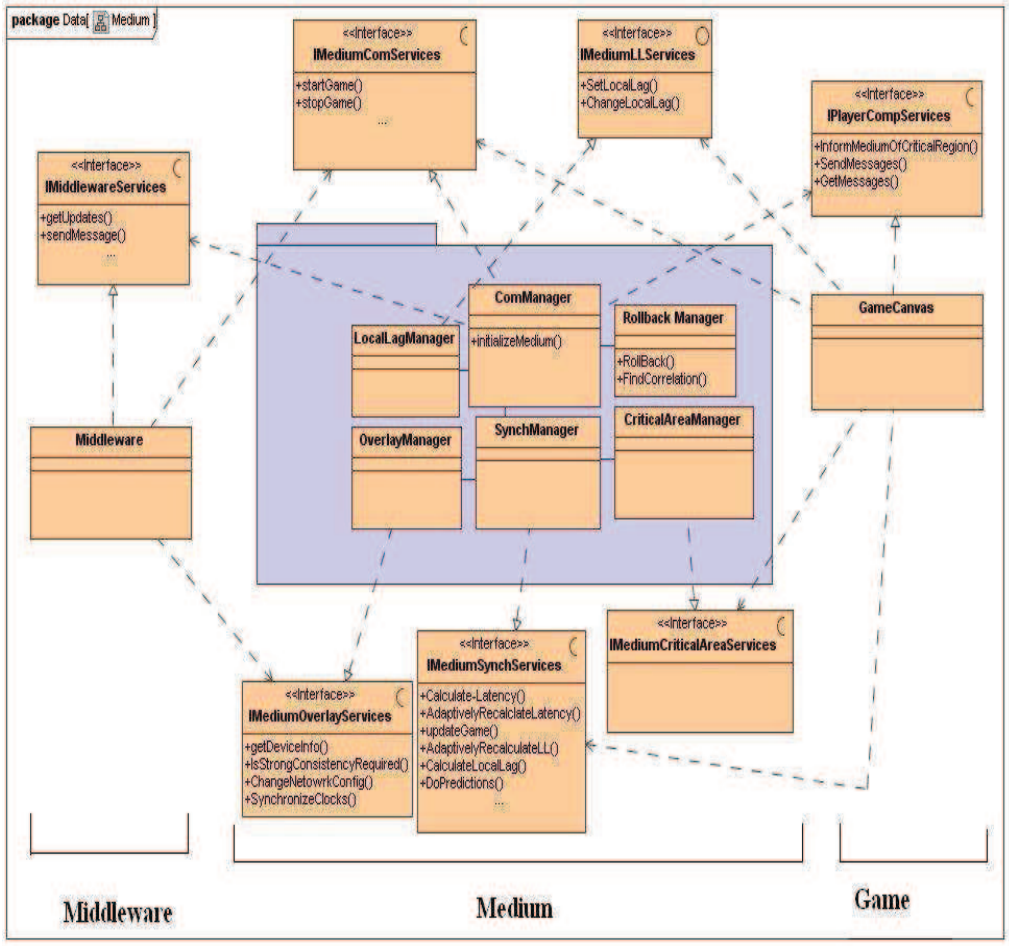


Figure 6.5.7: Detailed Architecture of the Synchronization Medium

information about the critical region from the game and uses this information during the consistency management process. The CAM offers its services through its "IMediumCriticalAreaServices" interface. This interface is used by the game application to send information about the critical area to the CAM. This information can be the coordinates of the critical area such as area around the base line in a tennis game or it can be the center of a circle of a certain radius around a player. In the latter case, the critical area is not fixed, but instead moves along with the player.

In the critical regions we need to change our technique of synchronization because we need strong consistency in these areas instead of loose consistency as can be tolerated in some other *less important* areas of the game environment. When a player enters a critical region, the CAM informs the Synchronization Managers (section 6.5.3) about the players ID and its motion (speed, direction etc). It is then the responsibility of the Synchronization Manager to take the necessary actions so as to achieve strong consistency in that area.

6.5.2 Communication Manager

The Communication Manager (ComManager in the figure) is responsible for the communication of general services between the application logic on one client and the rest of the clients through an underlying middleware or network infrastructure. This includes, for example, the starting and stopping of the game application. It offers services to the underlying game middleware to communicate information with other remote game clients. This manager also adds a time stamp to each emitted message that is being sent to the remote clients so that the remote clients could process the arriving messages in temporal order. We suppose that all clocks are synchronized using some clock synchronization mechanism such as the Network Time Protocol (NTP) (Mills, 1989) or through the Global Positioning System (GPS) (Sterzbach, 1997)

6.5.3 Synchronization Manager

The Synchronization Manager (SynchManager in the figure) is the part of the medium which is responsible for synchronization of game information and state consistency management. A first hand separation of the synchronization concern is the calculation of network delays. We think that this should be the responsibility of the Synchronization Medium and not the game itself to calculate the network delays. The Synchronization Manager is the medium component that calculates network latency. Based upon these latencies, the Synchronization Manager takes different actions such as prediction according to a certain Dead-reckoning algorithm. As we have shown in (Khan et al., 2008), the network delays may change during the game play because of the network load. Therefore, the Synchronization Manager calculates the network latency not only at the start of the game, but also during the game session. This re-calculation of delays can be periodical or it can be reactive to certain network conditions such as an increase in the network load etc. If the Synchronization Medium is aware of the game map, such as the racing track shared with it by the game application, it can do the necessary prediction and then

pass the predicted message to the game application. If the prediction is not possible in the Synchronization Medium, because of insufficient game canvas information on the part of the medium, then it is the responsibility of the game application to do the predictions there. The game application uses the "IPlayerMediumSynchServices" interface of the Synchronization Manager to use the services offered by the medium for the purpose of synchronization.

6.5.4 Local Lag Manager

The Local Lag Manager is responsible for taking the decision about the local lag to be produced locally before the playout of commands (Mauve et al., 2004). As we have proposed the idea of adaptable local lag in (Khan et al., 2008), where the value of local lag depends upon the network and game condition, it is the responsibility of Local lag Manager to fix a value for the local lag and decide when to change this value. As the value of local lag depends upon the object about which the information is being received or displayed, hence it is the responsibility of the game application to inform the Synchronization Medium about the initial value of local lag for that object. Note that different objects in the game world can be assigned different local lag values according to their synchronization needs as we have discussed in (Khan et al., 2008). The value of the local lag is also dependent on whether the object in question is in a critical region of the game or not. If the object is in a critical region, then decreasing the local lag value means using more real-time messages thus reducing the inconsistencies in the outcome of the game in that area.

6.5.5 Rollback Manager

The Rollback Manager component of the Synchronization Medium has the responsibility of keeping the receiving messages in temporal order. It observes all the incoming messages, and if a message m arrives late, then it informs the game through its "IPlayerCompServices" to rollback all those messages that have time stamps greater than that of m i.e. those messages which were issued after m by the emitting node(s) but arrived earlier than m as m was late because of the non-uniform network latency. This rollback can follow the time warp mechanism (Jefferson, 1985) or any other approach such as the correlation based approach (Ferretti and Roccetti, 2005b) to reduce the number of rollbacks on the receiving side. Note that the Rollback Manager does not offer any service to the game but uses game services for rollbacks.

6.5.6 Overlay Manager

The Overlay Manager is the component of the Synchronization Medium responsible for taking those network related decisions which can affect the synchronization. When exchanging messages during the game session, players need to synchronize themselves according to the global state of the game. Based upon their position in the game, different players have different requirements. For example, in a football game, the consistency requirement for the players in the D-area (a critical region)

is stronger than outside the D-area. So there is a need to exchange information quickly between those players which are in D-area as compared to those who are not in D-area.

When exchanging messages during the game, the player sends information about PlayerIDs which are in a critical region. So this information should be sent only to the related players. This can be sent directly (to avoid any delay) or through a server. In case, the IP address of the remote terminal is known, a direct pair-to-pair connection is possible without passing through the server. As an example, in case of playing a soccer game on mobile phones, the Bluetooth technology could be used to communicate with nearby physical players if they are also close to each other in the virtual world. This change of network configuration for the sake of strong consistency during the game play is the responsibility of the Overlay Manager. For the medium to be able to do this, it should collect some information at the start of the game. This information includes remote terminals' IP addresses, whether that terminal is equipped with a GPS, its memory size and processor power.

In case the remote terminal and the local terminal are both equipped with GPSs, the Overlay Manager can be used to synchronize their clocks through these GPSs so that the Synchronization Manager calculates the latency time between them by passing the time stamp in their message exchanges. Thus an Overlay Manager is responsible for taking network related decision during the game play thereby improving the overall performance of the game. The Overlay Manager can also decide to switch from one network to another during the game play. This decision can be based on different factors. For example, switching from a slower network to a faster one, or from a costlier one to a cheaper, etc. As another example, if a player's terminal detects that another player has come in its vicinity, it can connect with that player through Wi-Fi or Bluetooth to decrease communication costs and network latency.

6.6 Communication

6.6.1 Message Reception

Figure 6.6.8 shows how remote messages are received by the Synchronization Medium. All the messages received from a remote client are passed through the Synchronization Medium. The Synchronization medium receives PDU (protocol data unit) from the underlying network or middleware. This PDU contains information about the remote player's ID, its position, direction speed and the sender's local time at which this message was transmitted, called time stamp. The message is received by the Communication Manager of the medium. This Manager calculates the delay either from the time stamp of the message in case the clocks are synchronized or through some other means at the start of the game or periodically during the game. This message along with the network latency is sent to the Local Lag Manager. The Local Lag Manager calculates the necessary local lag to be introduced locally before sending the message to the Critical Region Manager which decides whether a switch to strong consistency is necessary or not. The message is then passed to the RollBack Manager to decide if a rollback is required depending upon

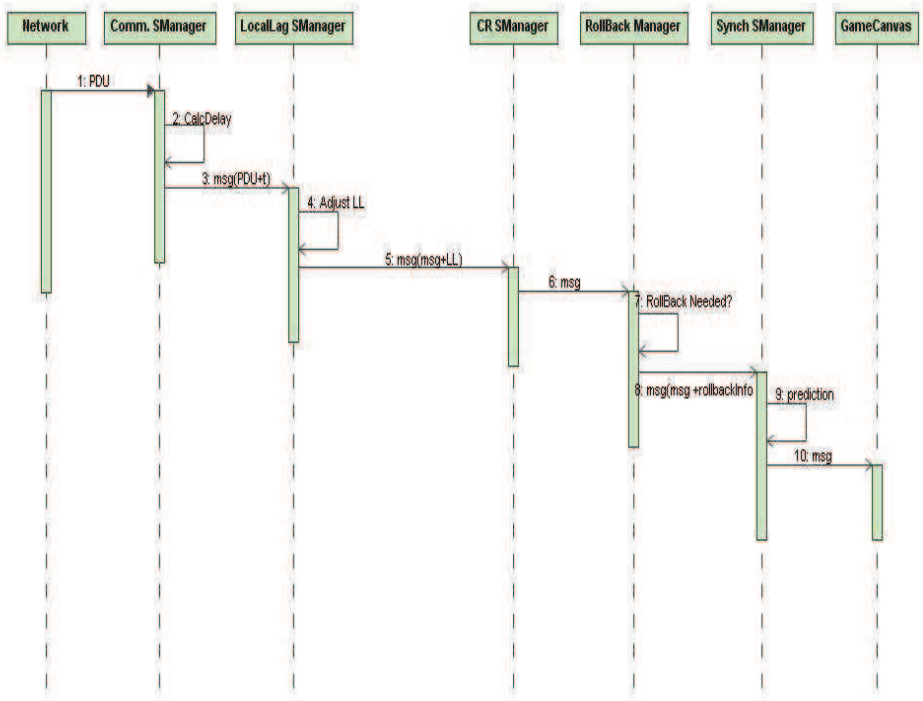


Figure 6.6.8: Message reception by Synchronization Medium

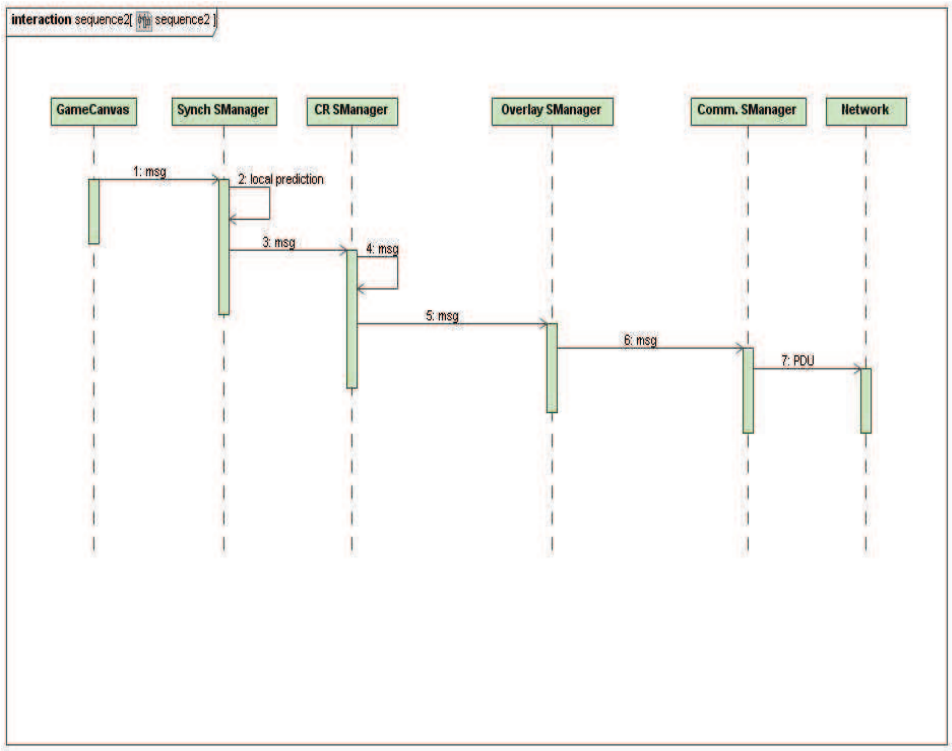


Figure 6.6.9: Message sending by Synchronization Medium

whether the message was received on time or late. The message is then passed to the Synchronization Manager to apply the prediction algorithm before passing to the game canvas.

6.6.2 Message Transmission

All the messages sent by the game client to the remote players are passed through the Synchronization Medium as shown in figure 6.6.9. The message is first passed to the Synchronization Manager to apply local prediction. This local prediction is necessary because it is based on this calculation that the client will decide when to send the next message. The local client sends a message only when a local prediction error reaches a certain threshold, and hence the message sending frequency depends upon this calculation. The message is then passed to the Critical Region Manager which sees if the player is in the critical area or not so that the Overlay Manager can decide if a direct connection, such as through Bluetooth is necessary or not when it is physically possible. The Communication Manager then sends the message to the remote players through the network.

A deployment diagram of a deployment example of a Synchronization Medium in a client-server architecture is shown in figure 6.6.10. The *playerManager* resides

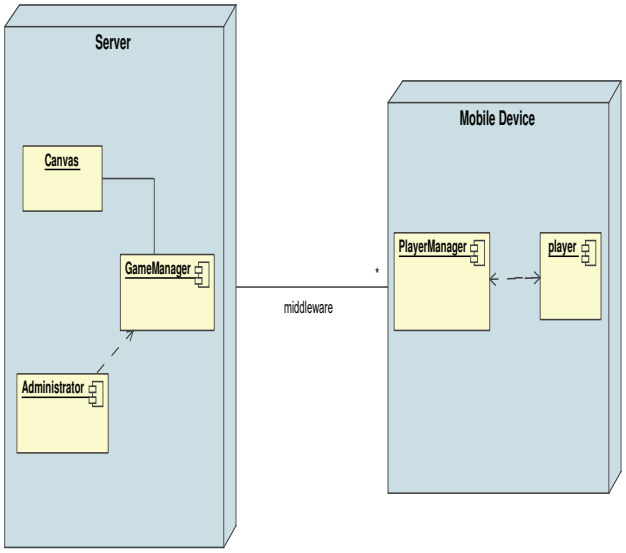


Figure 6.6.10: Deployment of Synchronization Medium in a client-server architecture

on the mobile device while the *gameManager* resides on the server. Depending on the deployment constraints, one can have many deployment choices.

With many deployment variants, a multiplayer game application can adapt at run time when the context of the game changes, thereby using a different synchronization algorithm suitable in that context using the approach in (Phung-Khac et al., 2008). In this approach, many variants of the role managers at the deployment level are integrated into the target adaptive application (a multiplayer game in our case) that can dynamically select the running variant in order to adapt to the changing context. With the help of an adaptation guide generated by the transformation process, the adaptive application can coordinate distributed applications to transfer data of the replaced variant to new one and to maintain the architectural coherence between distributed parts of the application. Hence, a multiplayer game application can correctly adapt at runtime by switching to a new synchronization algorithm without loss of data.

6.7 Concluding Remarks

The primary advantage of the approach we proposed is its reusability. A given Synchronization Medium with a specific synchronization algorithm can be reused many times by different applications. We also believe that separating the synchronization concern from the game logic, and putting it in a communication component, will facilitate game development by letting the developers concentrate on the game logic only. Programmers will not be concerned by synchronization issues. They can use an off-the-shelf medium with a given synchronization technique by looking at

its specification. Hence, the evolution of the game program will be made simpler during the course of time.

Furthermore, the Synchronization Medium will facilitate the experimentation of different synchronization algorithms. Having different synchronization mediums with different algorithms available, one can use them in the same game and compare and analyze the results. Dynamic adaptability mechanisms can also be inserted in the medium thus allowing to have more than one algorithm in the medium and use these algorithms according to the context of the game. For example, a measure of the mobile network latency can help in the choice of the most appropriate algorithm. When the latency is below some threshold but still noticeable by the player, the Perceptive Consistency algorithm can give good results. When the latency increases, PC can be combined with a dead-reckoning algorithm by adding predicted intermediate states during the local lag period introduced by PC.

Chapter 7

Synchronization Medium Evaluation

In the previous chapter, we discussed an approach for separating consistency and communication concerns from the game logic code and inserting them into a synchronization component called Synchronization Medium. In this chapter, we present the evaluation of our work carried out on the Synchronization Medium. We evaluate our Synchronization Medium according to the following steps:

- the implementation of a game using the medium
- showing the reusability of the Synchronization Medium by implementing another game on the top of it
- showing the advantages of the Synchronization Medium in terms of ease of development
- and finally, performance evaluation.

As in figure 7.0.1, we show three different layers, representing respectively the game layer, the medium layer and the network layer.

The game layer represents the logic of the game which uses the underlying medium layer for communication as well as consistency maintenance. The medium either uses the network infrastructure directly or through an underlying middleware to communicate with other remote players. In the next section, we show the implementation of our Synchronization Medium.

7.1 Implementation of the Medium

In this section, we show the implementation of our Synchronization Medium and its use by a car racing game which executes on the top of the Synchronization Medium.

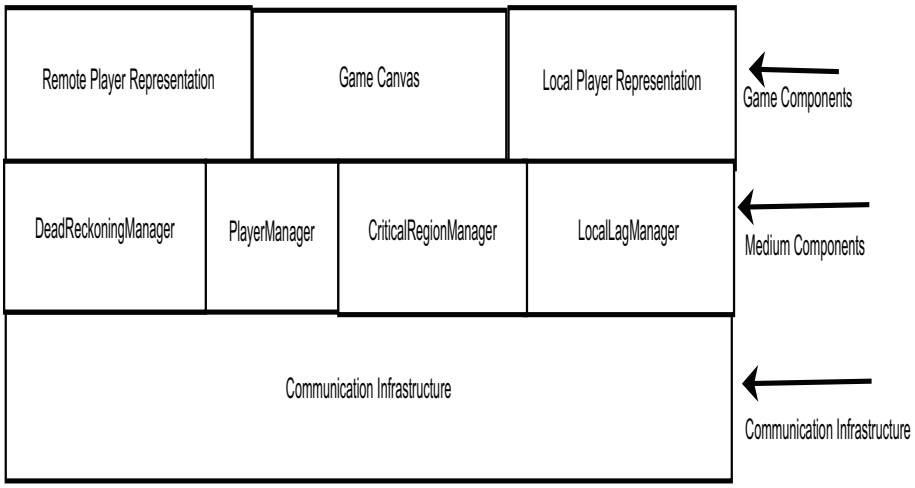


Figure 7.0.1: A Layered Approach to Synchronization Medium

The architecture of the Synchronization Medium is shown in the class diagram of figure 7.1.2

In figure 7.1.2, the medium is formed of four classes: The *PlayerManager* class is the core medium class and is responsible for coordination amongst the rest of the medium classes and also with the game classes. It is responsible for message reception from the remote players as well as message transmission to the network infrastructure. Note that to communicate through the underlying network infrastructure, we use the services of the GASP middleware (Pellerin et al., 2005; Pellerin, 2010). The *DeadReckoningManager* class is responsible for applying the dead-reckoning algorithms and specifying their thresholds. Although dead-reckoning is dependent on the game player's direction, speeds and the terrain in general, we can provide a Synchronization Medium with a dead-reckoning Manager for all the games in a certain class of games thanks to the classification presented in section 6.4. The CriticalRegionManager class and the LocalLagManager class are responsible for managing critical regions and the local lag usage in the game respectively.

7.2 Reusability of the Medium

In this section, we argue that the architecture we proposed is re-usable without modifying the underlying medium architecture. In the previous section, we showed an example of a car racing game using the medium. In this section, we show the re-usability of our Medium by deploying two different multiplayer games, a Space War game and a multiplayer battle tank game, on the top of it without modifying the underlying Medium and middleware code.

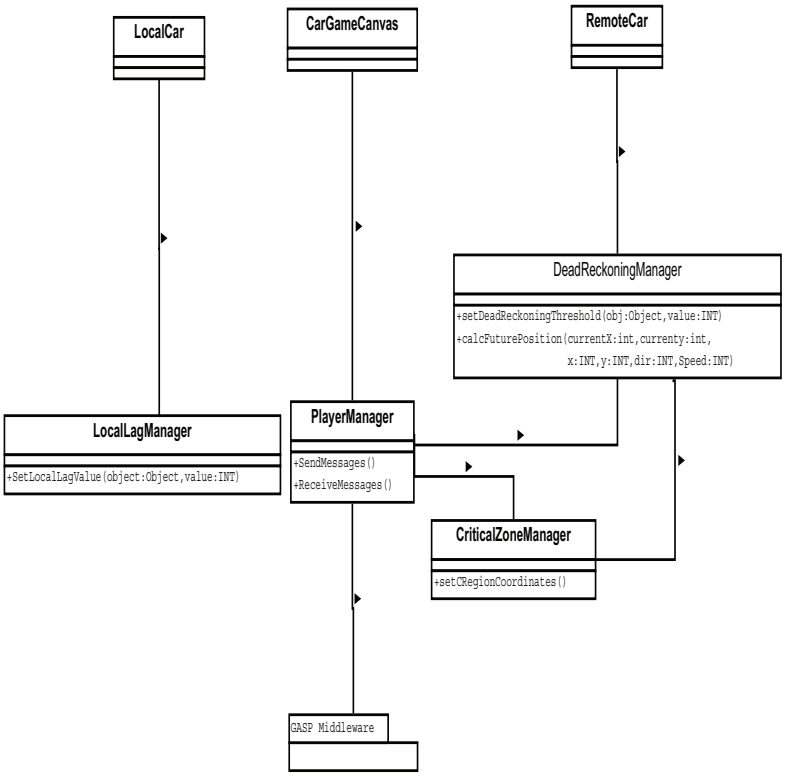


Figure 7.1.2: A Medium as used by a car racing game

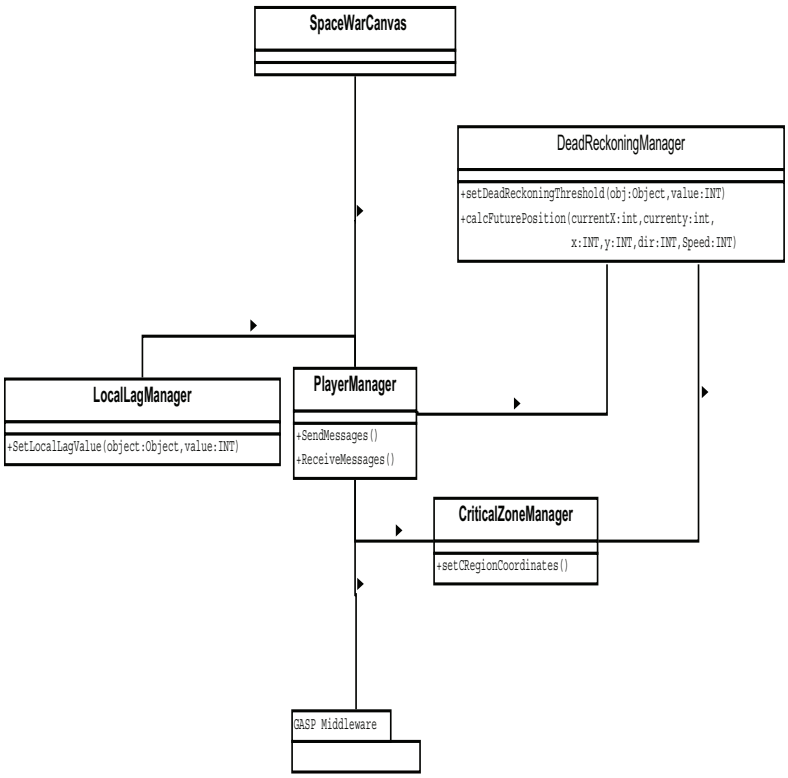


Figure 7.2.3: SpaceWar game using the medium

7.2.1 Case study 1 - SpaceWar Game

For the medium to be reused by the game developers, it must provide some general consistency services through interfaces. In the diagram of figure 7.2.3, we show how an existing medium can be re-used by a game developer. The game we re-used, called Space War, is a multiplayer game that can be downloaded from [1] (Powers, 2006).

In the figure, we show the game classes interacting with the medium using its interfaces. In the real implementation, this game has more than one class, but for the sake of simplicity we only show the main canvas class which interacts with the medium. As shown in chapter 6, the medium is represented by a set of Managers. The set of all the managers on all the clients constitutes the Synchronization Medium.

In figure 7.2.4, we show the spaceWar game without any medium. The original game had nine classes representing three different building blocks of the game: 1) the midlet 2) the game 3) the communication part. In the diagram, the whole game classes are represented by the SpaceCanvas class. The midlet class is responsible for initializing the game and doing the necessary administration tasks such as managing user accounts and passwords before starting the game. The communication part is

[1] http://developers.sun.com/mobility/midp/articles/gamepart2/

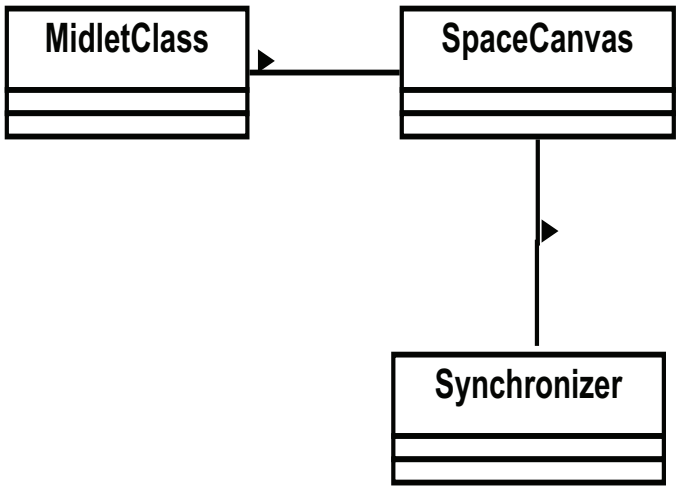


Figure 7.2.4: SpaceWar game without using any medium

represented by the synchronizer class responsible for the communication aspects as well as some part of the consistency maintenance.

When using the medium, the midlet part of the game is not required as it is the medium which is responsible for initializing and starting the game by calling the midlet's startApp() method (Wells, 2004). The communication part is the responsibility of the Synchronization Medium and the underlying middleware. Hence the developer is relieved from developing this part of the game when using the Synchronization Medium. The only part of the game that the developer must concentrate upon is the game itself. Therefore, the development of the game in figure 7.2.3 comes down to only concentrating on the game part itself, the remaining parts being handled by the medium.

In the original game code, the code for synchronization and consistency maintenance is embedded into different classes depending upon the developer's approach. If we have to change the code in the future for further enhancements or to use another consistency maintenance algorithm, we have to change a lot of synchronization related code weaved into the game logic thus costing up lots of resources. On the other hand, when using the Synchronization Medium, we write the code in a class which is not part of the game logic and which could be re-used in the future.

The space canvas represents the set of classes containing the necessary game logic. Whenever a message is received by the underlying infrastructure, it is accepted by the medium's PlayerManager class. This class calculates the time this message has taken from its delivery at the sender's terminal till its reception by this class. We suppose that the clocks of all the players are synchronized using some clocks method such as Network Time Protocol (NTP) (Mills, 1989), GPS-based (Sterzbach, 1997) or any other method (Diamond, 2006; Elson et al., 2002).
The medium buffers the message for a specific time corresponding to the local lag threshold value. This threshold value is provided by the developer of the game by implementing the *setLocalLagValue()* abstract method of the LocalLagManager class of the synchronization Medium. The message is then sent to the game canvas

class of the game. The game canvas calls the *calcFuturePosition()* method of the DeadReckoningManager class to apply the dead-reckoning algorithm for the calculation of the correction of the future position corresponding to this update message. This is necessary, because the message belongs to the past and now the sender of the message has moved to a new place depending upon its speed, direction and previous positions.

The CriticalRegionManager is responsible for defining the critical regions in the game. The critical regions are defined for each object (such as a circle around an object for example) and for the game terrain. The critical region around an object is mobile while the terrain critical regions have fixed coordinates. These regions are defined by the *SetCRegionCoordinates()* method of the Critical Region Manager.

In our first implementation of the Synchronization Medium with a car racing game, we have a single critical region and that is an area of the track just before the arrival line. When a car enters that region, the game canvas signals it to the player manager which then changes the values of the dead-reckoning and local lag thresholds accordingly to achieve a strong consistency. Through the interface provided by the medium for a specific genre of games, the game developer must provide the coordinates of the critical regions to the medium.

The local lag manager is responsible for setting the local lag value for each class of objects and changing it dynamically according to the objects pace and position in the game. In our implementation of the car racing game with just two cars, we kept this value equal to the maximum network delay between the two players. It was 500 milliseconds and 1000 milliseconds in two different experiments. This value is set by the developer of the game by implementing the *setLocalLagValue()* method of the LocalLag Manager. As this is an abstract method, the developer has to implement it.

7.2.2 Case study 2 - Tank Game

In order to confirm the reusability of our approach, we proposed a project to a student not familiar with synchronization or mediums. Jeanne-Marie Lavergne, a 3rd year engineering student at Telecom Bretagne realized the experiment. She transformed a single player battle tank game into a multiplayer game reusing the Synchronization Medium architecture (Laverge, 2010). A class diagram of the multiplayer version of the battle tank game using the synchronization medium is shown in figure 7.2.5.

In the game, each player owns a tank and tries to explode the tank of an opponent by targeting it with tank bullets. The main game logic classes of the battle tank game are Tank, RemoteTank, HeroTank, Bullet and Explosion. They use different managers of the synchronization medium to communicate and synchronize with remote players. The student eventually realized different versions of the multiplayer games with different synchronization strategies : no synchronization, a version using dead-reckoning for the tank and bullet positions, and local lag (for the bullet). Figure 7.2.6 shows the results of a version without using any synchronization algorithm and without any medium, comparing the positions of a tank locally and the ones displayed on the remote mobile terminal. We see that the divergence

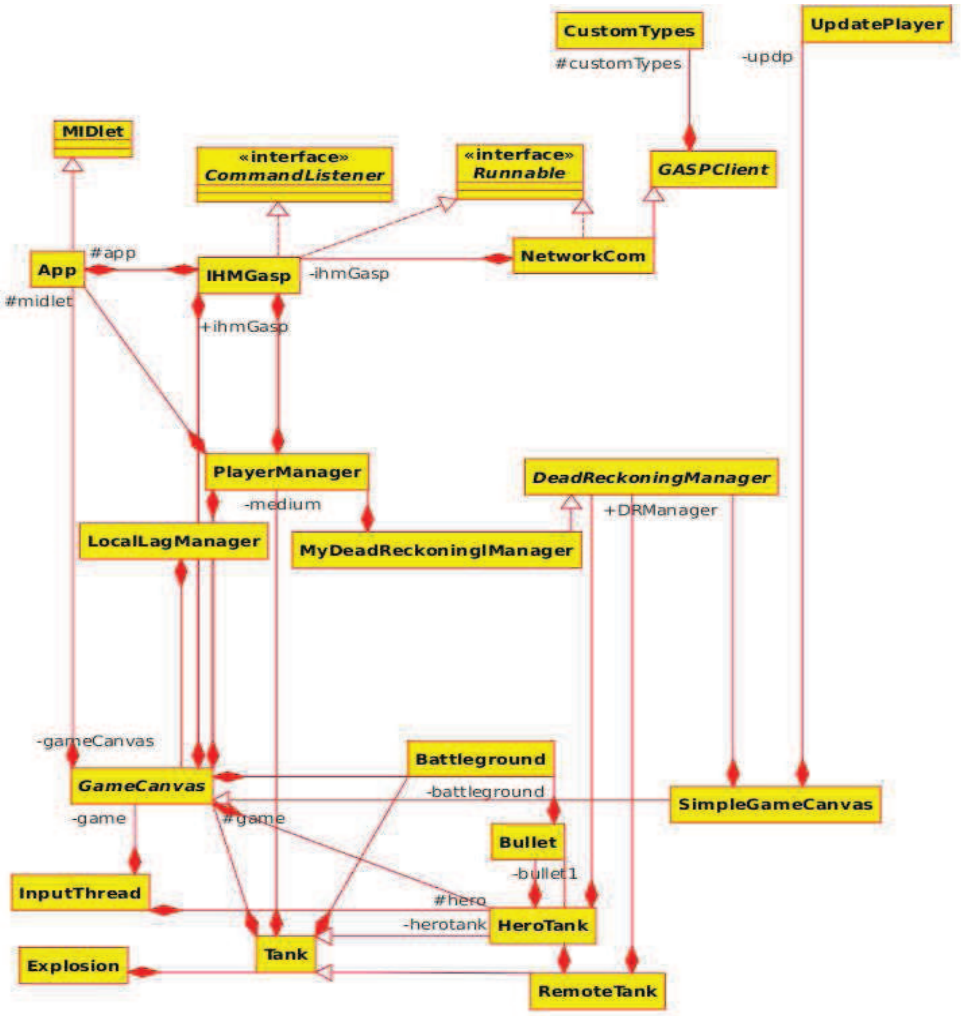


Figure 7.2.5: Class diagram of a multiplayer tank game using the medium

between both x and y positions for a local tank and its display on the remote mobile corresponds to a time period of 400ms i.e. the position is displayed on the remote player after a delay of 400 ms. Also, there is no smoothness in the motion of the tank (not shown in the figure). Figures 7.2.7 and 7.2.8 show the results when using dead-reckoning algorithms for calculating the movement of a tank and its bullet respectively. We see an improvement by decreasing the difference of the positions between the local player and its display on the remote player.

This project gives us confidence in the reusability and modularity of our approach. The deployment of three multiplayer games (car racing, space war and battle tank) on the top of the synchronization medium confirms its reusability.

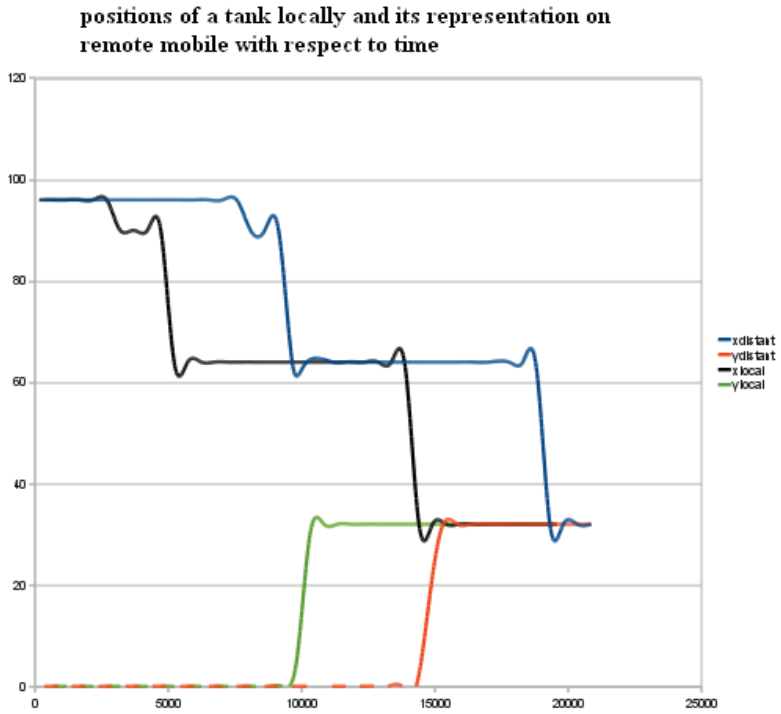


Figure 7.2.6: Position of a tank (local and distant) without any synchronization algorithm

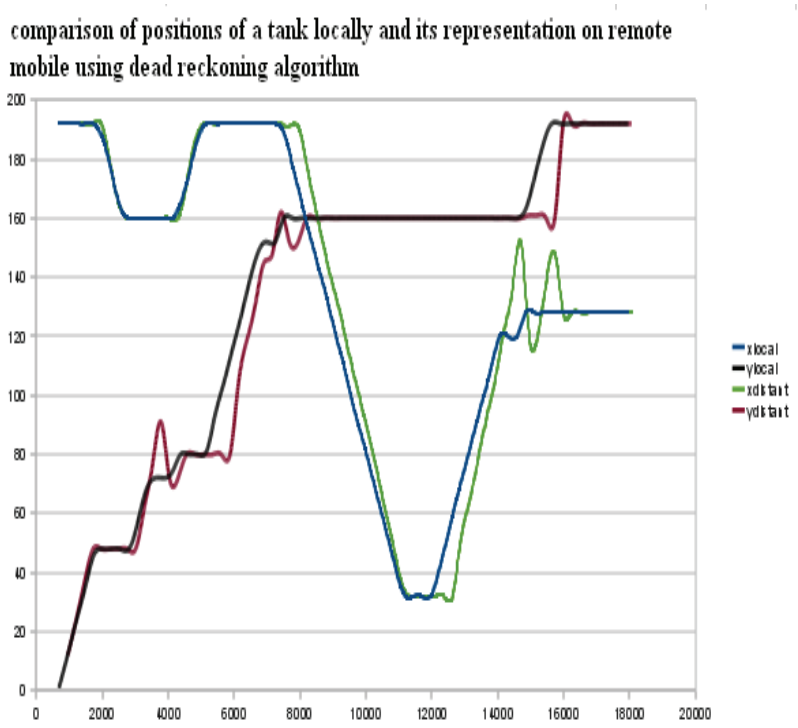


Figure 7.2.7: Position of a tank (local and distant) using dead-reckoning algorithm

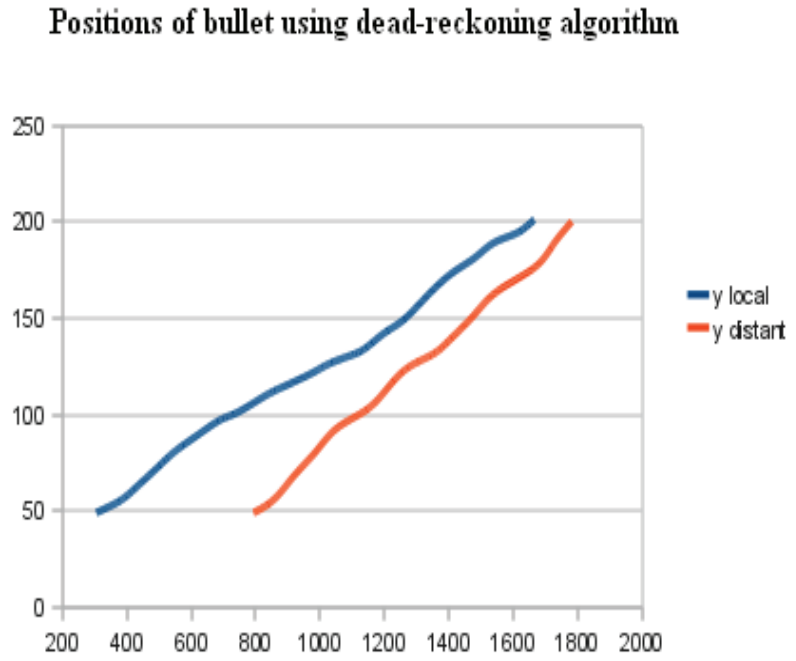


Figure 7.2.8: Position of a tank bullet (local and distant) using dead-reckoning algorithm

Table 7.3.1: Comparison of development efforts with and without a Synchronization Medium

Game	Existing Status	Game Logic Classes Using Medium	Reused Code (Medium Classes)
Car Racing	Developed	6	4
Space War	Existed with 9 classes	7	4
Battle Tank	Existed as Mono-Player game	8	4

7.3 Comparison of Development Efforts

In this section, we compare, quantitatively, the implementation efforts of game with and without medium. We argue that the architecture we proposed facilitates the development efforts in terms of the number of classes and lines of codes.
Table 7.3.1 shows a comparison of the development efforts when using the game on the top of the Synchronization Medium. As can be seen, the number of core game logic classes when using the Synchronization Medium is less than when game is deployed without a Synchronization Medium (in case of Space War game). The medium classes (related to synchronization and communication) are reused by the game. Hence the programmer is relieved from writing the lines of code responsible for consistency maintenance and this part of the code is re-used as a part implemented by the medium.

7.4 Performance Evaluation

In this section, we show that the insertion of a new consistency layer below the game logic classes does not introduce any untoward inconsistency, for example, due to an increase of the message processing time. Table 7.4.2 shows a comparison of the delay of the message processing with and without a Synchronization Medium. For an average network delay of 500 ms to 1000 ms, the average time for message processing is 0-5 ms when using the medium as compared to 0-2 milliseconds when the game is not using a Synchronization Medium. It means that, for an average delay of 500 milliseconds, the additional processing time is only one percent at maximum as compared to non-medium game where additional processing time is about 0.4 percent of the network delay. This is not a significant delay as compared to its advantages in terms of game architecture and development time.

In figure 7.4.9, we compare the positions of the bullets launched from a battle tank (as discussed in the previous section) locally and when displayed at the remote player using a certain local lag value (Laverge, 2010). The average difference, in

Table 7.4.2: Qualitative comparison between a stand-alone game and the one using Synchronization Medium

Network Delay	Stand-alone Game message processing time	Processing time for message when game is using Synchronization Medium
500ms	0-2ms	0-5ms
1000ms	0-2ms	0-5ms

terms of display time, between the bullet positions locally and when displayed on the remote site is 21,5ms which is well in the acceptable range of human perception. Hence, the insertion of a medium layer into the game logic code does not introduce performance degradation.

7.5 Concluding Remarks

In this chapter, we showed the reusability of the Synchronization Medium on different games of the same category. We also showed that the insertion of this consistency layer between the game application and the communication infrastructure does not affect the overall latency between two game applications. However, we believe that a comprehensive evaluation is still needed. For example, we could implement different Synchronization Mediums for different underlying infrastructures and used by the same game's client code. This will allow developers to re-use their game code without being worried about the network architecture, latency and bandwidth issues.

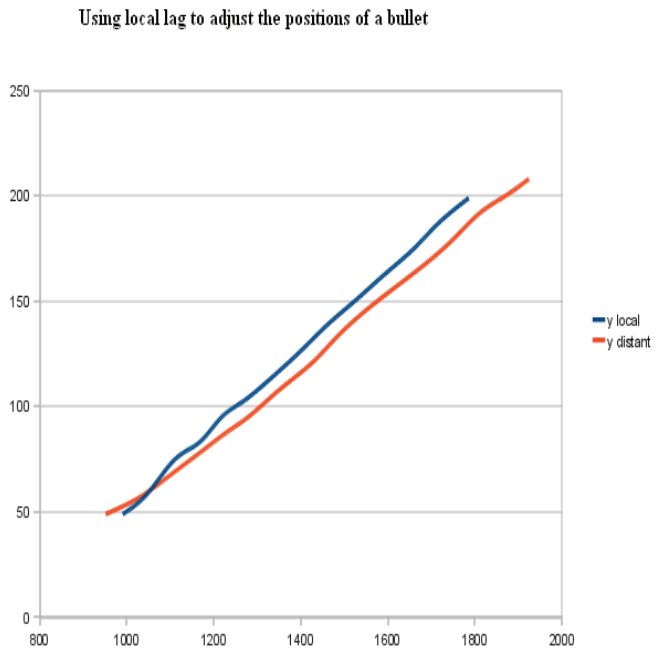


Figure 7.4.9: Position of a tank bullet at local and remote tank

Part III

Conclusion

Chapter 8

Conclusion

In this thesis, we have worked on three different aspects, i.e algorithmic, system architecture and software design, of multiplayer games on mobile phones in order to improve the state synchronization between remote players playing a game on high latency networks.

1. We proposed an adaptive approach to consistency maintenance in distributed virtual environments, especially multiplayer games, keeping in view the virtual world conditions as well as that of the underlying network. Based upon the conditions of the network and the game's virtual world, we proposed to have adaptable threshold values for dead reckoning and local lag keeping in view the latency in the network and the position of the game's objects at a particular time. We introduced the notion of critical regions to denote those areas in the game's world where strong consistency is needed. We also proposed an event synchronization mechanism that reduces the number of rollbacks in the face of high and varying latency thus maintaining the required responsiveness from the system. The key concept of our approach is to consider both networks conditions and game event semantics. This approach enables to vary the consistency degree between strong and relaxed depending upon the need of the situation. We evaluated our approach with multiplayer mobiles games executed in different conditions. The results show that our approach performs better in both low and high latency networks. The results also show that interactivity and fairness in the game can be obtained by dropping events that have no effect upon the final outcome of the game.
2. Based upon the mirrored server architecture, we presented a session-server architecture for wireless networks. In this architecture, different players (clients) playing a game on mobile phones are connected to the nearest session server. Session servers themselves are connected in a peer to peer manner. In case of disconnections, a mimicking mechanism, both on the client as well as on the session server sides, tries to mimic the game state until the connection is re-established. In case of a permanent or long duration disconnection, the player is removed gracefully from the virtual world. We believe that this approach helps game developers to have more control over the game so that

they can really benefit from the game they have deployed. This approach also tackles the issues of mobility and frequent short time disconnections. It remains to evaluate the proposed approach on a real testbed platform. We leave it as future work.

We also elaborated a protocol for maintaining consistency between game states on the clients and the server, in a client-server architecture, by dynamically switching to client side consistency when necessary. We evaluated our results with a multiplayer game and showed that better consistency can be achieved by applying mechanisms at clients' terminals in case of high latency and/or high game consistency requirement at that point in time. We showed that the client and server sides consistency maintenance approach results in better consistency than the server side consistency mechanism alone in case of high network latencies.

3. As a third contribution, we designed a software architecture for game development from the developer's point of view. We propose that the consistency maintenance as well as the communication issues are separate concerns from the game core logic and hence could be treated more appropriately. We introduce the idea of Synchronization Medium, a re-usable software component for consistency maintenance and communication. The advantage of such a component is that the development of the game code is simplified by letting the software developer concentrate on the logic of the game. Thanks to this separation of concerns, the evolution of the game code in the future is facilitated. Besides, the same game code can be re-used by picking a different synchronization medium for a different underlying platform. Also, the same synchronization medium can be re-used by the same category of games having similiar consistency requirements. We implemented our architecture on three different games and showed that the same synchronization medium be can reused by different multiplayer games. We also showed that the efforts in the development of the game logic code become easy and that the number of lines of code is reduced when using the synchronization medium. Moreover, the performance evaluation of the synchronization medium shows that this new layer does not introduce any significant performance degradation from the consistency maintenance point of view.

8.1 Short Term Future Work

In the near future, we would like to carry more experiments of our synchronization algorithms with different parameters and would like to apply our mechanism to a very large game with many more players.

Also, we have evaluated different synchronization algorithms supposing that the clocks of the remote clients were synchronized through some clock synchronization protocol. We plan to remove this assumption and implement some clock synchronization techniques to synchronize the remote mobile clocks in our experiments on different multiplayer games.

We would like to evaluate the session server approach by deploying a mobile multiplayer game. This evaluation should be from the points of view of efficiency in consistency maintenance, handling mobility and frequent disconnections.

For the synchronization medium, in the future, we would like to implement different synchronization mediums for different platforms, such as P2P, Mirrored-Server and Client-Server and then use these synchronization mediums for a same game application. This means that we would be able to re-use the same game code deployed on different platforms (with different bandwidths, latencies and control), without changing the game logic. Furthermore, we would like to evolve this synchronization medium into a middleware for multiplayer mobiles dealing with other issues apart from consistency and communications, such as managing players' sessions, defining roles like player or spectator etc.

It would also be interesting to have different variants of the synchronization medium deployed, and to select, at run time, the one whose algorithm is better suited at that particular time. This auto-adaptation, inspired from the (Phung-Khac et al., 2008) approach, would help in incorporating different synchronization schemes in the same game and in using the most suitable one, according to the context of the situation.

8.2 Long Term Future Plans

Multiplayer games represent a subset of the distributed virtual environments (DVE). Continuous applications such as distributed military simulations, distributed story-telling, distributed virtual concerts etc. have similar requirements from the consistency maintenance point of view. In the future, we would like to extend our ideas to these types of applications to see if we could conceive a common consistency model for these different types of distributed applications. Additionally, the concept of synchronization medium can be extended to these types of applications by identifying and developing common interfaces relating to consistency and communication for all these types of distributed applications.

Also, for the medium part, a multi-networks medium can be proposed mixing the IP, Bluetooth and 3G networks etc., so that the switching between different networks according to the needs of the situation can be carried out smoothly by the synchronization medium.

At the end, I will reproduce a quote from one of the greatest scientists, Isaac Newton, to show what I have learned during my PhD studies.

"After all these years, I do not know what I may appear to the world, but to myself, I seem to have been only a boy playing on the sea-shore and diverting myself in, now and then, finding a smoother pebble or a prettier shell than ordinary whilst the great ocean of truth lay all undiscovered before me." Isaac Newton

Bibliography

Akkawi, A., Schaller, S., Wellnitz, O., and Wolf, L. (2004a). A mobile gaming platform for the IMS. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 77–84, New York, NY, USA. ACM.

Akkawi, A., Schaller, S., Wellnitz, O., and Wolf, L. C. (2004b). Networked mobile gaming for 3G-networks. In Rauterberg, M., editor, *Entertainment Computing - ICEC 2004, Third International Conference, Eindhoven, The Netherlands, September 1-3, 2004, Proceedings*, volume 3166 of *Lecture Notes in Computer Science*, pages 457–467. Springer.

Bakhuizen, M. and Horn, U. (2005). Mobile broadcast/multicast in mobile networks. *Erricsson Review 2005*, (1).

Balakrishnan, M. and Sadasivan, M. (2007). Mobile interactive game interworking in ims. In *IP Multimedia Subsystem Architecture and Applications, 2007 International Conference on*, pages 1–5.

Billinghurst, M. and Kato, H. (2002). Collaborative augmented reality. *Communications of the ACM*, 45(7):64–70.

Bouillot, N. (2004). The auditory consistency in distributed music performance: a conductor based synchronization. *ISDM (Information Science for Decision Making)*, (13):129–137.

Bouillot, N. (2005). Fast event ordering and perceptive consistency in time sensitive distribued multiplayer games. In *7th International Conference on Computer Games (CGAMES'2005)*, pages 146–152.

Bouillot, N. (2006). La Cohérence Dans les Applications Multimédia Interactives : du concert réparti sur Internet aux jeux multi-joueurs en réseau . *Rapport de thèse*.

Bouillot, N. and Gressier-Soudan, E. (2004). Consistency models for distributed interactive multimedia applications. *Operating Systems Review*, 38(4):20–32.

Cacciaguerra, S. and D'Angelo, G. (2008). The playing session: enhanced playability for mobile gamers in massive metaverses. *Int. J. Comput. Games Technol.*, 2008:1–9.

Cariou, E., Beugnard, A., and Jézéquel, J.-M. (2002). An archictecture and a process for implementing distributed collaborations. In *The 6th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2002)*, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland.

Carless, S. (2006). Analyst: Online game market $ 13 billion by 2011. `http://www.gamasutra.com/php-bin/news_index.php?story=9610`.

Cavazza, M., Charles, F., and Mead, S. J. (2002). Emergent situations in interactive storytelling. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 1080–1085, New York, NY, USA. ACM.

Chandler, A. and Finney, J. (2005a). On the effects of loose causal consistency in mobile multiplayer games. In *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–11, New York, NY, USA. ACM.

Chandler, A. and Finney, J. (2005b). Rendezvous: an alternative approach to conflict resolution for real time multi-user applications. In *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pages 160–167.

Chandler, A. and Finney, J. (2005c). Rendezvous: supporting real-time collaborative mobile gaming in high latency environments. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 310–313, New York, NY, USA. ACM.

Chandler, A., McCaffery, D., and Finney, J. (2004). Rendezvous: The case for a highly optimistic real-time consistency mechanism. In *IEEE WACERTS'04 Proceedings*, Lisbonne, Portugal.

Cheok, A. D., Fong, S. W., Goh, K. H., Yang, X., Liu, W., and Farzbiz, F. (2003). Human pacman: a sensing-based mobile entertainment system with ubiquitous computing and tangible interaction. In *NetGames '03: Proceedings of the 2nd workshop on Network and system support for games*, pages 106–117, New York, NY, USA. ACM.

Cheriton, D. R. and Skeen, D. (1993). Understanding the limitations of causally and totally ordered communication. *SIGOPS Oper. Syst. Rev.*, 27(5):44–57.

Cronin, E., Filstrup, B., Kurc, A. R., and Jamin, S. (2002). An efficient synchronization mechanism for mirrored game architectures. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 67–73, New York, NY, USA. ACM Press.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421.

Diamond, P. (2006). Synchronization for future mobile networks. Website. `http://www.openbasestation.org/Newsletters/February2008/semtech.htm`.

Duncan, T. P. and Gračanin, D. (2003). Algorithms and analyses: pre-reckoning algorithm for distributed virtual environments. In *WSC '03: Proceedings of the 35th conference on Winter simulation*, pages 1086–1093. Winter Simulation Conference.

Elson, J., Girod, L., and Estrin, D. (2002). Fine-grained network time synchronization using reference broadcasts. *SIGOPS Oper. Syst. Rev.*, 36(SI):147–163.

Everquest. Everquest's website. `http://www.everquest.com`.

Falk, J., Ljungstrand, P., Bjork, S., and Hansson, R. (2001). Pirates: proximity-triggered interaction in a multi-player game. In *Proceedings of ACM CHI 2001 Conference on Human Factors in Computing Systems*, volume 2 of *Interactive posters: mobility*, pages 119–120.

Ferretti, S. (2005). Interactivity maintenance for event synchronization in massive multiplayer online games. Technical report, UBLCS-2005-05.

Ferretti, S. and Roccetti, M. (2004a). Event synchronization for interactive cyberdrama generation on the web: a distributed approach. In Feldman, S. I., Uretsky, M., Najork, M., and Wills, C. E., editors, *WWW (Alternate Track Papers & Posters)*, pages 226–227. ACM.

Ferretti, S. and Roccetti, M. (2004b). A novel obsolescence-based approach to event delivery synchronization in multiplayer games. *Int. J. Intell. Games & Simulation*, 3(1):7–19.

Ferretti, S. and Roccetti, M. (2005a). Fast delivery of game events with an optimistic synchronization mechanism in massive multiplayer online games. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 405–412, New York, NY, USA. ACM.

Ferretti, S. and Roccetti, M. (2005b). Fast delivery of game events with an optimistic synchronization mechanism in massive multiplayer online games. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 405–412, New York, NY, USA. ACM.

Ferretti, S., Roccetti, M., and Cacciaguerra, S. (2004). On distributing interactive storytelling: Issues of event synchronization and a solution. In Göbel, S., Spierling, U., Hoffmann, A., Iurgel, I., Schneider, O., Dechau, J., and Feix, A., editors, *TIDSE*, volume 3105 of *Lecture Notes in Computer Science*, pages 219–231. Springer.

Fletcher, R. D. S., Graham, T. C. N., and Wolfe, C. (2006). Plug-replaceable consistency maintenance for multiplayer games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 34, New York, NY, USA. ACM Press.

Frécon, E. and Stenius, M. (1998). DIVE: a scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering*, 5(3):91–100.

Fujimoto, R. (1999). Parallel and distributed simulation. In *Winter Simulation Conference*, pages 122–131.

Funkhouser, T. A. (1995). RING: A client-server system for multi-user virtual environments. In *1995 Symposium on Interactive 3D Graphics*, pages 85–92.

Gautier, L. and Diot, C. (1998). Design and evaluation of mimaze, a multi-player game on the internet. In *ICMCS '98: Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, page 233, Washington, DC, USA. IEEE Computer Society.

Griwodz, C. (2002). State replication for multiplayer games. In *NetGames '02: Proceedings of the 1st workshop on Network and system support for games*, pages 29–35, New York, NY, USA. ACM.

Hsu, C., Ling, J., Li, Q., and c. Jay Kuo, C. (2003). The design of multiplayer online video game systems. In *Multimedia Systems and Applications VI. Edited by Tescher, Andrew G.; Vasudev, Bhaskaran; Bove, V. Michael, Jr.; Divakaran, Ajay. Proceedings of the SPIE, Volume 5241*, pages 180–191.

IEEE, S. (1995). Application protocols. In *IEEE Standard for Distributed interactive Simulation*. IEEE Std 1278.1-1995.

Ikedo, T. and Ishibashi, Y. (2006). An adaptive scheme for consistency among players in networked racing games. In *MDM '06: Proceedings of the 7th International Conference on Mobile Data Management*, page 116, Washington, DC, USA. IEEE Computer Society.

Ishibashi, Y., Tasaka, S., and Tachibana, Y. (1999). A media synchronization scheme with causality control in network environments. In *Annual IEEE Conference on Local Computer Networks*, pages 232–241.

Ishibashi, Y., Tasaka, S., and Tachibana, Y. (2001). Adaptive causality and media synchronization control for networked multimedia applications. In *IEE International Conference on Communications*, pages 232–241.

Jefferson, D. R. (1985). Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425.

Kawahara, Y., Aoyama, T., and Morikawa, H. (2004). A peer-to-peer message exchange scheme for large-scale networked virtual environments. *Telecommunication Systems*, 25(3-4):353–370.

Khan, A. M., Arsov, I., Preda, M., Chabridon, S., and Beugnard, A. (2010). Adaptable client-server architecture for mobile multi-player games. In *DISIO'10: Proceedings of the Workshop on DIstributed SImulation & Online gaming*, Torremolinos, Malaga, Spain, 15 March.

Khan, A. M., Chabridon, S., and Beugnard, A. (2007). Synchronization medium: a consistency maintenance component for mobile multiplayer games. In Armitage, G. J., editor, *Proceedings of the 6th Workshop on Network and System Support for Games, NETGAMES 2007, Melbourne, Australia, September 19-20, 2007*, pages 99–104. ACM.

Khan, A. M., Chabridon, S., and Beugnard, A. (2008). A dynamic approach to consistency management for mobile multiplayer games. In *NOTERE '08: Proceedings of the 8th international conference on New technologies in distributed systems*, pages 1–6, New York, NY, USA. ACM.

Krishna Balan, R., Ebling, M., Castro, P., and Misra, A. (2005). Matrix: adaptive middleware for distributed multiplayer games. In *Middleware '05: Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, pages 390–400, New York, NY, USA. Springer-Verlag New York, Inc.

Kumagai, J. (2001). Fighting in the streets. *IEEE Spectrum*, 38(2):68–71.

Laverge, J.-M. (2010). Development of a multiplayer game on mobile phone: Research on improving the synchronization (in french). Technical report, Final Project for Engineering studies, TELECOM Bretagne, Brest France, 17 March.

Li, F. W., Li, L. W., and Lau, R. W. (2004). Supporting continuous consistency in multiplayer online games. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 388–391, New York, NY, USA. ACM.

Liang, D. and Boustead, P. (2006). Using local lag and timewarp to improve performance for real life multi-player online games. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 37, New York, NY, USA. ACM.

Lin, Y.-B. and Lazowska, E. D. (1991). A study of time warp rollback mechanisms. *ACM Trans. Model. Comput. Simul.*, 1(1):51–72.

Lineage. Lineage's website. `http://www.lineage-us.com`.

Mateas, M. and Sengers, P. (1999). Narrative intelligence. In *Fall Symposium on Narrative Intelligence*. American Associatin for Artificial Intelligence.

Mauve, M., Vogel, J., Hilt, V., and Effelsberg, W. (2004). Local-lag and Timewarp: Providing Consistency for Replicated Continuous Applications. *IEEE Transactions on Multimedia*, 6(1):47–57.

McCaffery, D. J. and Finney, J. (2004). The need for real time consistency management in p2p mobile gaming environments. In *ACE '04: Proceedings of the 2004 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 203–211, New York, NY, USA. ACM Press.

Mills, D. L. (1989). Internet time synchronization: The network time protocol. Network Working Group Request for Comments (RFC1129).

Moon, K. S., Muthukkumarasamy, V., and anh Nguyen, A. T. (2006). Reducing network latency on consistency maintenance algorithms in distributed network games. In *Proceedings of the IADIS International Conference: Applied Computing 2006*. IADIS Press.

Morse, K. L. (1996). Interest management in large-scale distributed simulations. Technical Report ICS-TR-96-27, University of California, Irvine, Department of Information and Computer Science.

Morse, K. L., Bic, L., and Dillencourt, M. (2000). Interest management in large-scale virtual environments. *Presence: Teleoper. Virtual Environ.*, 9(1):52–68.

Natkin, S. (2003). Computer games: A paradigm for new media and arts in the xxi century. In Mehdi, Q. H., Gough, N. E., and Natkine, S., editors, *GAME-ON*, pages 13–19. EUROSIS.

Okanda, P. and Blair, G. S. (2004). OpenPING: a reflective middleware for the construction of adaptive networked game applications. In chang Feng, W., editor, *Proceedings of the 3rd Workshop on Network and System Support for Games, NETGAMES 2004, Portland, Oregon, USA, August 30, 2004*, pages 111–115. ACM.

Palant, W., Griwodz, C., and Halvorsen, P. (2006). Evaluating dead reckoning variations with a multi-player game simulator. In *NOSSDAV '06: Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video*, pages 1–6, New York, NY, USA. ACM.

Palazzi, C. E. (2006). *Fast and Fair Event Delivery in Large Scale Online Games over Heterogeneous Networks*. PhD thesis, Department of Computer Science, University of Bologna.

Pantel, L. and Wolf, L. C. (2002). On the impact of delay on real-time multiplayer games. In *NOSSDAV '02: Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 23–29, New York, NY, USA. ACM Press.

Pellegrino, J. D. and Dovrolis, C. (2003a). Bandwidth requirement and state consistency in three multiplayer game architectures. In *NetGames '03: Proceedings of the 2nd workshop on Network and system support for games*, pages 52–59, New York, NY, USA. ACM.

Pellegrino, J. D. and Dovrolis, C. (2003b). Bandwidth requirement and state consistency in three multiplayer game architectures. In *NetGames '03: Proceedings of the 2nd workshop on Network and system support for games*, pages 52–59, New York, NY, USA. ACM Press.

Pellerin, R. (2010). *Contribution à l'ingénierie des jeux multijoueurs ubiquitaires*. PhD thesis, CEDRIC, CNAM, Paris, sept. 2009, updated in 2010.

Pellerin, R., Delpiano, F., Gressier-Soudan, E., and Simatic, M. (2005). Gasp: A middleware for multiplayer games in mobile phone networks (in french). In *UbiMob '05: Proceedings of the 2nd French-speaking conference on Mobility and uibquity computing*, pages 61–64, New York, NY, USA. ACM Press.

Phillips, W. G., Graham, T. C. N., and Wolfe, C. (2005). A calculus for the refinement and evolution of multi-user mobile applications. In Gilroy, S. W. and Harrison, M. D., editors, *DSV-IS*, volume 3941 of *Lecture Notes in Computer Science*, pages 137–148. Springer.

Phung-Khac, A., Beugnard, A., Gilliot, J.-M., and Segarra, M.-T. (2008). Model-driven development of component-based adaptive distributed applications. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 2186–2191, New York, NY, USA. ACM.

Plotkin, G. D. (1981). A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus.

Powers, M. (2006). Mobile multiplayer gaming, part 2: Applied theory. `http://developers.sun.com/mobility/midp/articles/gamepart2`.

Rhyne, T.-M. (2002a). Computer games and scientific visualization. *Commun. ACM*, 45(7):40–44.

Rhyne, T.-M. (2002b). Computer games and scientific visualization. *Commun. ACM*, 45(7):40–44.

Santos, N., Veiga, L., and Ferreira, P. (2007). Vector-field consistency for ad-hoc gaming. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 80–100, New York, NY, USA. Springer-Verlag New York, Inc.

Schloss, H., Botev, J., Esch, M., Höhfeld, A., Scholtes, I., and Sturm, P. (2008). Elastic consistency in decentralized distributed virtual environments. In *AXMEDIS '08: Proceedings of the 2008 International Conference on Automated solutions for Cross Media Content and Multi-channel Distribution*, pages 249–252, Washington, DC, USA. IEEE Computer Society.

Singhal, S. K. and Cheriton, D. R. (1994). Using a position history-based protocol for distributed object visualization. Technical Report CS-TR-94-1505, Stanford University, Stanford, CA, USA.

Smed, J., Kaukoranta, T., and Hakonen, H. (2002). A review on networking and multiplayer computer games. Technical Report 454, Turku Centre for Computer Science.

Steinman, J. S. (1990). Multi-Node Test Bed: A Distributed Emulation of Space Communication for the Strategic Defense system. In *Proceedings of the Twenty-First Annual Pittsburgh Conference on Modelling an Simulation*, volume 21, pages 1111–1115.

Steinman, J. S. (1991). SPEEDES: Synchronous parallel environment for emulation and discrete event simulation. In *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, pages 95–101.

Sterzbach, B. (1997). Gps-based clock synchronization in a mobile, distributed real-time system. *Real-Time Syst.*, 12(1):63–75.

Terraplay (2000). Terraplay systems. `http://www.terraplay.com`.

Tsang, M., Fitzmaurice, G., Kurtenbach, G., and Khan, A. (2003). Game-like navigation and responsiveness in non-game applications. *Communications of the ACM*, 46(7):56–61.

Unreal. Unreal technology. `http://www.epicgames.com`.

Vogel, J. and Mauve, M. (2001). Consistency control for distributed interactive media. In *MULTIMEDIA '01: Proceedings of the ninth ACM international conference on Multimedia*, pages 221–230, New York, NY, USA. ACM.

Wagner, C., Schill, M., and Männer, R. (2002). Intraocular surgery on a virtual eye. *Communications of the ACM*, 45(7):45–49.

Wells, M. J. (2004). *J2ME GAME PROGRAMMING*. Premier Press.

Xiang-bin, S., Fang, L., Ling, D., and Xing-hai, Z. (2007). An event correlation synchronization algorithm for mmog. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, volume 1, pages 746–751.

Yang, L. and Sutinrerk, P. (2007). Mirrored arbiter architecture: a network architecture for large scale multiplayer games. In *SCSC: Proceedings of the 2007 summer computer simulation conference*, pages 709–716, San Diego, CA, USA. Society for Computer Simulation International.

Zander, S., Leeder, I., and Armitage, G. (2005). Achieving fairness in multiplayer network games through automated latency balancing. In *ACE '05: Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 117–124, New York, NY, USA. ACM.

Zhang, X., Gracanin, D., and Duncan, T. P. (2004). Evaluation of a pre-reckoning algorithm for distributed virtual environments. In *ICPADS '04: Proceedings of the Parallel and Distributed Systems, Tenth International Conference*, page 445, Washington, DC, USA. IEEE Computer Society.

Zhou, S. and Shen, H. (2007). A consistency model for highly interactive multiplayer online games. In *ANSS '07: Proceedings of the 40th Annual Simulation Symposium*, pages 318–323, Washington, DC, USA. IEEE Computer Society.