UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

FÁBIO REIS CECIN

# Peer-to-peer and cheat-resistant support for massively multiplayer online games

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Prof. Dr. Cláudio Fernando Resin Geyer
Advisor

Porto Alegre, August 2009

**CIP – CATALOGING-IN-PUBLICATION**

Cecin, Fábio Reis

Peer-to-peer and cheat-resistant support for massively multiplayer online games / Fábio Reis Cecin. – Porto Alegre: PPGC da UFRGS, 2009.

214 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2009. Advisor: Cláudio Fernando Resin Geyer.

1. Massively multiplayer. 2. Online games. 3. Peer-to-peer. 4. Cheating. I. Geyer, Cláudio Fernando Resin. II. Título.

# ACKNOWLEDGEMENTS

I am very grateful for all individuals that have crossed their paths with mine and that have influenced, helped, or even enabled the conclusion of this thesis. I give a heightened thanks to: Carlos Eduardo Bezerra; Diego Gomes; Jorge Barbosa; Luciano Cavalheiro da Silva; Marcio Martins; Paulo Zaffari; Renato Hentschke; Rodrigo Luque and Solon Rabello.

A special thanks goes to Diego Midon, which has implemented the FreeMMG 2 simulator and generated the results presented in Chapter 5. I also give a special thanks to Felipe Vilanova and to Marcos Crippa for developing the collusion cheat simulator that was used to generate results for my thesis proposal.

Bringing this thesis to acceptable standards would not have been possible without the help of my advisor, Cláudio Geyer. Thank you for the unending patience and support.

I am forever indebted to the Informatics Institute of Federal University of Rio Grande do Sul (UFRGS) for providing me 13.5 years of high quality and tuition-free education. Thanks to all the teachers and staff that made it such a great place to be in.

Thanks to the Brazilian agencies CNPq, CAPES and FINEP for providing funding through scholarships and project grants. And to everyone that paid taxes so that these agencies could fund this research.

I give a special thanks to my four grandparents, Veny & Wally, and Miguel & Maria. Thank you for all the love and warmth.

I dedicate this thesis to my brother, Lucas Reis Cecin, and to my parents, Carlos Marcelo Cecin and Marcia Hilgert dos Reis. I am very grateful for having you as my family.

Nobody contributed more to this thesis than my wife, Mari Leia Wilhelm. Mari, de todos os lugares que o destino já me carregou, o lugar mais gostoso que eu já visitei foi o qualquer lugar, segurando a tua mão e tu segurando a minha. This thesis would have not come through without your support and your love.

*"Blizzard has announced that the little game that could has hit another big, round number: they are now claiming 10 million World of Warcraft players globally. Gamasutra is reporting that WoW has about 2 million subscribers in Europe, about 2.5 million in North America, and approximately 5.5 million in Asia.*

*They've also taken the step of qualifying what they mean when they say 'subscriber', something critics have used as a knock against their impressive subscriber numbers in the past due to the way that accounts are used in Korea and China. A subscriber, according to Blizzard, is someone who has 'paid a subscription fee or have an active prepaid card to play World of Warcraft, as well as those who have purchased the game and are within their free month of access'.*

*That number notably doesn't count promo subscriptions, expired accounts, canceled subscriptions, or unused prepaid cards. They are counting 'PC Bang' players by considering an account active if it has been used within the last 30 days. Subscribers that connect to the game under licensees like The9 are counted under the same guidelines.*

*It's a little staggering to consider the sheer number of people playing this game now. When I started playing these games 50,000 people in one world was a big deal; Lineage, EQ, and FFXI were big-time by being at or over 500,000 people. Now with titles like WoW, Gaia Online, MapleStory, and even Habbo Hotel demolishing the old concept of a Massive world, it's ... more than a little bit exciting.*

*I also think it's time that people just stop comparing WoW to other US-developed MMOs. People keep talking about a 'WoW Killer', but ...* ***10 MILLION people****, people. World of Warcraft isn't a game, it's a city. It makes as much sense to compare WoW to Dungeons and Dragons Online as it does to compare Scrabble with Chicago. That's just my opinion ... what do you think? Will there be a 'WoW Killer' someday?"*

— MICHAEL ZENKE, *World of Warcraft hits 10 million players (2008)*
*http://www.massively.com/2008/01/22/world-of-warcraft-hits-10-million-players/*

*"So I ask, in my writing, What is real? Because unceasingly we are bombarded with pseudo-realities manufactured by very sophisticated people using very sophisticated electronic mechanisms. I do not distrust their motives; I distrust their power. They have a lot of it. And it is an astonishing power: that of creating whole universes, universes of the mind. I ought to know. I do the same thing. It is my job to create universes, as the basis of one novel after another."*

— PHILIP K. DICK,
*How to Build a Universe That Doesn't Fall Apart Two Days Later (1978)*

# CONTENTS

# LIST OF ABBREVIATIONS AND ACRONYMS

| | |
|---|---|
| 2D | Two Dimensional |
| 3D | Three Dimensional |
| ADSL | Asymmetric Digital Subscriber Line |
| AI | Artificial Intelligence |
| ALM | Application Layer Multicast |
| AoI | Area of Interest |
| BSS | Baseline State Synchronization |
| CA | Certificate Authority |
| CPU | Central Processing Unit |
| CVE | Collaborative Virtual Environment |
| DDoS | Distributed Denial of Service |
| DHT | Distributed Hash Table |
| DoS | Denial of Service |
| DVE | Distributed Virtual Environment |
| FPS | First-Person Shooter |
| ID | Identifier (generic) |
| ISP | Internet Service Provider |
| IM | Interest Management |
| IP | Internet Protocol |
| LAN | Local Area Network |
| LP | Logical Process |
| MAC | Message Authentication Code |
| MMOFPS | Massively Multiplayer Online First-Person Shooter |
| MMOG | Massively Multiplayer Online Game |
| MMORPG | Massively Multiplayer Online Role-Playing Game |
| MMORTS | Massively Multiplayer Online Real-Time Strategy |

| | |
|---|---|
| MOG | Multiplayer Online Game |
| MTU | Maximum Transmission Unit |
| MUD | Multi User Dungeon |
| NLA | Network-Level Attack |
| NPC | Non-Player Character |
| P2P | Peer-to-Peer |
| PDU | Packet Data Unit |
| PC | Personal Computer |
| PMTU | Path Maximum Transmission Unit |
| QoS | Quality of Service |
| RAM | Random Access Memory |
| RC | Region Controller |
| RPG | Role-Playing Game |
| RTS | Real-Time Strategy |
| RTT | Round-Trip Time |
| TC | Trusted Computing |
| TCP | Transmission Control Protocol |
| TSS | Trailing State Synchronization |
| UDP | Unreliable Datagram Protocol |
| VE | Virtual Environment |
| WAN | Wide Area Network |

# LIST OF FIGURES

# LIST OF TABLES

# ABSTRACT

Typically, games classified as 'massively multiplayer online games' (MMOGs) are competitive, real-time, large-scale interactive simulations of graphical virtual worlds. Currently, most (if not all) commercial MMOGs are implemented as centralized services, where hundreds or even thousands of 'server' machines, maintained by the game service provider, are responsible for running almost all of the virtual world simulation. This incurs a significant equipment and communication cost for the game providers. Several works attempt to reduce the cost of hosting a MMOG by proposing more decentralized, peer-to-peer models for distributing the simulation among client (player-owned PCs with consumer-grade broadband) and server (provider-owned) machines, with some going as far as eliminating the need for provider-owned machines altogether. Decentralizing a MMOG, however, creates security issues, as the simulation is now delegated to untrusted client nodes which gain opportunities to cheat the game rules, as the rules are now executed by them. There are several types of cheats, but we show in this thesis that a case can be made for considering state cheating and denial-of-service attacks as the most significant threats for peer-to-peer MMOGs. In light of this, we propose FreeMMG 2, a new MMOG decentralization model based on the division of the virtual world into cells that are maintained individually by separate groups of volunteer peers that are running a non-interactive, daemon simulation process. Each peer of a cell contains a full replica of the cell state and synchronizes both conservatively and optimistically with every other peers (replicas) of the cell, while at the same time receiving game commands and disseminating game updates to actual player machines. Due to its cell replication and random peer selection, we show that FreeMMG 2 is resistant to state cheating. And, due to the use of one secret back-up peer for every primary replica peer of the cell, we show that denial-of-service attacks don't significantly increase the odds of either state cheating or cell state loss happening. Through network simulation we verify that FreeMMG 2 is scalable and bandwidth-efficient, showing that a replication-based approach to peer-to-peer MMOG support, considering peers with realistic Internet connectivity (no IP multicast and consumer-grade broadband), is a viable one.

**Keywords:** Massively multiplayer, online games, peer-to-peer, cheating.

**Suporte par-a-par e resistente à trapaça para jogos online maciçamente multijogador**

# RESUMO

Em geral, jogos classificados como 'jogos online maciçamente multijogador', ou *massively multiplayer online games* (MMOGs) são simulações interativas, competitivas, em tempo real e em larga escala, de mundos virtuais gráficos. Atualmente, a maioria (se não todos) os MMOGs lançados comercialmente são implementados como serviços centralizados, onde centenas ou até milhares de máquinas servidoras, mantidas pelo provedor do serviço do jogo, são responsáveis por executar quase toda a simulação do mundo virtual. Isto implica em gastos significativos em equipamentos e comunicação por parte dos provedores do jogo. Vários trabalhos tentam reduzir o custo de hospedar um MMOG propondo modelos de distribuição da simulação mais descentralizados (*peer-to-peer*), onde a simulação é movida parcialmente ou totalmente dos servidores (máquinas dos provedores do jogo) para os nós clientes, tipicamente PCs de jogadores conectados por banda larga residencial. Porém, a tentativa de descentralizar um MMOG cria problemas de segurança, na medida em que a simulação passa a ser delegada a nós clientes, que são nós intrinsecamente não-confiáveis que ganham a oportunidade de trapacear no jogo, burlando as regras, visto que as regras da simulação serão executadas por estes. Existem vários tipos de trapaças, mas nós mostramos nesta tese que é possível argumentar que a trapaça de estado (*state cheating*) e ataques de negação de serviço são as ameaças mais significativas para MMOGs *peer-to-peer*. Como consequência, nós propomos o FreeMMG 2, um novo modelo de descentralização de MMOGs baseado na divisão do mundo virtual em células que são mantidas individualmente por grupos separados de peers voluntários que executam um processo *daemon*, não-interativo de simulação. Cada *peer* de uma célula contém uma réplica completa do estado da célula e se sincroniza de forma tanto conservadora quanto otimista com cada outro peer (réplica) da célula, enquanto ao mesmo tempo recebe comandos de jogo e dissemina atualizações de jogo para as máquinas 'cliente' dos jogadores do jogo. Devido à replicação e à seleção aleatória de *peers* para as células, nós mostramos que o FreeMMG 2 é resistente a trapaça de estado. E, devido ao uso de um *peer back-up* secreto para cada *peer* réplica primária da célula, nós mostramos que ataques de negação de serviço contra os *peers* não irão aumentar de forma significativa a probabilidade de ocorrência de trapaça de estado ou de perda total do estado da célula atacada. Através de simulação de rede, nós mostramos que o FreeMMG 2 é escalável e que utiliza a largura de banda dos clientes de forma eficiente. Assim, mostramos que uma abordagem baseada em replicação de suporte a MMOGs, considerando clientes com conectividade à Internet realística (sem IP multicast e com banda larga doméstica), é viável.

**Palavras-chave:** jogos online, maciçamente multijgador, par-a-par, trapaça.

# 1 INTRODUCTION

Multiplayer online games (MOGs) have been enjoying an ever-growing popularity since Internet access became commercialized in the early 1990s. A study from 2006 estimated that the online games market would grow from US$ 3.4 billion in 2005 to over US$ 13 billion in 2011 (DFC INTELLIGENCE, 2006). There are several different types of MOGs, from two-player turn-based computer versions of traditional board games (e.g. Chess) to real-time 3D adventure and combat simulations with thousands of players.

The vast majority of MOGs are analogous to most real-world gaming or sports activities: a session of the game supports a small number of players, and they are self-contained activities in the sense that subsequent sessions are not affected, even if the same set of players is present. For instance, a MOG that simulates a realistic soccer match could support 22 players split in two teams of 11 each, last for one hour and a half, and end by displaying the final score for the match, which would not affect the starting score of any subsequent matches anywhere else. As a more concrete example, most online first-person shooter games (FPSs) support a few tens of players (usually up to 32, 64 or 128 players maximum) in matches that can last from 10 or 20 minutes to 1 or 2 hours, depending on the game and the game server configuration. When the game ends, the score is displayed, and a new match starts. Players can enter and leave at any time, and no results are recorded anywhere except for the occasional ranking which does not actually affect the rules or the outcome of later matches.

This thesis is about *massively multiplayer online games* (MMOGs), which are a special kind of MOG. These 'massive' MOGs are less like simulations of real-world *activities* and more like simulations of the real world *itself*. A MMOG is basically an online simulation of a virtual world which is filled with both opportunities for social interaction and for competitive activities found on traditional MOGs such as combat, strategy, economy and management, etc. For example, the most popular and successful MMOG, 'World of Warcraft', is a 'medieval fantasy' world where players upon connecting to the game find themselves controlling a 3D humanoid avatar (human, orc, elf, dwarf or other) inside some kind of city environment, surrounded by other player avatars. In World of Warcraft and other MMOGs, upon connecting the player generally isn't immediately thrown into a typical MOG activity such as fighting a dragon or other players, but rather finds itself in a larger-scale and freer-form environment that, notably, is more analogous to being in the real world. Also, most (if not all) MMOGs feature their own virtual currency and playing is often centered around managing virtual wealth, which also encompasses a great deal of being a human in the current real world.

The virtual worlds of MMOGs have proven to be quite popular and the number of MMOG players is ever-growing. A study estimates that the market for subscription-based MMOGs could reach US$ 2 billion by 2013 (SCREEN DIGEST, 2009). As of

April 2008, at least 16 million people around the world were playing MMOGs (WOODCOCK, 2008a), of which 10 million were for World of Warcraft (a market share of over 62%) (WOODCOCK, 2008a; BLIZZARD ENTERTAINMENT, 2008).

As of 2009, the majority of players of MMOGs have to pay a monthly fee to play. For example, World of Warcraft players are charged a fee of around US$ 15 per month. This is an economical necessity since, unlike traditional MOGs, MMOGs require that all game servers be controlled by a trusted game provider, which is usually the same enterprise involved with the game's development. This, in turn, generates high costs for the game provider, which must pay for all the server-side infrastructure: CPUs, power, bandwidth, staff, etc. Most notably, bandwidth for the MMOG server clusters is a significant recurring cost: one major MMOG operator has reported to spend one third of its subscription revenue to pay for the server-side bandwidth consumed by players (FENG, 2007).

Alternatives to pure client-server MMOGs have been proposed. Most of those are inspired by the *peer-to-peer* paradigm. Currently, **peer-to-peer support for MMOGs** is an active area of research. In that context, a significant amount of models have already been proposed which apply peer-to-peer to MMOG network support. There are several potential motivations (or advantages) for applying peer-to-peer to MMOG support, and below we list the main ones we have identified so far among current peer-to-peer MMOG proposals:

- To reduce the communication or computational load on the server-side infrastructure. By having some features of the MMOG be supported by client-server interaction, while off-loading other features to a peer-to-peer network, the cost of the server-side infrastructure can be reduced significantly;

- To eliminate the need for a server-side infrastructure. By making MMOGs run like existing peer-to-peer systems such as file-sharing systems, the central operator's role can be eliminated, resulting in a pure peer-to-peer system that doesn't have a central point of failure;

- To increase the quality of game play. Peer-to-peer games can force lower interaction latencies upon players since, in principle, peer-to-peer communication allows players to exchange game event messages directly (a minimum of one logical hop), whereas client-server systems often require players to synchronize with each other through the server (a minimum of two logical hops);

- To facilitate the development of different kinds of games. As with any architecture, centralized MMOGs are adequate for some types of games, but not so for others. Real-time strategy (RTS) games, for example, allow each player to control hundreds of units at the same time, which is not the ideal pattern for a centralized simulator. In a centralized architecture the bandwidth consumed by each client-server link increases linearly with the amount of independently-moving units. On the other hand, peer-to-peer protocols can support RTS games while making players exchange small command packets instead of large object update lists (BETTNER; TERRANO, 2001);

- To enhance server-side infrastructures. Most MMOG instances are not scalable past a few tens of thousands of players. The World of Warcraft service is actually an array of a hundred or so instances, each one currently supporting around 50,000 players at most (MCGRAW; HOGLUND, 2007), without support for interaction

between players located in different instances[1]. Some works show that computing resources allocated to isolated instances could provide better results if organized in more sophisticated ways such as a server-side peer-to-peer network, allowing for larger-scale instances, reduced player latency, etc.

The key motivation which has sparked this thesis is the first one on the above list. We want to significantly reduce communication and computing costs for MMOG server operators. We hope that by reducing the costs of MMOG service hosting significantly we can allow small game companies, independent game developers or researchers to deploy innovative or experimental MMOGs to large audiences.

However, developing a more decentralized alternative to MMOG support that is based on peer-to-peer computing presents many challenges. Two key reasons for MMOGs being centralized are the large scale and the long-lasting aspects of MMOGs highlighted earlier. Traditional MOG instances such as Quake (ID SOFTWARE, 2008) or Counter-Strike (WIKIPEDIA, 2008a) servers, being short-lived and small-scale with no stateful connection between any two individual instances, are easily hosted by volunteer servers not owned by the game developer or publisher. Serving a MMOG instance, however, requires a large pool of server machines with low-latency and high-bandwidth links to the Internet and to each other acting in concert. This is much more difficult to assemble from heterogeneous volunteer hosts that are scattered all around the Internet and that can go offline at any time.

Additionally, and perhaps even more importantly, there is the issue of trust. Misplaced trust in a distributed system leads to cheating. In a distributed server scenario, each server holds part of the state of the virtual world. For instance, Server A holds the state for players Alice, Adam, and others, Server B holds the state for player Bob, Bruce, and others, and so forth. In this simplified game state partitioning scenario, even if there are hundreds of honest and well-provisioned volunteer servers, it only takes one dishonest server to bring down the game's economy. For instance, Server A may grant an arbitrarily high amount of wealth to Alice for no reason, and the other servers (and players) won't know any better, thus instantly and trivially ruining the relationship between effort and reward in the economy and turning the game pointless. This game-obliterating cheat will be called **state cheat** through the remaining of this text, and it will be the main cheat that this thesis will try to address.

To avoid state cheating and to allow the service to be provided reliably to millions of players willing to pay for the privilege, commercial MMOG providers have resorted to centralizing all of the virtual world's critical functions, which means almost everything except (typically) client-side graphics rendering and, to some degree, character position updates. By doing this, state cheating is simply not possible and the game developers can instead focus on the *other* security problems that plague online games (WEBB; SOH, 2007; YAN; RANDELL, 2005) such as speed and position cheats, wall-hacks and other state exposure cheats, server-side software flaw exploitations such as item duping, auto-aiming cheats, reflex enhancers, online fraud, etc.

In this thesis we propose FreeMMG 2, a new distribution model for MMOG services. FreeMMG 2 is a hybrid of client-server and peer-to-peer, where most of the communication and computing load is transferred from the servers to the clients which, in turn,

[1]Usually, players are bound to their instance upon avatar instantiation. However, some MMOGs may allow players to travel between instances. For example, Blizzard Entertainment started allowing character transfers between World of Warcraft instances in June 2006, initially charging US$ 25 per transfer operation (CALDWELL, 2006).

coordinate directly with each other in a peer-to-peer fashion. The design of FreeMMG 2 is driven by two key points:

- To allow FreeMMG 2 to support MMOGs in an hostile, competitive environment, we attempt to make it highly resistant to state cheating, even in the presence of network-level attacks;
- To guarantee that a FreeMMG 2 game can execute on top of a peer network of consumer-grade Internet hosts with realistic specifications, we focus on making the model bandwidth-efficient and compatible with current consumer broadband technology. Also, no IP multicast support is required: only UDP/IP unicast support.

The FreeMMG 2 model is based on what we have called a 'multiple arbiter' approach. In a multiple arbiter model, several hosts (clients) hold a copy of the same cell (which is a part of the virtual world), and all such hosts hold equal power when it is necessary to determine the current state of that cell. This makes it difficult for a minority of colluding cell replicas to silently pass arbitrary modifications to the cell's state. The multiple arbiter approach contrasts with what we have called the 'single arbiter' approach, where the state of a cell is replicated, but one of the replicas is the master replica which has absolute power in determining the state of the cell. That single arbiter of each cell is then able to perform state cheating without depending on the cooperation of other nodes.

From our point of view, the main drawback of cell replication approaches in general, and of multiple arbiter approaches in particular, is the amount of messaging required to keep all replicas synchronized. When a replica changes its own state, it will need to somehow send its modification to all other replicas. Considering that IP multicast cannot be relied upon, achieving the desired bandwidth efficiency becomes a challenge. That adds up with several other difficulties inherent to peer-to-peer approaches to MMOG support in general, such as other vulnerabilities that can be exploited for cheating, high interaction latencies and temporal inconsistencies (if there are too many hops in the peer-to-peer overlay), tolerating peer churn (unplanned host disconnections), etc. That is to say, attempting solution to most problems can inevitably lead to an increase in messaging. For example, to prevent cheating, some multiple arbiter models cause multiple state update packets to be sent to a player machine at the same time (WEBB; SOH; TRAHAN, 2008; KABUS; BUCHMANN, 2007; ENDO; KAWAHARA; TAKAHASHI, 2005), while in FreeMMG 2 each player machine needs to receive only one update packet at a time.

Nevertheless, our main hypothesis is that an hybrid (client-server and peer-to-peer), cell-based, replication-based, multiple arbiter approach can result in a MMOG support model that is both bandwidth-efficient and resistant to state cheating, the latter even in the presence of network-level attacks.

To verify that hypothesis, first we describe the FreeMMG 2 model in detail, showing that it is possible to develop a complete peer-to-peer MMOG solution that is, in principle, reasonably complete and coherent. Next, we quantify the bandwidth usage of FreeMMG 2 through simulations, assessing its scalability and showing that FreeMMG 2 can support peer-to-peer MMOGs where the peer machines feature current, consumer-grade connectivity such as ADSL broadband. Finally, we explain why the multiple arbiter approach of FreeMMG 2 is resistant to state cheating and to network-level attacks throughout the text.

Thus, the main contribution and the goal of this thesis is to verify that hypothesis by providing a model (FreeMMG 2) as an example. And finally, in Chapter 6 we verify that the FreeMMG 2 model is indeed a novel contribution by comparing it with related peer-to-peer MMOG models.

The remainder of the text is organized as follows:

- **Chapter 2 – Background**: This chapter provides background information required for understanding the later chapters. This includes distributed simulation, multiplayer online games support, cheating in online games and related work in peer-to-peer MMOG support;
- **Chapter 3 – Intra-cell replica synchronization**: This chapter proposes BSS, a new event synchronization algorithm for small-scale games. After this, we show how BSS is used as the intra-cell synchronization mechanism of FreeMMG 2, our cell-based peer-to-peer MMOG architecture;
- **Chapter 4 – FreeMMG 2 architecture**: This chapter describes FreeMMG 2, which a design for a peer-to-peer MMOG middleware. We show what are FreeMMG 2's main modules and how they work together to achieve our vision of effective middleware support for peer-to-peer MMOGs;
- **Chapter 5 – Validation**: This chapter presents FreeMMG 2 simulation results and discusses how those validate our claims;
- **Chapter 6 – Comparison**: This chapter performs a comparative analysis of FreeMMG 2 against a selection of similar peer-to-peer MMOG middleware designs. Through that comparison, we show that FreeMMG 2 is an original contribution;
- **Chapter 7 – Conclusion**: This chapter draws some conclusions, presents a summary of the contributions in this dissertation, and discusses some opportunities for future work.

# 2 BACKGROUND

The goal of this chapter is to cover both basic concepts for and the state-of-the-art in peer-to-peer MMOG research. Section 2.1 defines what exactly is a MMOG in the context of our research. In Section 2.2, we use the predecessors of MMOGs, the MUDs (multi-user dungeons) to expose the motivation behind using distributed simulation for MMOGs. Section 2.3 covers distributed simulation concepts. The focus of this section is in explaining the concepts of conservative and optimistic synchronization, which are a pre-requisite for understanding all of our contribution and especially the part of it that is presented in the immediately following chapter (Chapter 3).

Section 2.4 covers support for client-server gaming, showing what techniques go into a client-server action game protocol and how they help game clients achieve not only synchronization but a high-quality game play experience. With the distributed simulation background from Section 2.3 and the background on client-server games support from Section 2.4, the reader will be equipped to understand MMOG architectures since most of them are hybrids of techniques both from distributed simulation and from client-server gaming.

In preparation for the upcoming MMOG decentralization discussion, Sections 2.5 and 2.6 discuss security topics. Section 2.5 explains the problem of cheating in online games, both for massive and non-massive online games. We make our case on the importance of providing *prevention* (rather than *detection*) of *state cheats* in peer-to-peer MMOGs by contrasting other possible cheats against state cheats. Section 2.6 considers the threat of network-level attacks against the peer-to-peer overlays of decentralized MMOGs. We make our case on why considering network-level attacks is important for a peer-to-peer MMOG.

Finally, in Section 2.7 we review existing state-of-the-art solutions for peer-to-peer MMOGs. We make our case that a gap exists where, at least for some possible peer-to-peer MMOGs, there is not enough *protection offered against state cheating*, and also not enough *resilience against network-level attacks* against selected nodes in the peer-to-peer network such as a DDoS. We finish the chapter in Section 2.8 by cementing the motivation behind FreeMMG 2, what to expect and what to not expect from it, and where exactly it fits among the other peer-to-peer MMOG models.

## 2.1 What is a Massively Multiplayer Online Game (MMOG)

Before proceeding, we will need a definition of Massively Multiplayer Online Games (MMOGs) to distinguish them from other MOGs and thus avoid confusion. In this thesis we differentiate MMOGs essentially by their **large-scale** and **long duration** traits, in contrast to traditional MOGs which are **small-scale** and have a **short duration**.

Though these two characteristics are generally enough to explain what a MMOG is, there are other key characteristics that are usually present in games considered as being 'MMOGs'. In this dissertation they will also be considered essential parts of MMOGs, that is, if a game doesn't have any of them, it will not be considered a MMOG. These are listed below:

- MMOGs are *persistent-state games*. This is an allusion to the fact that MMOG state is persisted through a game instance's (long) lifetime[1];
- MMOGs are *real-time simulations*. This is used to differentiate games such as World of Warcraft, Lineage and EverQuest, which are usually cited as examples of MMOGs, from turn-based, web-based games that support large player counts and also last for a long time. The latter are also called MMOGs, but in the context of this thesis, we will use the term 'MMOG' exclusively to refer to real-time games, that is, games where the discrete turns are measured in tens of milliseconds and thus give the illusion of being continuous simulations to human players[2];
- MMOGs are *graphical simulations*. This is used to differentiate from Multi-User Dungeons (MUDs), which were the text-based precursors of MMOGs. As with the web-based 'massive' games, implementing a MUD is not as near a technical challenge as implementing a real-time graphical virtual world, and so there is no distribution problem to tackle: the 'many clients, one server' approach is enough;
- MMOGs have *virtual economies*. This is a natural consequence of putting massive amounts of players inside the same game instance for years, be it in a real-time graphical MMOG, be it in a web-based massive game or in a MUD. That's a natural trait of the humans behind the avatars: we tend to hoard, protect and trade stuff, be it real or virtual. The 'virtual economy' aspect of MMOGs is where the most weird and wonderful phenomenons occur, such as players spawning real lawsuits for loss of virtual property (EGAN, 2008) and real-money trading of virtual assets (PAPAGIANNIDIS; BOURLAKIS; LI, 2008; WHITE, 2008)[3].

It should be now clear that real-time, graphical MMOGs (from now on, those will be called just 'MMOGs') such as World of Warcraft and EverQuest are significantly different from traditional MOGs, web-based 'massive' games and MUDs. Next, we will touch on why *supporting* MMOGs (i.e., designing, programming and deploying a distributed system that is used as middleware by the game logic) is a noteworthy (and hard) feat.

[1] However, the short-term MOG instances could also be seen as having a 'persisted' state, even if that state is discarded several times a day by the server. The difference here is that MMOG instances hold on to their shared state for a longer time, usually several years, with no concept of a sequence of separate 'sessions', 'levels' or 'maps' which are usual in traditional MOGs. So, the duration of a game instance determines for how long the shared state should be maintained alive, and the 'persistent' label is more or less redundant for abstract discussion. The 'persistent-state' distinction may however be useful at the software engineering level as it implies the use of some sort of persistent storage technology such as databases. Persistent storage is mandatory for MMOGs but seldom used by MOGs.

[2] In other words, if there is a conter anywhere on the screen telling the player how many seconds or minutes he must wait until he can submit another command, then that is not the kind of MMOG we're talking about in this work.

[3] This is also why cheat-proofing is much more crucial for MMOGs than it is for MOGs: cheating in a MOG instance affects only a small group of people and their effects disappear once the MOG instance ends (minutes or hours later), while cheating in MMOGs affects thousands of people and their effects can persist for years, causing cheats to compound with each other, eventually destroying the economy aspect.

## 2.2 The origin of MMOGs: Multi-User Dungeons (MUDs)

In 1975, 'Adventure', the first widely used adventure game (also known as 'interactive fiction' (WIKIPEDIA, 2008b)) was written by Will Crowther on a DEC PDP-10 computer (WIKIPEDIA, 2008c). In adventure games, players are presented a textual representation of the virtual environment surrounding a fictional character (the game player's avatar) and a textual prompt. Upon entering textual commands, a player acts on the game's environment. The result of those textual actions (such as 'walk north', 'examine the current location', 'pick up object X', 'use object Y', etc) results in another textual description being presented to the player that presented the resulting environment. Like most games, an adventure game usually has goals such as accumulating points through specific interactions or moving the avatar to a final location within the game world.

Adventure games were first promoted to multiplayer games somewhere around 1978 with the development of the original 'MUD' (Multi-User Dungeon or Multi-User Dimension) program by Roy Trubshaw and Richard Bartle (WIKIPEDIA, 2008d). The term 'MUD' can refer to both the name of the original program and the term used to describe any text-based on-line multiplayer virtual world. Because of that, the original MUD program is currently referred to as 'MUD1' or 'Essex MUD' (WIKIPEDIA, 2008d). The actual pioneer MUD was not the Essex MUD however, but a game called 'Oubliette' which was written by Jim Schwaiger in 1977 (WIKIPEDIA, 2008d) and deployed in a mainframe-based educational network called PLATO (WIKIPEDIA, 2008e).

The text-based MUD is the ancestor of the graphical real-time MMOG. A MUD is basically an adventure game where players can chat with other players and also interact with their respective in-game avatars in addition to interacting with the shared virtual world environment. The experience of adventuring in a fictional world together or against other human players was so strong that player addiction ensued (CURTIS, 1992; SCHIANO; WHITE, 1998; SEMPSEY, 1997), something that is also observed in players of MMOGs (NG; WIEMER-HASTINGS, 2005). According to Wikipedia, 'The popularity of MUDs escalated in the USA during the 1980s, when (relatively speaking) cheap, at-home PCs with 300 to 2400 baud modems enabled role players to log into multi-line BBSes. In Europe, MUD development and use centered around academic networks around the same time.' (WIKIPEDIA, 2008f). Today, MUDs are still popular and they can be played through the Internet via textual terminal protocols such as TELNET.

Both MUDs and real-time MOGs need several computers and a communication network to allow the human players to synchronize their actions between each other. MUD clients and servers will only exchange messages whenever some user finishes reading the current environment's description, thinks of his next command, and enters it on a prompt. The interval between two interactions from the same player will typically be in the order of several seconds, and usually there will not be any fixed deadlines for exchanges at the network level. This communication pattern could be well supported even with the earliest modem technology.

For this reason, MUD instances are well supported using a simple client-server solution, with one server that actually runs the game simulation and many clients that are the dumbest of terminals. From a software engineering point of view, the typical MUD is a solo computer program running in one machine, not a distributed system. The fact that the keyboards and screens used to interact with it are miles away and that their commands have to travel through a telephone network or a packet-switched network makes no difference to the game developer. There is no issue of 'trusting' what each keyboard submits to the centralized game logic running at the server; each terminal is free to send anything it

wants and it is solely the server's job to validate each user's input.

However, real-time MOGs (such as MMOGs) tax the underlying network much more heavily than MUDs. There is a reason why real-time MOGs usually require players to be equipped with *broadband connections*[4]. That reason is that real-time MOGs provide networked simulations of dynamic 2D or 3D spaces that change constantly, usually in the order of tens of times per second. This requires intense and constant network messaging between players, and in the case of MMOGs the load can require multiple servers to handle the load of a single game instance.

Thus, the problem of *distributing an interactive simulator* arises. In the small-scale MOGs, this happens because the clients are not dumb terminals, but actually run part of the logic of the simulation, albeit a limited one. In the large-scale MMOGs, client participation is also true, and the problem is compounded by the limit on the load that can be thrown at a single unit of computing, which mandates the use of multiple server machines. Distributed and interactive simulation is therefore our next topic.

## 2.3 Simulation background for MMOGs

Many of the fundamental concepts used for the construction of the networking subsystems of MOGs were already known before the Internet was commercially available: it turns out that online games are a prime example of *distributed simulation systems*. More specifically, online games are *interactive* simulations composed of many processes running on different machines distributed across the Internet[5].

Owing not only to this relationship of ancestry between games and simulation, but also to the applicability of distributed simulation to MMOG decentralization, we cover some basic concepts from simulation in this section. We show an overview of time and events in simulation, distributing the simulation among Logical Processes (LPs), conservative versus optimistic synchronization techniques and networking architectures for DVEs (Distributed Virtual Environments). We only present here the absolute minimum necessary for enabling later discussion on peer-to-peer MMOG support. For more on the subject, the reader is directed to Richard Fujimoto's *Parallel and Distributed Simulation Systems* (FUJIMOTO, 2000).

### 2.3.1 Time and interactivity

"A simulation is a system that represents or emulates the behavior of another system over time. In a *computer simulation*, the system doing the emulating is a computer program" (FUJIMOTO, 2000). By this definition, all computer games are simulations of another system: the one imagined by the game designer, which exists only inside the designer's mind. *System* is the keyword that brings the definition to life: we're not discussing an imagined static image being rendered in a painting. By *system* it is meant an object that, when observed by a human mind, is unavoidably perceived as containing other objects, movement and change over time. If the simulation is *interactive*, the human observer can also interact with the simulation while it is executing.

A central concept in computer simulation is time. In a simulator program there is always a variable that counts time as perceived from the inside of the virtual system. For instance, in a sports match simulation there will probably be a time variable that counts

[4] Links to the Internet that move hundreds of kilobits per second or more.

[5] A distributed simulation system can also have LAN connections between the processes (machines). The more interesting systems combine LAN and WAN (Internet) connections between peers.

the seconds or milliseconds since the virtual match has started, and that variable can be read by e.g. the referee object which then decides whether to end the match or not. The passing time as counted by the simulator in program variables is called *simulation time*, whereas the time as counted by humans in the real world is called *wall-clock time*. If there is a correspondence between the two times, that is, if simulation time advancement tries to mimic the advancement of wall-clock time, then the simulation is called a *real-time simulation*. However, if the simulation time advancement process completely disregards wall-clock time, then the simulation is called an *as fast as possible simulation*.

An example of *as fast as possible* simulation is the one used to generate the results presented in Chapter 5. There we describe the implementation of a simplified version of what an actual FreeMMG 2 game network would look like. We used it to simulate a game network running for several days of real time. The goal was to see whether FreeMMG 2 would work smoothly or break under the simulated load of thousands of players during these virtual operation days. Of course, though inside the simulator 'days' have passed, each actual simulator run took only minutes of real time to complete. While the simulation is running there is no need or want for human interaction, so it can run *as fast as possible*: simulated time advances as fast as the CPU of the machine running the simulation can cope with the computation.

Real-time simulations, on the other hand, need simulation time to have some relationship with wall-clock time (that is, the *real* time). This is usually the result of humans needing to continuously interact with the simulation while it is running. Thus in this thesis 'interactive simulation' and 'real-time simulation' will be considered synonyms, and we will consider that all games are interactive, and thus, real-time simulation applications.

As an example, consider a realistic, single-player racing game where a user controls a car. Also consider that the game loop runs at 50Hz, that is, it renders 50 frames per second to the player's screen, enough to present the illusion that the car is moving. Finally, let's assume that computing and drawing a frame takes 1 millisecond. Since there is a human interacting with the simulator, it will be expected that each frame represents 20 milliseconds of 'virtual car' movement corresponding to what a human would expect from 20 milliseconds (1/50th of a second) of movement from real cars (remember that it is a 'realistic' game). That is, not only the graphics have to be drawn on scale but the passing of simulated time must correspond to human expectations of time advancement, that is, of wall-clock time.

So, if a frame only takes 1 millisecond of wall-clock time to compute, this means that the CPU will have to remain idle for the next 19 milliseconds of real time, when it will be allowed to compute the next frame. If this waiting isn't performed, then the user would be interacting with a simulated world of car racing where everything moves 20 times faster than in real life, give or take some depending on the particular computing complexity of each actual frame. In essence, an interactive simulation is one where there is enough CPU power to compute the worst-case steps, and one where there can be a lot of CPU 'waiting' between steps in order to allow more wall-clock time to pass and 'catch up' with simulation time[6].

[6]Real game software usually do not implement waiting between frames but rather use any and all extra CPU time to draw additional frames or to employ it in other tasks such as AI computations. When there is shortage of CPU power, games automatically reduce the frame rate or the graphical quality of each rendered frame (usually the former).

### 2.3.2 Events, event-driven and time-stepped simulation

Fujimoto (FUJIMOTO, 2000) classifies simulation[7] as having either *event-driven* or *time-stepped* execution. Though this classification also applies to as-fast-as-possible simulation, we are only interested in its relationship with interactive simulators (games).

Both *event-driven* and *time-stepped* execution depend on the concept of *event* which, at an abstract level, is simply a happening that can cause changes to the state of the simulator. An event usually has a *timestamp* associated with it, which corresponds to a point in time when it happened or is supposed to happen[8]. Events can be the result of human activity, such as pressing game-command keys, or internal events, such as a computer-controlled opponent in a combat game deciding to shoot at the player.

In time-stepped execution, the simulator advances simulation time in discrete intervals or *steps* and, when doing so, runs some code that can potentially change all of the state variables at once. The key feature of time-stepped execution is that all modifications to the state happen at the same time, that is, all events scheduled to the same step have the same timestamp. This usually results in the simulator collecting events and executing them in batch when it is time to perform the next step computation. Thus, the step frequency must be high enough that the effects of this discretization do not cause errors that would be significantly prejudicial to users.

The single-player racing game example from before could be implemented as a time-stepped simulation. For instance, during the 20ms interval between frames, the simulator simply collects any and all input from the player that indicates whether the car should steer to the left or to the right, accelerate or slow down, etc. Then, when it is time to compute the next graphical frame for display, the simulation time steps by 20ms and all pending player events are applied with that same timestamp. Supposing that, within a frame, the player first ordered the car to steer left and instants later ordered the car to steer right, the time-stepping simulator would ignore the time distance between the keypresses and treat them as simultaneous events, as if the player pulled the steering wheel to both directions simultaneously. Though this is a simplification and that the car should have steered a bit to the left, most humans wouldn't be able to feel an interaction latency under 20ms.

In an event-driven execution, the simulator also advances simulation time in a discrete fashion, but this time the simulation time delta is always dictated by an event timestamp. This makes more sense for as-fast-as-possible simulation, but it can be mapped to games with some tweaks. First, one could generate a 'step' event in fixed intervals and have it actually move all objects by the amount of time of the step interval. The gain would lie in player events being more precise: each player command would be assigned a timestamp not bound by the step ticks. Executing a 'player command' event could be as simple as placing it in a queue so that the next 'step' event performs more complicated calculations that consider the exact timestamp of each command inside the step. Note that we could also have reached the same result by extending the time-stepped example to include timestamps in events. Fujimoto (FUJIMOTO, 2000) has some more detail on how to

[7] Fujimoto also classifies simulation as either continuous or discrete (or discrete-event). Continuous simulators describe the state of the system as a function of time, while discrete simulators describe the state of the system as a collection of variables which are updated by arbitrary pieces of logic in discrete points in time, repeatedly. All games that we know of are discrete-event, so by *simulation* we always mean *discrete-event simulation* in this text.

[8] Remember that in interactive simulation, simulated time corresponds to wall-clock time. Thus event timestamps, though being usually expressed in simulation time, can also be mapped to wall-clock time.

make each model emulate the other.

In this text we assume that games can use a hybrid of both approaches. From now on we will assume and describe games as time-stepped executions. However, we consider that time-stepping functions can order events inside the same step using their distinct timestamps and that these sub-step timestamps can affect the outcome. In the racing game steering example, the car would have had steered slightly to the left due to the 'left arrow key down' event before the later 'right arrow key down' event would neutralize it.

The abstract concept of *event* presented here gets more complicated in a distributed simulation scenario, that is, in a MOG or MMOG. In these scenarios, an event may have multiple time values associated with it, such as the time it is generated, the time in which it is sent, the time in which it is received and the time in which it is executed by the receiver. However, one of these times is usually chosen as 'the timestamp' of the event. In a typical client-server MOG, fairness and responsivity can be maximized when an event is timestamped at the player (client) since that minimizes the delay between event issuance and its timestamping, and security and temporal consistency are maximized when it is timestamped at the server since that disallows players (clients) from issuing arbitrary (false) timestamps. Determining who has authority to timestamp, order and execute events is a critical design point in online games.

### 2.3.3 Distributed simulation: the basics

The CPU time required to complete a simulation step depends on the complexity of the calculations. For a game server, complexity usually means the amount of avatars and other dynamic objects in the game world that need calculations such as movement and collision checking, AI tasks such as strategy and pathfinding, etc. Research works that assess scalability of game servers and MMOG network support schemes do so primarily through counting the amount of avatars supported on a server.

Using a single server machine to perform all simulation is not feasible for a centralized MMOG service that has a significant player base (e.g., hundreds of thousands of concurrent players). Thus, some sort of strategy to distribute the computing load becomes necessary. The trivial way out is to simply run an independent instance of the MMOG simulation process on each available server machine. This confers scalability to the service as a whole, but the amount of players supported at each game instance (shard) will still have the same (low) bound which is given by the current processor technology and how fast a single sequential program can be made to run.

A more scalable (and complex) solution is to distribute the simulation among several cooperating processes. The idea is that each process spends a bit of its resources to synchronize with other cooperating processes (the overhead) and the larger part of it to support part of the objects in the simulation. The end result is, hopefully, an increased game scale at the resulting MMOG instance. Below, some basic distributed simulation concepts are presented.

Fujimoto (FUJIMOTO, 2000) uses the term *Logical Process* (LP) to denominate distributed processes that cooperate in a global simulation effort. Each LP holds part of the global simulation. In the case of games, each LP could hold a cross-section of a contiguous virtual world terrain and any objects currently residing on its section, thus allowing many machines to contribute their CPU power to the same instance. The immediate problem with this approach is how to make those LPs cooperate, since they are separate processes running asynchronously, each on its own machine.

For LPs to cooperate, one basic requirement is for them to have a synchronized view

of the passing of simulation time. One way to achieve this is to have LPs synchronize their physical clocks or (more conveniently) a time offset variable that is to be added to the physical clock value. Clock synchronization can be achieved through a bit of messaging conducted between the LPs, and the quality of the synchronization depends mainly on the quality of the network latency estimation between pairs of LPs. In this work we assume that all LPs can have synchronized clock with errors in the order of tens of milliseconds, which is a reasonable assumption for NTP (MILLS, 1992) running over the Internet and generally acceptable for online games. The adverse effects of clock synchronization imprecision in distributed simulation are similar to those of network latency, and the mechanisms to tolerate the latter can be the same ones used to also tolerate the former.

The other basic requirement for LP cooperation is to have them exchange events. Each LP generates events while running. An event may be relevant only to the local LP, and it can also be relevant to other remote LPs. If the event is potentially relevant to remote LPs, the local LP has to typically fill an application data structure with event information such as what happened and at what time, and serialize that into a byte buffer that can be sent as a 'message' to the remote LP through some sort of communication stack. The receiving LP then deserializes the message into an event data structure, which is then processed by the receiving LP.

What exactly is done in incoming event processing at an LP, and also other factors, depends on the choice of *synchronization algorithm* which is a vital component of a distributed simulation. The synchronization algorithms attempt to solve *synchronization problems*. The next section shows an example of synchronization problem in a simple distributed virtual environment (DVE).

### 2.3.4 Synchronization problem example

To illustrate this problem of synchronizing multiple views of a shared simulation state, we start with a straightforward scenario. Consider an online game with exactly two players each with their own machine, three objects in the virtual world (two avatars and a ball), and a network link between the two machines. In essence, the game is about two avatars whose only possible action is to either pick up a ball first or watch the adversary pick up the ball first. More formally, both players run each an instance of the same game software which enforces the following set of rules:

- The shared virtual world can only be in three possible states: no avatar holds the ball (state FREE_BALL), the black player holds the ball (state BLACK_BALL) and the white player holds the ball (state WHITE_BALL);
- Each player maintains a version of the virtual world, which is in one of the three possible states;
- The only two legal events in the system are 'black player picks up the ball', which can cause a transition from FREE_BALL to BLACK_BALL, and 'white player picks up the ball', which can cause a transition from the FREE_BALL state to the WHITE_BALL state;
- To each player is granted the ability to make their respective avatar pick up the ball locally, that is, the black player's version of the world can legally transition from FREE_BALL to BLACK_BALL, but it cannot generate an event that transitions from FREE_BALL to WHITE_BALL, since only the white avatar's controller can decide to have the white avatar pick up the ball;

- A non-cheating simulator will only pick up the ball if it is available, that is, if the white client is in state BLACK_BALL, it knows that the black player has picked up the ball prior to him, so he won't be able to transition to WHITE_BALL;

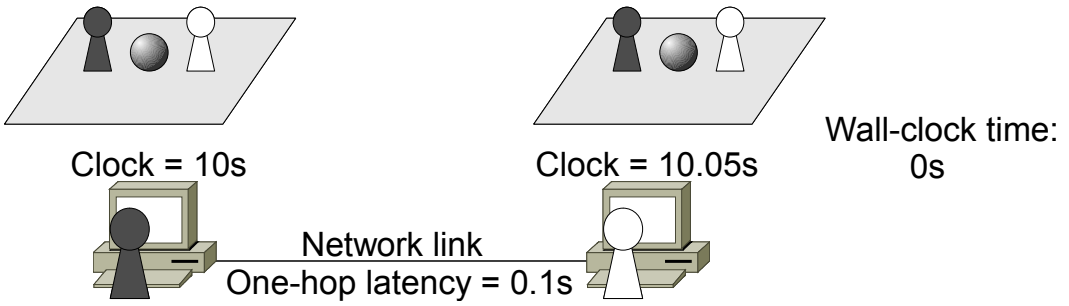


Figure 2.1: A snapshot of a working peer-to-peer game with two peers (players).

Figure 2.1 shows both the architecture and a possible state of the system at wall-clock time 'zero'. The system is composed of two real-time interactive LPs, each holding the full state of the simulation. The network delay between both players is considered a constant of 100ms one-hop in both directions with no message loss. Clock synchronization for simulation time is attempted by the system, with a slight error of 50ms in estimation: machine 1 assumes that simulation time is at 10s, while machine 2 assigns the value 10.05s to simulation time.

In this example, the simulation is not time-stepped for simplicity's sake. Events just take the timestamp of the sending (generating) LP which is derived directly from the synchronized clock. The state of each LP and thus the local view of each player only changes upon sending or receiving an event. Let's assume that the simulation times of 10s and 10.05s are a result of both players having started the game quasi-simultaneously about 10 seconds of wall-clock time ago, and since then the system simply remained dormant due to no player taking any action thus far. So, the initial game state was drawn at the simulation start, with both screens showing the FREE_BALL initial state, and the 'game loop' on each machine is waiting for a local input event (e.g. keyboard activity) or an incoming network message (incoming event from the other LP). So, if there is no activity from the local player controls or from incoming network messages, the simulation just keeps the simulation time going and the same local state and image on screen.

Figure 2.2 shows what happens when the simulator at the black player's machine receives a command (e.g. keyboard event) from the local player which requests that the ball be picked up by the black avatar. This happens one second of wall-clock time later. At this point in time, the white player still has not taken any action, as far as the black player is concerned. The black LP is able to verify that, at least as far as Black's version of the virtual world is of concern, the ball is available, and thus modify the local state from FREE_BALL to BLACK_BALL. We have the global vision of the system and we know that this is not a problem: white player has not taken any action and his copy still shows the ball as available.

After the black LP changes its state, it will have to let the white LP know that, so that the while player won't attempt to pick up the ball. To this end, the black LP builds a network message which contains the event (labeled as E1 in the figure) and the simulation time at which it happened – the timestamp of the event. Though E1 could be sent without

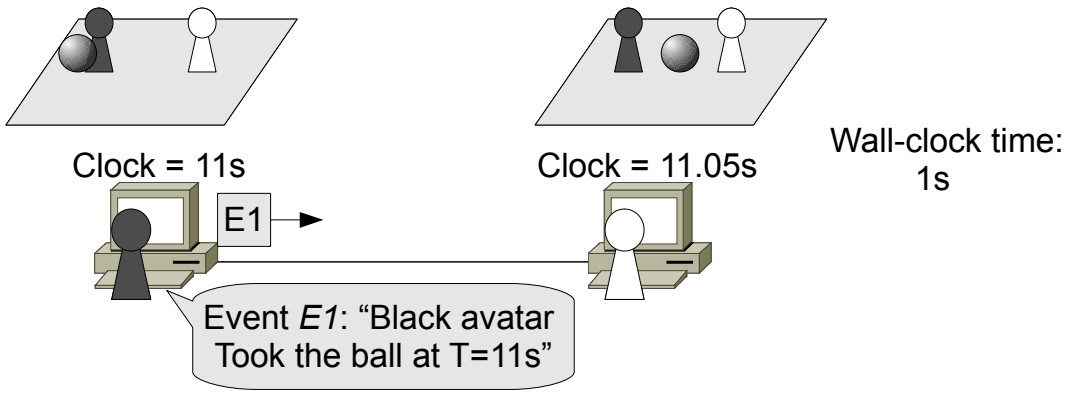


Figure 2.2: Example of temporal inconsistency.

a timestamp, it will become clear in a moment why maintaining synchronized clocks and sending events with time information is generally a good idea.

At the single point in wall-clock time depicted by Figure 2.2, a human-perceptible *temporal inconsistency* has happened. That is because the black player is now temporarily seeing a different version of the game world than the white player. The human controlling the black avatar is seeing the black avatar with the ball on his possession on the screen, while the human controlling the white avatar is seeing the ball available for pick-up between both avatars. This inconsistency can remain for the next 100 milliseconds, as the event-carrying network packet just emitted by the black LP has to travel our fixed-delay network to reach the white LP. Regardless of what the white LP will do with the event once it receives it, for the next 100ms it won't be able to perceive the event and thus the temporal inconsistency will remain.

The temporal inconsistency can be resolved after 100ms, when the white player receives E1 and also changes its view to state BLACK_BALL. After that, the game would be settled and no player would be able to perform any other action. However, there is a window of wall-clock time of 100ms (at least) where the white player can attempt to also pick up the ball, after black has already done the same. This is depicted in Figure 2.3.

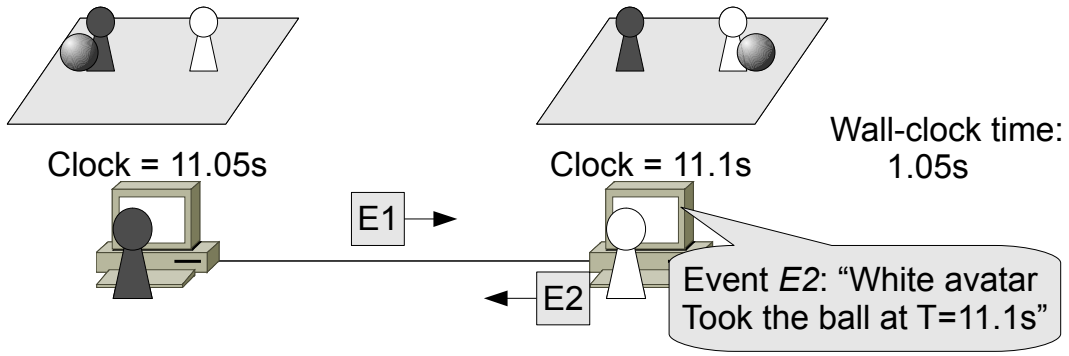


Figure 2.3: Example of a situation which will result in conflicting events.

Figure 2.3 shows White's player deciding to pick up the ball at wall-clock time 1.05s. This corresponds exactly to E1 being halfway through the network link. The figure illustrates a *conflict* situation, though one that is of impossible detection or prediction by

either LP at the current wall-clock time.

We consider a conflict to be a kind of temporal inconsistency which is not going to be solved simply by receiving and performing trivial processing of events in transit. In Figure 2.4, when E1 is received at White's computer, it cannot simply apply the event because the transition from state BLACK_BALL to WHITE_BALL is forbidden by the rules of the game: that is, an avatar cannot pry the ball away from the hands of another avatar.

This is what synchronization is all about: it is ensuring that processes distributed across a network can apply changes to a shared state while maintaining a consistent view of the shared state most of the time. The situations depicted above must be eventually resolved, or else the simulation is going to permanently 'fork' into two separate versions of the game world. Most games attempt to maintain the different views of the game world synchronized at least to the point that human participants cannot tell the difference between them[9] and temporal (temporary) inconsistencies that are proportional to network latency are tolerated by players as a natural consequence of networked gaming.

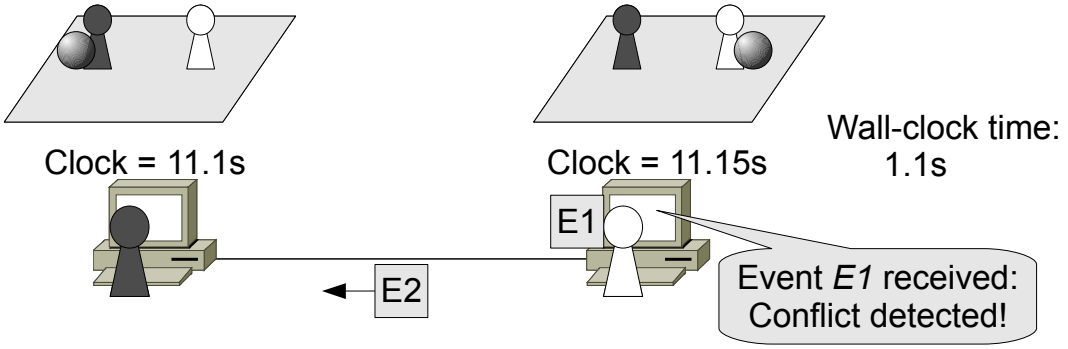


Figure 2.4: Example of a conflict being detected upon event receipt.

Most solutions to the conflict detected in Figure 2.4 begin by *trying to determine which player most likely acted first*, and then attempts to fix the simulation state at some LPs so that all reflect a consistent view. In our example scenario, the white LP could compare the timestamp of E1 (11s) to the timestamp of E2 (11.1s) and realize that the black player had acted first after all. This is a result of the clocks being synchronized and reflecting a 'good' approximation of the actual wall-clock time at which they were generated. The whole of DVE research in *fairness* concerns with increasing the odds that two events will be ordered by the distributed simulation software the same way they really occurred in the real world. That is, in the example, if the Black player has hit his keyboard controls first, then it is fair to to have his avatar pick up the ball.

However, there are some caveats to this approach. First, clock synchronization isn't perfect. Though the difference in the timestamps of E1 and E2 favors the black avatar by 100ms, the actual wall-clock time difference favoring E1 was of 50ms due to clock synchronization error. If the white player had acted earlier, an unfair situation might have

[9] A good example is a game where avatar positions are described in floating-point real numbers and the synchronization algorithm allows LPs to diverge below a certain precision. For instance, a stationary avatar at one LP could be at position 20.50 and at other LP the same avatar would be drawn at position 20.51, and this discrepancy would be neither detected nor corrected by the system since '0.01' in game distance units translates to e.g. less than one pixel. The system would then perform correction only when the avatar moves again, which has no time limit, but at least guaranteeing that separate discrepancies won't add up over time.

arisen where the clock synchronization error tilted the decision towards the white player. It is unlikely that a clock synchronization protocol would achieve a 50ms error with a link with perfectly constant latency (our example) but on the Internet this might possible due to the actual delay of each hop being different from the previous. In this thesis we have chosen to simply consider clock synchronization error as just another one of the undesirable side-effects of network latency, which in this case causes unfair decisions to be taken every once in a while.

Also, there is the problem of timestamp cheating. In a competitive game it is possible that an LP might lie about the timestamp of an event to gain an advantage. To cheat our proposed solution to conflict resolution thus far, the white LP could simply ignore E1, set its state to WHITE_BALL and emit an E2 with a timestamp of any value below the 11s timestamp value reported by E1. This would be interpreted at E1 as 'white acted first', which is incorrect, and the cheater player controlling the white avatar would win, causing the black LP to incorrectly and unfairly change to the WHITE_BALL state. There are several works that try to detect time cheats in the context of peer-to-peer games, as will be discussed in Section 2.5.5.

### 2.3.5 Optimistic synchronization

According to Fujimoto, 'Errors resulting from out-of-order event processing are referred to as *causality errors*, and the general problem of ensuring that events are processed in a time stamp order is referred to as the *synchronization problem*' (FUJIMOTO, 2000). In the example scenario of the previous section, the white LP executes first event E2, whose timestamp is '11.1s', at wall-clock time 1.05s. After that, at wall-clock time 1.1s, it receives event E1, whose timestamp is earlier: '11s'. When E1 is received, the white LP knows that E2 was executed out of time stamp order, because E1 has a smaller timestamp than E2. This, in turn, *may* result in a *causality error* in the simulation state if, for that particular simulation model, the order of execution of events E1 and E2 matters. In other words, if executing (E1, E2) leads to a different state than executing (E2, E1) then the latter order leads to a causality error. This is the case in the example, as the order of execution leads to two different and highly conflicting outcomes.

Our game example illustrates an **optimistic** approach to synchronization. At Figure 2.2, the black LP is optimistically assuming that the white player has not picked up the ball yet[10]. The *event-ordering* algorithm (or the 'protocol') decides that local events are executed immediately; by executing E1 immediately at the black LP, the distributed simulator is being optimistic that no *straggler* (late event with an earlier timestamp) will be received that conflicts with the optimistically processed E1 at the black LP.

This optimistic event-ordering doesn't solve the general synchronization problem by itself because, as we have seen, it guarantees not that the execution of events will always be in timestamp order. The solution, as suggested earlier, is to fix the simulation state after a conflict is detected so that execution of events in timestamp order is restored. Thus timestamp order is guaranteed in the long run, though it is temporarily violated between the receipt of straggler events and the end of the conflict resolution step.

Though we could come up with a finely-tuned mechanism that is specific to our model to fix the simulation once conflicts are detected (the common approach in MOG development), there are many general techniques that accomplish this. A basic kind of approach is *checkpointing* (or *snapshotting*). This consists of having LPs periodically save a copy of

[10] According to the approximative event ordering criteria used by the imperfect simulation apparatus, which may differ from the real-world order of events.

their state, and keep all events processed since the last checkpoint. Once a straggler event is received, the state is rolled back to the checkpoint state and all events are re-executed in the correct order, including the straggler. The various algorithms based on this vary on the frequency of checkpointing, the amount of checkpoints kept in history, the mechanism to detect when it is safe to discard a checkpoint state (that is, when a rollback to it is no longer possible), how to perform distributed checkpoints, etc. So, an optimistic approach may solve the synchronization problem if it is coupled with a recovery mechanism that is always able to restore time-stamp order from any late event whose out-of-order can result in a causality error.

### 2.3.6 Conservative synchronization: a stop-and-wait protocol example

In contrast to optimistic algorithms, there is another approach to event synchronization called **conservative synchronization** which, instead of fixing timestamp order violations, guarantees that events whose order of execution matters are never executed out-of-order at any LP. Thus, an LP that is conservative in executing events will usually defer the execution of received events until it can know that it is 'safe' to process that event, that is, that no events with smaller timestamps can be possibly received later. The main advantage is that recovery of conflicts is never an issue, and the main disadvantage is the added delay between the receipt of events and their execution, which severely limits the applicability of conservative approaches for real-time interactive simulation.

The topic of conservative synchronization algorithms can get complicated, especially for event-driven simulation (see (FUJIMOTO, 2000), Chapter 3). Fortunately, in this thesis we will only employ one of the simplest conservative protocols: a 'stop-and-wait' protocol with time-stepped execution and no *look-ahead* (more on look-ahead later). Below we modify our example game scenario from the previous sections to use the basic stop-and-wait protocol.

The initial state of the system will be the same as in the previous (optimistic) example, but with a slight modification. Instead of keeping simulation time as a relative measure of elapsed wall-clock time, we will model simulation time as a series of discrete time steps, and the simulation time variable $T$ will be set to the current step at each LP. The initial step is defined as being $T = 0$, and all LPs (black and white) must agree on an identical initial state for $T = 0$.

If this was supposed to be an 'as fast as possible' simulation, then each LP would attempt to advance to $T = 1$ immediately after it is started. However, in this example we are temporarily 'abusing' the concept of conservative simulation for a real-time game (more on this after the example). Thus, there is the need of a minimum interval of real time between each step so that the simulator can collect user events for execution in the next step. At each LP, the variable $R$ will count the amount of elapsed wall-clock time since the last step was computed at that LP. Whenever $R$ increases by 100ms (our arbitrary step size), the LP sends the events it collected on the current step, timestamps them with $T$ and sends them to all LPs, including itself.

Figure 2.5 shows the initial state of the system at an arbitrary wall-clock time 'zero'. Notice that we have modified the link to provide a constant 150ms of one-hop latency, which is to avoid confusion with the simulation step size of 100ms. $T = 0$ on both LPs, but $R = 0s$ only at the black LP: there is a slight clock synchronization error between the two LPs, and this has caused the black LP to start the simulation ahead of time.

Figure 2.6 shows the system 20ms later, when the white LP effectively starts. In the mean time, the black LP has advanced its $R$ by 20ms. Here, an important point must be

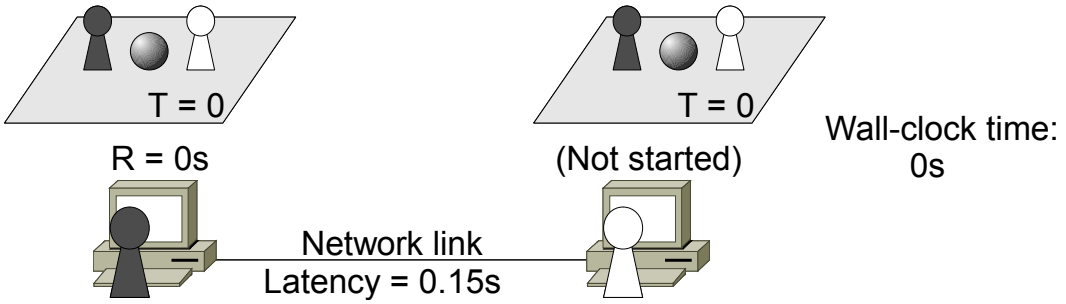


Figure 2.5: Stop-and-wait example (1 of 9).

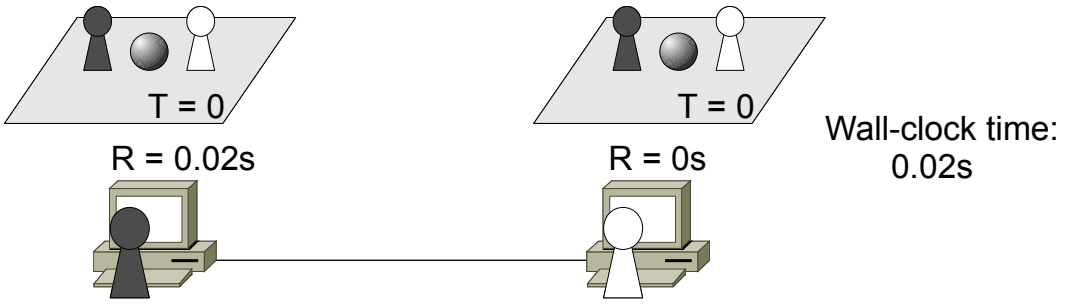


Figure 2.6: Stop-and-wait example (2 of 9).

made: the black avatar's human player has so far been presented to 20ms of on-screen animation. Though in our example nothing 'moves', this elapsed time can comprise physics simulation (e.g., non-interactive background animations or dead reckoning [11]). $R$, our measure of simulation time, has in fact advanced at wall-clock time pace so far, at the black LP.

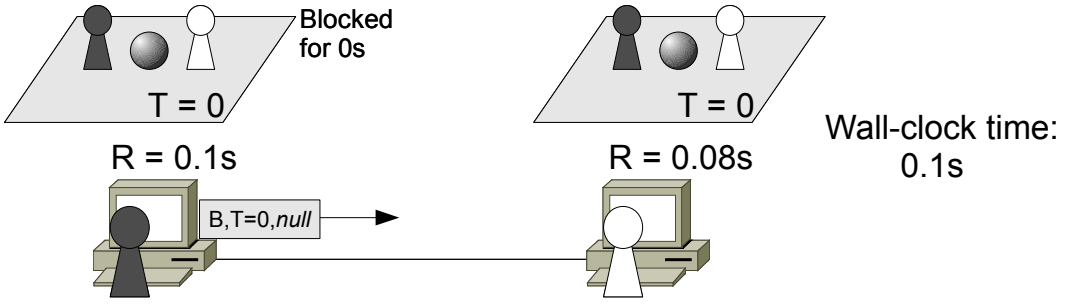


Figure 2.7: Stop-and-wait example (3 of 9).

Figure 2.7 shows $R = 0.1s$ at the black LP. The step interval is over at the black LP, and Black's human player has not issued any interactive command. So, per the stop-and-wait protocol, the black LP sends a message to all other LPs informing that, during step ($T = 0$), the black LP has not generated any events (Denoted by $null$).

[11] Dead reckoning will be addressed in Section 2.4. Basically, it means extrapolating the position of an object based on its last known position, velocity, acceleration and on the time elapsed since the object was at that state.

To keep simulation time ($R$) running at wall-clock pace at the black LP, it would be necessary to advance from $T = 0$ to $T = 1$ at the black LP, that is, to compute the state for the next step. This is necessary since $R = (0.1s, 0.2s)$ (the next range for the simulation time) only occurs during $T = 1$. However, the black LP cannot *safely* compute the next step since it doesn't know the set of events issued by the white LP for the current step. If the black LP assumed that the white LP didn't generate any events at $T = 0$, it could compute an erroneous $T = 1$ if the message for step $T = 0$ from the white LP contained the 'White picks up ball' event. The stop-and-wait protocol (and any other conservative simulation algorithm) strictly avoids processing events out of timestamp order. In our practical time-stepped example this means that events from all LPs with the same timestamp *must* be processed together. This forbids the black LP from processing its own $(T = 0, null)$ event without first receiving the event list for $T = 0$ from the white LP.

So, what happens at the black LP after the step is over is that the simulator *blocks*. That is, per the simple stop-and-wait protocol, $R$ at the black LP stops having correspondence with wall-clock time and stops being counted. $R$ advancement being blocked means that Black's human player is now seeing a *frozen* screen as far as object simulation is concerned. This will continue until the black LP has received enough information to safely compute $T = 1$. This means it will block until it receives a message from the white LP telling what is the complete set of events that it is committing for execution at $T = 0$.

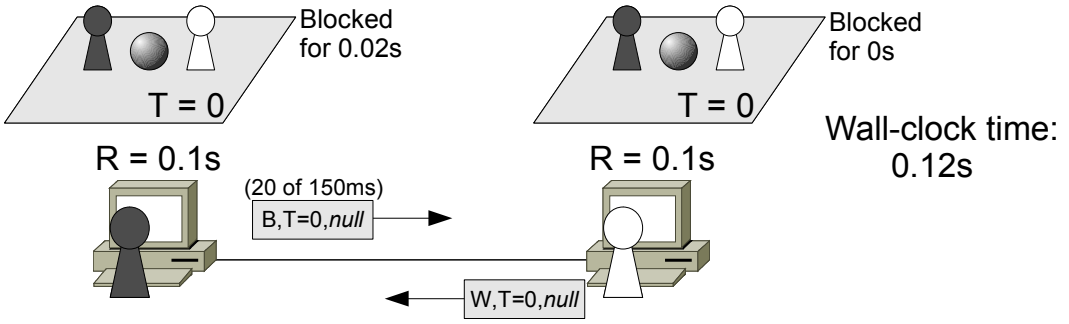


Figure 2.8: Stop-and-wait example (4 of 9).

Figure 2.8 shows the white LP finally reaching its threshold $R = 0.1s$ for the first step. It commits the same set of events for $T = 0$ as the black LP did, namely an empty set of events ($null$). In our example this means that neither human player issued any interactive commands. The white LP is now also blocked.

Figure 2.9 shows the white LP being unblocked. It may seem strange that the black LP is unblocked last, since it blocked first, but this is correct: by blocking first, the black LP emits its step message first and, assuming a perfect link with fixed delay and no loss, the white LP receives the remote event first and thus unblocks first. The white LP now knows all events scheduled to execute at $T = 0$ across the entire system: it knows it won't schedule any events for $T = 0$ and it knows the black LP has not scheduled and will not schedule any events for $T = 0$ since the received message $(B, T = 0, null)$ is a commitment from Black that it is not going to send any other events for $T = 0$ than the ones stated in that message.

Since White knows all events for $T = 0$, it can now safely compute $T = 1$. Since both Black and White didn't emit any useful events (both commited a $null$ set of events), the next step's state is only a function of the passing of simulation time over $T = 0$. In

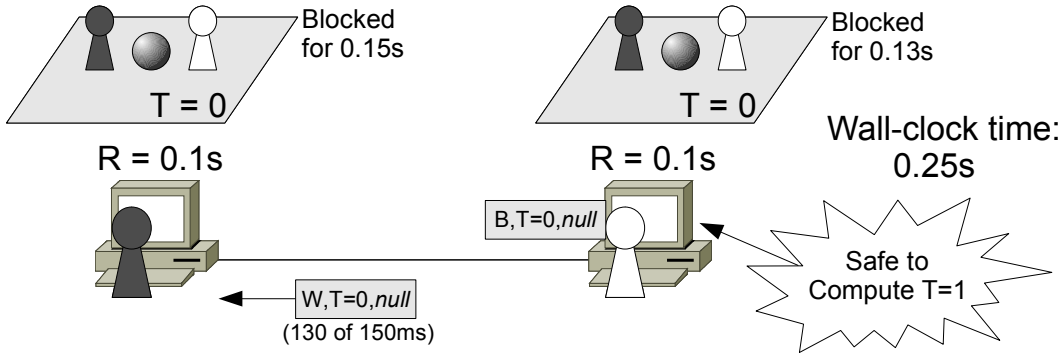


Figure 2.9: Stop-and-wait example (5 of 9).

our example game, the passing of time has no effect and thus the state on the next step is exactly the same as the previous one. In any case, the display on White is allowed to 'animate again' and simulation time once again flows at wall-clock time pace.

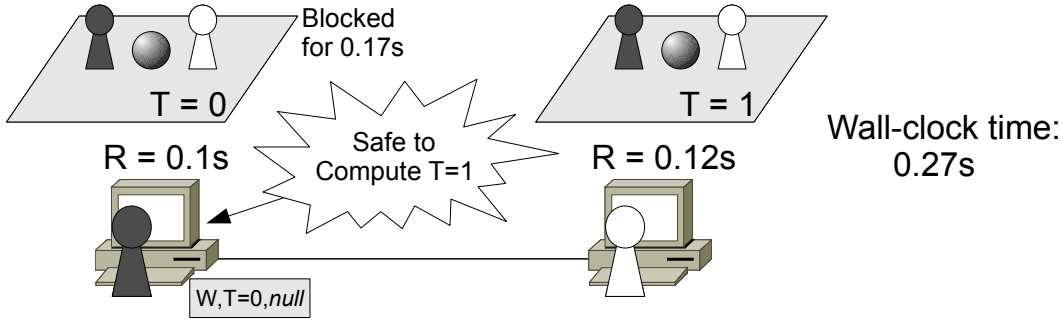


Figure 2.10: Stop-and-wait example (6 of 9).

Figure 2.10 shows the black LP unblocking. Per the conservative simulation contract, it will now be able to compute the exact same state for $T = 1$ that all other LPs in the system have, since it has the same previous state and the same list of events to apply. This requires event processing to be deterministic.

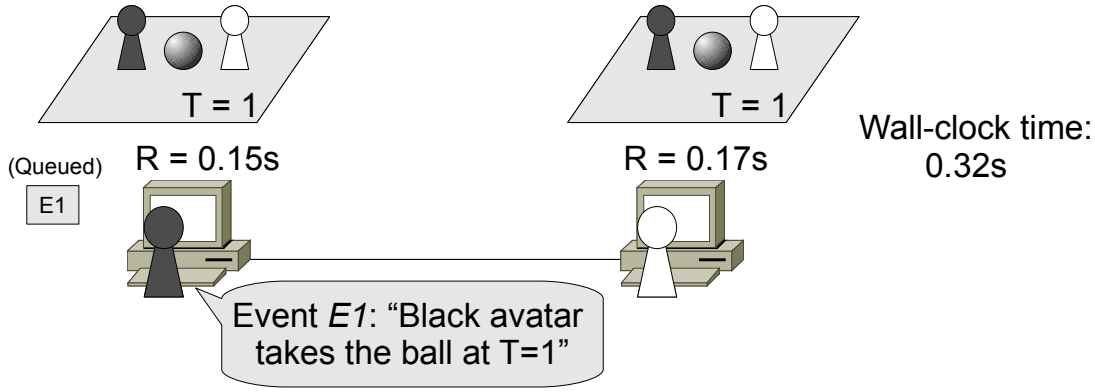


Figure 2.11: Stop-and-wait example (7 of 9).

Figure 2.11 introduces an action in the system. The black avatar's human player hits a key to pick up the ball. However, the black LP cannot execute the event immediately, because it does not yet know what White will do in this turn, which could be a conflicting action as we have seen earlier. Additionally, we have defined that events only apply at the end of each step's real-time duration: black only knows its *own* set of events for the current turn when the window for generating turn events ends locally. So, E1 is simply queued at Black, awaiting the end of turn $T = 1$ which will last for another 50ms of real time (since $R = 0.15s$). No ball is picked up anywhere yet.

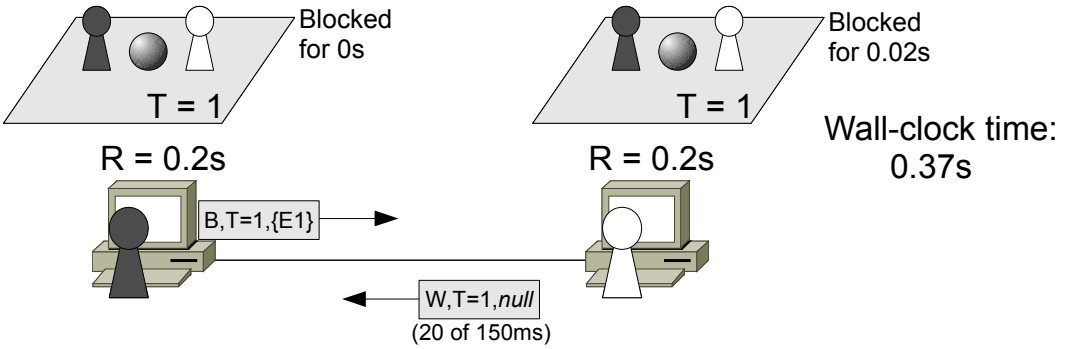


Figure 2.12: Stop-and-wait example (8 of 9).

Figure 2.12 shows the system 50ms later, when the step ends at the black LP. This causes the black LP to flush its event queue into the step message $(B, T = 1, E1)$, which informs that Black's player is trying to pick up the ball. In the mean time, White's step has already ended 20ms ago and it has already commited an empty set of events for $T = 1$. The step at Black has ended, but it cannot yet compute $T = 2$ since it doesn't know all events scheduled for $T = 1$ yet. This will happen 130ms later, when it receives White's message for step $T = 1$, currently in transit.

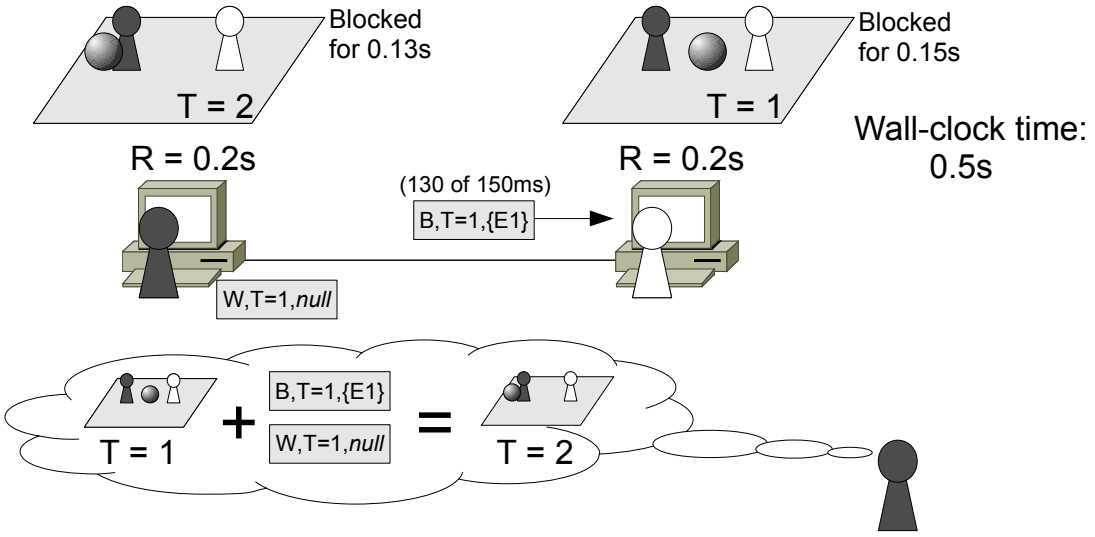


Figure 2.13: Stop-and-wait example (9 of 9).

Figure 2.13 shows Black receiving White's event commitment for $T = 1$. As before, by knowing all events for the current step, Black can now compute $T = 2$. For $T = 1$,

the black LP has received an event list from the black LP (itself) which contains a single event which is a request by the black avatar to pick up the ball. The black LP has also received an empty event list from the white LP for the current step. Thus, it is logical that the step update algorithm will grant the ball to the black avatar for $T = 3$. The white LP will eventually reach the same conclusion.

With this stop-and-wait protocol we thus achieve a synchronized view of the shared virtual world across distributed simulators without the need for state rollbacks. This simple protocol exposes the general thinking behind the conservative approach to synchronization, which is to avoid executing events out of timestamp order, for whatever modeling of simulation time and how events relate to each other in the simulation model (i.e., if total ordering is required or if some events can execute out of order). This has the benefit of simplicity and less computational overhead since a rollback mechanism has a cost.

However, the frequent 'blocking' of the game screen caused by stop-and-wait is not acceptable due to the poor resulting interactivity. As shown by the previous example, it is obvious that conservative event execution is not applicable to interactive (or 'human-in-the-loop') simulation as-is. However, this does not mean that conservative event execution is entirely non-applicable to games. Two possible applications of conservative simulation in real-time online games are the following:

- As a full synchronization solution for Real-Time Strategy (RTS) games: the successful commercial game Age of Empires (Ensemble Studios, 1999) uses a conservative event ordering scheme where player interactions generate commands that are scheduled for execution *two steps in the future*. Since each step in Age of Empires takes 200ms, this means that a game client only freezes when it experiences more than 400ms of one-hop latency from any other peer. This scheduling of events for the future is a form of *look-ahead* in simulation: an LP is able to 'look ahead' and know, for a certain window of time (or steps) in the future that these future steps are safe for execution (since the events for those should have already been received)[12]. This is only possible because the fixed 400ms delay between a command and its execution in an RTS game is acceptable. This is not acceptable for action games, so this is a special case of applicability of a conservative technique in an interactive application.

- As part of a more complex synchronization solution: in this thesis, we propose using the stop-and-wait protocol described above as a means of keeping a 'live' snapshot that an optimistic simulator can roll back to whenever necessary. The existence of conservative and optimistic simulator hybrids is not new (FUJIMOTO, 2000) but, as far as we know, for peer-to-peer MMOGs that is a new idea and one of our contributions. Our conservative and optimistic hybrid simulator is presented in Chapter 3.

More discussion on conservative simulation with look-ahead for RTS games can be found in our previous work on peer-to-peer massively multiplayer RTS games (CECIN et al., 2004). Also, Bettner and Terrano (BETTNER; TERRANO, 2001) describe in detail the Age of Empires distributed simulation engine.

[12] Actual look-ahead techniques for conservative non-real-time distributed simulation are a bit different (FUJIMOTO, 2000). It is possible that we have abused the original meaning of 'look-ahead' here but the basic idea of a safe window of time into the future for conservative event execution is the same.

It is important to note that by using the stop-and-wait protocol, all peers (LPs) are subject to the greatest communication latency between any pair of peers. That is, if two peers have 1000ms of latency between each other, a stop-and-wait will not run faster than one step per second. This also means that if a peer crashes, the simulation blocks forever. Thus, there is a need to define a *timeout* interval so that dropped peers are removed. Finally, removing peers from stop-and-wait is not trivial: it requires explicit negotiation between the peers (e.g., voting) in order to avoid some peers considering a live but lagged peer to be dropped in a step while others still see it as live, which can lead to different event sets being processed at different LPs for the same step and thus breaking determinism. An example removal protocol for stop-and-wait with look-ahead is described in the original FreeMMG dissertation (CECIN, 2005).

### 2.3.7 State partitioning versus replication

We have presented examples of a distributed simulation with two interactive LPs exchanging events that are timestamped according to a simulation time which is approximately the same at both LPs. However, each LP is holding the entire state of the simulation: both avatars and the ball, instead of partitioning it into two sets of objects and assigning a set to each LP. In principle, this would result in a scheme that would not scale as more LPs (player machines) and objects are added.

There are several ways to perform the distribution of objects among the LPs, and there is nothing stopping a programmer from using a different strategy for each type of object. For instance, each avatar can be assigned to the LP that has permission to control it (partitioning), while the ball is replicated at both LPs and permission to modify its state is given to both LPs.

However, in the example, even if an avatar is kept only at one LP, the other LP will be *interested* in knowing the state (position) of the remote avatar. This is important because it may affect the decision of a player to act or not. For example, assuming that we increment the game to allow avatar movement, the white avatar may want to pick up the ball only if the black avatar shows intent of doing the same by approaching the ball too. The natural way to accomplish this is to include an event in the system that tells the white avatar where the black avatar currently is (and vice-versa). While black is moving, its LP would emit a stream of such events to the white LP so that White's player would know about the current status of a relevant object (black avatar) which is no longer 'replicated' at the white LP. This way, even if the ball is still replicated at both LPs, partial state partitioning is achieved.

But is it really partitioned? In any DVE implementation scenario imaginable, White's LP is drawing the black avatar on-screen since it is nearby. This means that some data structure has to exists at the white LP to hold the black avatar's state between the receipt of the incoming *state updates*. If we count that data structure inside the simulation state of the white LP, then the black avatar isn't partitioned, but replicated. In this kind of distribution model, the object replicas don't have equal status: one will be primary (the authoritative copy) while others will just be slave or cached copies of the primary version (BHARAMBE; PANG; SESHAN, 2006). Some works (YU; VUONG, 2005; MÜLLER; GÖSSLING; GORLATCH, 2006) use the term 'replication' to mean this kind of master-slave object ghosting.

This differentiation also applies to as-fast-as-possible, non-interactive simulations. One could devise a simulation of two moving characters (black, white) on a grid, each controlled by one of two LPs in the simulation, similarly to our previous example. To

be able to move a character into a cell, it must be vacant. When one character moves, it must send an event to the other LP telling about its new location. How the receiving LP registers this move depends on how the simulation is *modeled* by the programmer. An alternative is to keep the current position of the remote character in a local variable, and another is to keep locally the status of each cell (either vacant or occupied) in addition to the local avatar's position.

Even in a third scenario where only the two avatars are present, if the LPs need to exchange events, then at least some *properties* on one of the simulation objects (avatar) are being affected by the other avatar. For instance, the black avatar could emit 'attack' events which increase a 'damage' numerical attribute of the white avatar, and both LPs could be oblivious to any other information about the avatar object that is remote to it. This scenario would more perfectly approach a 'partitioned state' perception. However, we could characterize the echoes of Black's events at the white avatar's 'damage' property as a kind of replication, since this too affects what white can and cannot do in the future. For example, if white is too damaged it can't move – perfectly analogous to being unable to move to an occupied spot in the previous grid example.

As it turns out, partitioning and replication are somewhat elastic concepts which need definition prior to usage. If you model a simulation using an object-oriented approach, you may draw the line on the object boundary, assuming that partitioning is achieved if events only modify object attributes instead of creating either replicas (*ghosts*) or copies which are all authoritative (that is, any LP with a copy can modify the object and emit events that mandate the change over other copies). Another approach is to consider *authority* as the boundary: subordinate 'ghost' objects are just side-effects of event processing and do not count as replicas. In this case, only the shared ball between the two players is considered replicated, since both players can change their copies.

The key to nailing down the concept of state partitioning in a distributed simulation is to remember that partitioning is supposed to achieve scalability. In a scenario where only one LP is responsible for computing the behavior of an object while other LPs only hold 'ghost' versions of the object, distribution of the *CPU load* for computing object behavior is achieved. In our game example, there is no CPU problem: the simulation is network-bounded. However, in a simulation where either an event is preceded by complex calculations (e.g. A.I. decision), or there is a large number of objects, or great speed-up of execution is desired (or all of these reasons), the simulation is likely CPU-bound. So, having object ownership distributed across several LPs means 'partitioning' is achieved, and it becomes somewhat irrelevant to determine which data structures are generated at event receivers as a side-effect of incoming event processing.

However, in the case of DVEs, especially for peer-to-peer games like in our example, the limiting factor is often the network. In this case, the key is also object authority, that is, how many LPs per object can emit events that change the other copies and how conflicts are to be resolved. Considering, in our example, that one LP owns an avatar and the other holds a 'ghost' of the avatar, we can say that the messaging complexity is $O(N)$, since the single avatar owner will be sending events to the N LPs that are *interested* in the avatar's state changes at any given time. However, the ball has at least $O(N^2)$ complexity, since all copies can send events to all other copies. The complexity is probably worse in the case of optimistic writes to the shared ball state, since conflicting writes will have to be undone, generating even more messaging.

When our proposed algorithms are described in Chapters 3 and 4, we will make extensive use of the term 'replication'. In this context, we always mean replication as $O(N^2)$

messaging. However, some works which we review in Section 2.7 mean replication as in the owner-ghost object paradigm and thus $O(N)$ messaging. We suspect that the term replication is used because a 'ghost' version of the object sometimes can take over the role of object owner if the LP with the master copy fails (or 'churns', in peer-to-peer parlance). However, in this text, such owner-ghost, $O(N)$ messaging scenarios are always considered a case of state partitioning, even if they can be seen as 'replicas' from the point of view of fault tolerance.

### 2.3.8 Distributed virtual environments (DVEs) and architectures

Most multiplayer online games can be classified as DVEs. As discussed previously, a DVE is a simulation of a shared 2D or 3D space which is interactive, allowing players to move avatars about it. Though in games research the term DVE is not frequently used, most MOGs and virtually all MMOGs are DVEs since they present the illusion of immersion in a shared space.

Another type of DVE is the CVE (collaborative virtual environment) which are DVEs aimed more at immersion and at facilitation of collaboration between users (e.g. hazard drills, combat simulations for the military, facilitator for virtual meetings, etc.). Many important results originally geared towards CVEs such as scalable distribution architectures (peer-to-peer CVEs), networking optimizations (area-of-interest filtering, publish-subscribe, dead reckoning) are reused in MOG works. Though CVEs usually focus on features that are still far from mainstream games such as capturing user's facial expressions and eye-gaze (MURRAY; ROBERTS, 2006), haptics (SHEN et al., 2004), CAVE interfaces (ASPIN; LE, 2007) and others, they also don't usually focus on cheat prevention which is an important issue to when considering support to competitive online games.

Figure 2.14 shows a possible classification of architectures for DVE support (FUJIMOTO, 2000). They are the following:

- *Centralized server architecture:* In this scenario, there is a *single server machine* which is running a sequential 'DVE server' process which is unique (a singleton) in the DVE system. The server maintains the authoritative version of the entire simulation's state, while a client maintain a subordinate version of part of the simulation's state. For example, in a simple game with several players, each able to move an avatar object, the server holds position information of all avatars, while a player may know about its avatar's position and maybe a couple other avatars which happen to be nearby and as such are relevant to him (e.g. to be drawn on the player's screen). Also, a player never decides the position of other player's avatars; instead, it is constantly receiving packets from the server telling where the nearby avatars are. And, even in protocols which permit the player to tell the server about the exact position of the player's avatar, the server is still the final authority and may reject impossible requests (e.g. move player avatar inside a wall). Protocols for this kind of architecture will be discussed in the next section. Fujimoto notes that computation-only entities (in games these would be any non-avatar dynamic objects such as monster, NPCs, items, environment objects such as doors, elevators, etc.) are best handled by the server;

- *Distributed server architecture:* This is similar to the centralized server architecture in respect to the authoritative version of the simulation being handled only by trusted server machines, in this case more than one and all in the same LAN. This allows the load of the simulation to be distributed (either partitioned or replicated, or

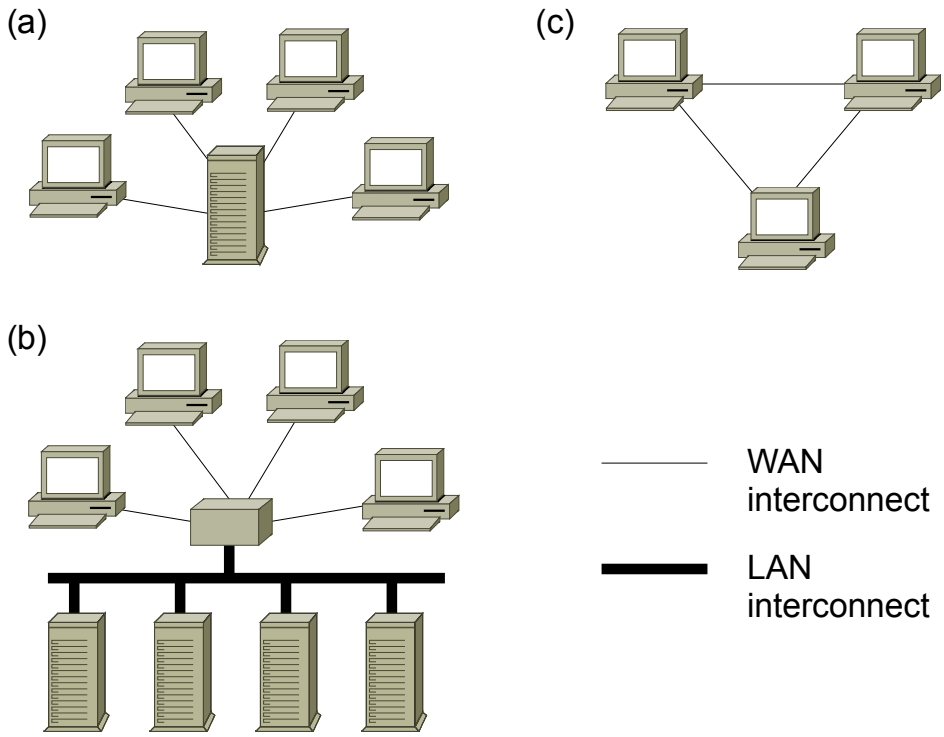


Figure 2.14: Three DVE architectures with geographically distributed users. (a) Centralized server architecture; (b) Distributed server architecture; (c) Distributed serverless architecture. Adapted from (FUJIMOTO, 2000).

both) across several server machines. Note that the WAN communication cost between the server cluster and the clients is the same as in the single-server scenario. As discussed previously, the server machines will have to exchange messages to achieve synchronization in addition to the client-server messaging. Computation-only entities are assigned to servers using a criteria that is probably the same criteria used to distribute player-controlled objects, which is almost always one that tries to group nearby objects in the same LP to minimize inter-server messaging;

- *Distributed serverless architecture:* In the serverless approach there is no central location and the distributed simulation is performed by a group of geographically distributed machines. If these machines happen to be client (player) machines and the DVE has security concerns (e.g. competitive games), then cheating becomes a problem. In this case, replication among the simulators is more common than partitioning since that allows distributed, anti-cheating verification procedures. For example, if it is assumed that less than 1/3 of the replicas are faulty, then a Byzantine agreement (LAMPORT; SHOSTAK; PEASE, 1982; DOLEV, 1980) is enough to separate the 'honest' version of the simulation state from the ones computed by malicious or faulty LPs.

This classification is useful as a starting point for understanding how a DVE simulator can be distributed. Some DVE designs can be supported by more than one of the techniques above, and it is possible to derive a new architecture from these three archetypes. Below we present some examples of these situations.

A sharded MMOG, that is, one composed of several isolated instances, can be supported with either the *centralized server architecture* or the *distributed server architecture* as exactly presented above. In the centralized case there is one server with a copy of the entire or part of the game content. In the distributed case there is a cluster of computers supporting each shard, with each computer running one or more LPs. It is likely that the synchronization mechanism employed between the LPs internal to a shard of a sharded MMOG will take advantage of (a) the limited scalability requirements of each shard (usually under 100,000 players for current MMOGs); (b) near-zero latency and high bandwidth communication between LPs (fast LAN bus or cluster interconnect technology) and (c) the high reliability of the machines and trustworthiness of server-side simulation. This makes things much simpler than designing a synchronization solution with LPs distributed across unreliable, high-latency, low-bandwidth and generally untrustworthy player machines.

A way to implement a peer-to-peer DVE would be to have player machines connect to each other and send events to each other directly in a truly *peer-to-peer* fashion as in the *distributed serverless architecture* above. However, it could be extended to increase fairness, consistency and cheat-detection by adding a 'helper' server component that performs some light-weight functions. Pellegrino et al. (PELLEGRINO; DOVROLIS, 2003) propose the PP-CA architecture which allows players to exchange events directly that would normally be sent to a game server, but also makes the players send their events to a central server. The server, however, is generally 'silent': it receives events, validates them, but does not usually send update packets to the players and only intervenes whenever it is necessary to resolve a conflict. For instance, if Player B shoots player C, but player A shoots player B first, it may be the case that player B had no right to shoot C because it was already dead. The server would detect that B's shot was illegal and intervene by sending authoritative messages to all involved undoing B's shot, thus restoring fairness and consistency, and avoiding (to some extent) cheats where a player would lie about having successfully shoot someone.

The PP-CA architecture is not completely serverless, but it is also not completely centralized: it is a hybrid of both architectures. This is an hybrid architecture by any classification criteria, since the server is receiving messages in real-time from all players and thus is an active participant on the main loop of a DVE. Sometimes, hybrid architectures are just the result of juxtaposition: the application is composed of two parallel aspects and each is treated in a different way. An example would be a DVE which is serverless during normal operation but which requires a central server to act as a Certificate Authority (CA) which just performs CA-like infrequent operations such as issuing a certificate for a new player. In this case, if 'architecture' means the complete DVE solution then it can be said to be a hybrid. However, if the CA component is abstracted away then the DVE architecture in question could be considered a serverless architecture in practice.

Fujimoto's taxonomy doesn't explicitly cover geographically distributed servers, which have since then been explored by many works in MMOG support (see Section 2.7). However, most of these works assume low-latency and high-bandwidth links between server sites, a fact that blurs the line between the classic definitions of LANs and WANs. That is, if two nodes are separated by thousands of kilometers but they have a dedicated network connection that transmits data at near-light speed and with gigabit bandwidth, then that approaches the quality of service of LANs. Many works in MMOG server-side distribution assume such interconnects, thus in practice falling in Fujimoto's 'LAN distributed server' category.

Informally, it might be useful to abstract peer-to-peer logical (socket) links as either 'long distance and last-mile' (slow, 'WAN') or 'others' (fast, 'LAN'). Pairs of end-user peers (players) from Brazil and Japan exchanging packets will likely have latency, congestion and packet loss problems. However, peers inside the same country, considering a country with a good Internet infrastructure and traffic-neutral ISPs, will likely not, at least most of the time, even if this would be classified as 'WAN' due to geographic distance (country-wide). Also, two servers from the same MMOG-serving company, each in a different country, may experience latency proportional to the geographic distance, but the link will probably be dedicated and as such will not be subject to significant packet loss and congestion, distancing itself quite a bit from the 'WAN' label and approaching the 'LAN' label, though still having latency physically limited by geographic distance and the speed of light. So, geographic distance is not sufficient for characterizing the usefulness of a link in building DVE architectures, though it is still an essential component and a good overall indicator of what to expect in terms of network service quality.

## 2.4 Client-server game networking basics

This section offers the reader an insight on the actual content of messages exchanged by client and server machines of highly interactive games. In essence, we explain what a 'client' communicates to a 'server' and vice-versa. The focus is on what is really tried and proven to work for existing *action games* such as FPS games. The focus is on action games because they are the ones that strain the underlying network the most, usually requiring low latency, low packet loss, low jitter and high bandwidth. This, in turn, results in action games having very interesting client-server protocols that attempt to closely emulate the single-player experience by, for instance, hiding network latency from the player.

As mentioned earlier, most multiplayer online games (MOGs) are based on a centralized server architecture. In this architecture, a single machine (the *server*) provides synchronization for all player machines (the *clients*) connected to it. All messaging has to go through the server, which is a single point of failure and where most of the 'intelligence' of the system is situated. This contrasts with the 'distributed simulator' concept presented in the previous section, with emphasized distributing the simulation load across symmetric LPs. Thus, we have followed the approach of Fujimoto's simulation book (FUJIMOTO, 2000) and we tackle MOG (DVE) synchronization separately in this section.

The popularity of the client-server architecture for MOGs is in its relative simplicity, if compared to some peer-to-peer[13] or distributed server architectures. However, the task of actually realizing the networking aspect of a client-server game such as a small-scale 3D FPS game is not trivial. At first, a single-player game engine can perform instant communication between any two objects inside the simulator. With added multiplayer, the state of the game has to be either partitioned or replicated (or both), requiring distributed objects to synchronize with their copies through an imperfect network.

Pragmatic solutions and palliatives for most online game networking woes are well

[13]There is a perception that peer-to-peer can be simpler than client-server systems due to the former requiring only one, symmetric process to be designed (the 'peer') as opposed to the two processes of the latter. However, this depends on the protocol and application. A peer-to-peer search system that uses flooding for query propagation takes advantage of the symmetry, whereas a competitive game where participants don't necessarily trust each other to modify a shared state is vastly simpler to implement with clients (untrusted) and servers (trusted) separated.

known to the games industry. After all, MOGs are popular and have been performing acceptably over the Internet since the 1990s. In this section we will explain how online games compensate for the undesired network characteristics such as high latency, packet loss and jitter by applying some techniques. This section gradually evolves a simple example client-server action game protocol to demonstrate these techniques. At first, we present a 'terminal' protocol which is the one of the simplest client-server game protocols possible but that presents many problems due to not dealing adequately with network imperfections. For instance, the initial protocol doesn't try to hide the fact that a player command to move an avatar locally must first be acknowledged by the server, making the client's command await the network to be applied locally. This reduces responsivity and can break the illusion that the human player is in control, turning the game unplayable.

To counter the inadequacies of the basic protocol, it is augmented with interest management, local input prediction, remote object interpolation and extrapolation, and finally an advanced technique that is specific to 3D shooter games: lag compensation for instant-hit weapons (BERNIER, 2001). The augmented example protocol is much more representative of how existing online action games such as Counter-Strike actually work.

### 2.4.1 Using client-server for a shooter game

A *shooter* game is a kind of action game where the main activities performed by players are *moving* an avatar around a virtual space and *shooting* at other avatars, trying to hit them. Though the simulation is actually a discrete-event simulation, movement of objects inside a shooter game must appear to be continuous. This means that the simulation must perform updates to object positions in the order of tens of times per second to present the illusion of motion. This is trivial to accomplish in a single-player, non-networked shooter game: just implement a single-threaded loop that alternates between collecting user input, updating object positions and drawing the next graphics frame. If this can be done quickly enough, the user will feel that he is interacting with a virtual space, with moving objects, collisions, etc.

However, in a multiplayer scenario, the player's machines have to synchronize over the Internet. Each player will need to hold a version of the world's state to draw on their respective screen. With this, the issue of synchronizing these views arises. As we have seen previously, synchronizing views of distributed players is quite complex. Besides the issues of synchronization and conflicting simultaneous changes to the shared state discussed in Section 2.3, there is the issue of trust: anonymous players of a competitive online game don't trust each other. When given the chance, players will cheat by, for instance, sending fake synchronization messages that misrepresent the timeliness or the precision of the actions performed locally.

The easiest way to solve these issues is to choose a special machine (or 'LP') in the system to act as a trusted simulator: the server. By communicating through a server, all player machines (now *clients*) can synchronize easily and trust-related problems are minimized. Synchronization is achieved by having the 'official version of the simulation'[14] be dictated by the trusted server. This makes conflict resolution much easier, as the clients

[14] In this thesis we will use 'official' (as in 'official state' or 'official version') to denote an element inside a set of alternative states which is considered the correct one. We do not use 'correct state' since we will usually employ the term 'correct' to mean semantic integrity (compliance to game rules). By contrast, all the alternative versions of a simulator state may be 'correct' (i.e., in accordance to simulation laws) but nevertheless be different from each other. In that case, usually one of these versions will be the 'official' one because of, for instance, being kept by an authoritative source such as a central server.

just have to re-synchronize to the server's state continuously to maintain convergence. Also, some problems such as out-of-order event execution can disappear by simply having the server timestamp all incoming events.

This shows that client-server is the more practical way to support a client-server online action game. There are downsides, such as the server being a central point of failure and a performance bottleneck, usually needing to run in a machine with more processing capacity and more bandwidth than client machines. However, the popularity of client-server for MOGs shows that these issues are more tolerable than having to solve peer-to-peer synchronization or trust-related issues.

### 2.4.2 The basic client-server shooter protocol

This section describes an example, simple client-server protocol that could be employed to support a shooter game. The protocol is based on UDP. There are only two UDP packets in the protocol:

- **Client-to-server input packet.** This packet contains the raw and absolute state of all relevant input devices at the client, which were sampled shortly prior to the packet being sent. For instance, for a range of relevant keyboard keys, the packet can encode each in one bit as 'key is pressed' or 'key is not pressed';
- **Server-to-client update packet.** This packet contains a snapshot of the entire game state at the server just prior to the packet being sent.

The whole system is shown in Figure 2.15. The client continuously polls its current input state and dispatches a packet to the server. The interval between client packets can be fixed at, for instance, 50ms between packets, resulting in each client sending 20 packets per second[15]. These input packets in average should be very small[16]. Upon receiving an input packet from a client, the server simply saves the client's input device state, overwriting the previous state.

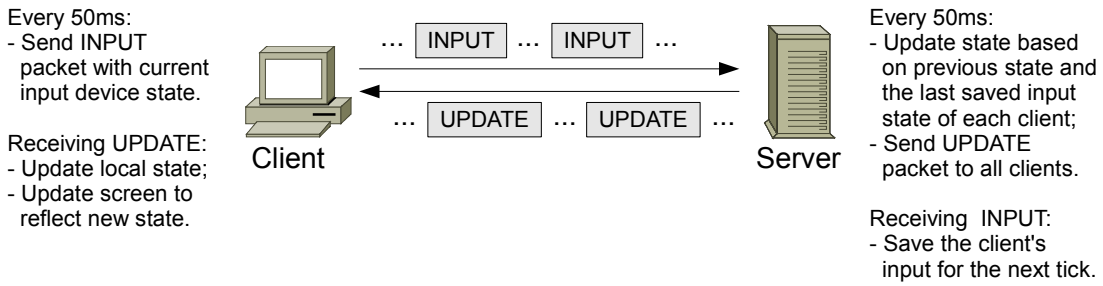


Figure 2.15: Example system running a basic client-server game protocol with a fixed tick rate of 20Hz for both clients and server.

In its main loop, the server must continuously perform two tasks. The first task is running the game physics: updating (recomputing) its copy of the game state. The second

[15] As a reference of client-to-server packet rates for shooter games, the developers of the Source game engine, which backs Half-Life 2 (a 3D FPS game), recommend setting the rate of client packets to somewhere in between 20 to 30 packets per second, which would correspond to an interval of 50ms to 33ms between each outgoing client packet (Valve Software, 2009).

[16] As an extreme example, the open source game Outgun (RITARI et al., 2006) (a 2D shooter game) encodes all client input device state in a single byte.

task is updating clients: sending an update packet to each client. These tasks can run in different frequencies, but for simplicity we assume that the server performs a single *update* step which both runs the physics to update the server's state and, right after that, sends an update packet to each client contaning the newly computed server state.

When performing a physics step, the server advances all objects in simulation time proportionally to the time interval between steps. For instance, if the server is running at 20 steps per second, each physics step will advance objects in 50ms of simulation time. The new state is a function of the previous state and the client input device states. For instance, if the previous state contains an avatar at rest and the current client input tells that the player has the 'shoot' key pressed, the next state should show the avatar firing its weapon or preparing to do so, if allowed by the game rules (e.g., if the avatar had available ammo in the previous state).

In our basic protocol, the update packet sent by the server always contains all game state. For instance, in a shooter game, the update packet would contain the current position, viewing angle and animation frame of all player avatars. The only thing it would not contain is constant information, such as the *game map* (terrain and structures), if the game happens to have an immutable map. The clients, upon receiving an update packet, simply display the contents of the packet to the screen.

### 2.4.3 Analysis of the basic protocol

With the basic protocol presented above, a simple action game can be supported. Notice the absence of a mechanism for delivering messages in order or reliably: all new messages in a stream completely supersede all previous others. Notice also that the client is a trivial terminal, with all simulation performed by the server. Though simple, this basic system performs poorly in practice. The main problems are the following:

- Client keypresses can be missed. For instance, if the player presses a key and releases it between outgoing input packets, the server will miss that keypress event. This can be solved if the client uses a reliable messaging channel (e.g.: using TCP or a custom packet retransmission protocol over UDP) to send its input to the server;

- Conversely, some server-side happenings can be missed. For instance, an avatar is in 'dead' state for only a single server step and a client does not receive the respective update packet in which that avatar is reported dead. In this case, the client code is not able to detect that the avatar has died. This can be solved in several ways. In practice, this is solved by implementing additional server-to-client reliable messaging channel for unmissable events;

- The fixed rate of outgoing input packets introduced unnecessary latency into the interaction loop. A way to solve this is having the client send an input packet immediately after any input device state change instead of sending packets in a fixed rate. However, this can cause an overflow if the client presses too many keys in a short time, or with input devices that can change state hundreds of times per second such as mouses. Anyway, this is a minor issue, but worth mentioning since reducing latency is paramount for action MOGs;

- The protocol does not take into account different link capacities at each node, nor it adapts to changing network conditions. Some clients and servers may not be able

to cope with the (fixed) data rates. Optimal results are achieved if the rate of ticks or messages is adapted to match each machine's capacity[17];

- The server always sends the full game state to each client, wasting bandwidth and aggravating the problem of state-revealing cheats. In an avatar-based virtual world, a client is only interested in state updates for objects that are near its avatar. So, server update messages can be smaller if they only include information that is relevant to each client. Additionally, sending irrelevant information to a client, such as positions of enemies that are behind walls, may allow a malicious client process to reveal that information on screen, giving an unfair advantage to the player which can now see its enemies behind walls, an obvious tactical advantage. In a following section we will add interest management to the server which helps to address these issues;

- The client cannot execute its own commands immediately. The client has not enough built-in intelligence to know what to do with its own commands: it only sends keypresses and receives the results of executing those keypresses at the server. From a simplicity standpoint this is a good thing, but this also means that the client is forced to wait for a full client-server network round-trip between the time the player issues some command (e.g.: shoot) and the time it is perceived on screen. Depending on the game type, this latency can break the illusion that the player is in control. If the network latency is significant, responsivity can be reduced to the point that the game becomes unplayable. We address this in a following section by adding a client-side input prediction mechanism to the protocol;

- Movement is perceived as choppy at the clients. After receiving an update from the server, the remote objects at the client's screen are instantly moved to the reported position, and they stay on these positions for a full frame time until the next update packet is received. One solution would be to just send more packets until the illusion of movement can be sustained (e.g. 60 packets per second). However, that wastes bandwidth and similar results can be obtained with remote object interpolation and extrapolation (added to the protocol in following sections);

- The protocol may be unfair to clients on high-delay connections. The wall-clock time in which client commands happen are not taken into consideration as all timestamping is done at the server side. This causes low game fairness (BRUN; BOUSTEAD; SAFAEI, 2006) for players which have a high delay to the server, which will have less time to react to the moves of their opponents that have a low delay link to the server. There are several ways to ameliorate this situation, such as letting players inform the current position of their avatars, the simulation time in which their commands happen, the simulation time in which they have reached some position with their avatars or performed some action (e.g. fire a weapon), among others. In the next sections we will address this with the client-side input prediction mechanism for movement and a 'lag compensation' mechanism for the instant-hit weapons of our hypothetic example game.

[17]In practice, most action MOGs allow the rate to be configured individually at each client or server, but adaptation to changing network conditions is not commonly supported. The assumption is that both clients and servers generally know the typical bandwidth that is being provided to them. Whenever a node's bandwidth drops, that is assumed to be temporary so there is limited usefulness for a long-term rate adaptation mechanism.

Besides these problems, the simple protocol actually presents an advantage: it is pretty resilient against some common cheating techniques employed in 3D shooter games such as Counter-Strike. For one thing, the protocol is immune to speed cheats (FUNG; LUI, 2009), since the client is not allowed to inform its position to the server. It is also immune to timestamp-based cheats (DI CHEN; MAHESWARAN, 2004) since the client does not even transmit client-side simulation time information. Typical commercial game protocols include both client-side state information (namely, position, speed or other exploitable derivative) and client-side timestamps, which explains why many online games have been known to be compromised by these kinds of cheats (FUNG; LUI, 2009). We will explain these cheats and others in Section 2.5.

### 2.4.4 Saving bandwidth with interest management

The first optimization that can be done to the protocol is to reduce the size of update messages. Currently, the update packet contains updated attribute values for all of the objects in the game. So, if there are N avatars in a shooter game, each one of the N clients will receive an update packet which always contains the most up-to-date position of all N avatars. In a MMOG, N can be in the order of thousands and this scheme is clearly not scalable. Even for a small-scale game it is interesting to reduce the size of each update message in order to conserve server bandwidth, which is the network bottleneck of a client-server MOG.

In the typical shooter MOG avatars are scattered around a maze and an avatar can only 'see' a subset of all other avatars since most of them will be occluded by the game map (walls). Thus, instead of blindingly building a single update packet and sending that to all clients, the server can perform an additional check for each client to determine what is the scope of that client's *interest* for the current simulation state. Then, for each client, a custom update message is assembled which only includes information for which the client is interested in.

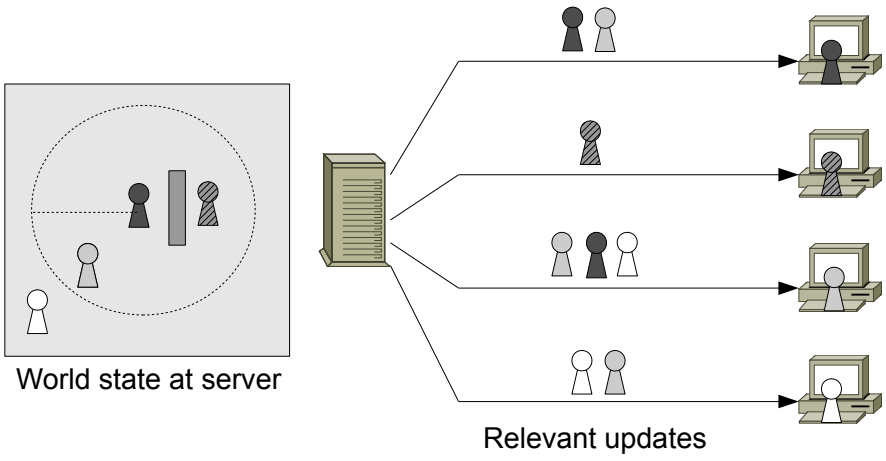


Figure 2.16: Example of relevance filtering being applied to server update messages.

Figure 2.16 shows an example scenario. In the figure, the hatched avatar is behind a wall and is not relevant to any other avatar. Conversely, no avatar is interested in the hatched avatar. The gray avatar is between the white and black avatars, and thus they are relevant to it and it is relevant to both. The dotted circle in the figure represents the *area of interest* of the black avatar. It is used to detect that the white avatar is too far

away from the black avatar to be relevant to it. An area filter is useful in a large-scale environment because it prevents the execution of more refined (and expensive) criteria of relevance such as traces, which can be too expensive to compute for a large number of avatar pairings. Conversely, the white avatar, which also has an identical area of interest (not shown in the figure to avoid visual clutter) considers the black avatar irrelevant.

Thus, we augment our hypothetical server with a *relevance filter* for clients: for each client, send an update message which only contains relevant avatar position updates. This can be easily accomplished in most game engines by executing traces from one avatar to the other and determining whether opaque geometry is hit or not. This saves bandwidth and, as mentioned in the previous section, helps to prevent state-revealing cheats, where a player sees advantageous information that it shouldn't be able to (more on cheating on a following section).

There is more to client-server interest management (IM) than simple boolean filtering at the server. If a client has not enough bandwidth to receive all information after the filter, the server must then prioritize the individual update items in the packet and choose the most important ones to send. Prioritizing individual update items is not so trivial, as some pitfalls must be avoided such as starvation of individual update items that never get chosen for sending. Frohnmayer and Gift (FROHNMAYER; GIFT, 2000) describe the network engine of the Tribes series of FPS games, which performs this kind of prioritization.

Lastly, it should be noted that realizing IM optimizations is more complicated for peer-to-peer games or VEs. This is because that in a truly distributed (decentralized) virtual environment simulation there will not be a single 'server' node where all filtering can be concentrated.

### 2.4.5 Smoothing movement with client-side extrapolation

The rate of server-to-client network packets in online games is significantly lower than the rate of graphical frames that a client machine can render. For instance, a client may render 60 graphical frames per second while it is only fed 20 network frames (updates) per second from the server. In this case, and by following our basic protocol, the client would just redraw the screen three times with the same update data between updates. The human player would only perceive 20 different graphical frames of animation, which looks choppy.

The first technique that can address this issue is extrapolation. It is also known as *dead reckoning*, though some use that term to imply ballistic position prediction, with constant acceleration or velocity. The basic idea of extrapolation is to update the state of the local 'ghost' objects at the client independently from their master copies at the server. In other words, instead of redrawing a graphical frame that shows the same world state than the previous frame, the client process instead extrapolates the current state of the world proportionally to the amount of time elapsed since the last received update and then draws that guess to the screen.

Usually, extrapolation is employed first and foremost for remote object positions, and the extrapolation function is usually a simple ballistic equation that assumes that the last known velocity (or the acceleration, if informed) is constant. An object's extrapolated position is calculated based on the object's last known good position, velocity and, optionally, acceleration. With some luck, the extrapolated guesses are correct and the graphical frames end up 'filling in' for the unavailable network frames.

Figure 2.17 shows an ideal scenario where a remote avatar's velocity remains constant between two network updates that are 50ms apart from each other. The example shows

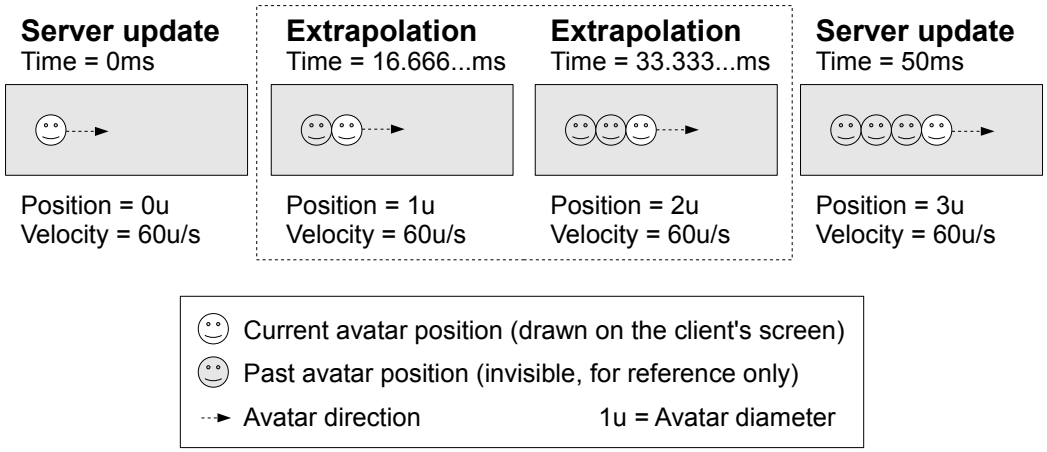


Figure 2.17: Example of correct client-side extrapolation of a remote object's position between the receipt of two server updates that are 50ms apart from each other.

the game state evolving at the client, the current time at the client and whether the frame is an extrapolation or a server packet. In this ideal example, the client is showing a constant 60 frames per second, and the server is providing a constant stream of 20 updates per second. Thus, there are two fill-in guess frames for each 'official' frame from the server. For simplicity, the example shows a single avatar moving along an unidimensional world.

The downside of this technique is that it breaks when the assumptions of the extrapolation function are wrong. For ballistic position extrapolation this means a wrong guess happening when the remote object being extrapolated changes velocity or acceleration.

Correcting a wrong extrapolation can cause more visual artifacts than the ones avoided by the technique. In a game like the open-source game Outgun (RITARI et al., 2006), where avatar movements are highly inertial, almost ballistic, linear extrapolation can be applied judiciously. However, this doesn't hold for the typical 3D FPS online game, where avatars can perform sudden changes in movement direction and speed. In 2001, Bernier of Valve Software, developers of massively popular shooter games such as Half-Life and Counter-Strike, hinted at limiting extrapolation to at most 100ms of real-time from the last received server update (BERNIER, 2001). More recently, a developer-oriented page at Valve Software's website cites that their Source engine sets a default of 250ms for the extrapolation limit (Valve Software, 2009).

There is more to extrapolation (or *dead reckoning*) than simple linear position extrapolation. Besides just limiting extrapolation outright, one way to minimize the impact of wrong guesses is to interpolate between the wrong position and the new, correct position, instead of just resetting the object's position to the correct one. Aggarwal et al. (AGGARWAL et al., 2004) show that dead-reckoning augmented with timestamp information can result in better visual consistency. Yonekura (YONEKURA, 2006) proposes an augmented extrapolation technique that outperforms simple dead-reckoning. Also, some works attempt to predict the behavior of objects that don't respond well to simple linear extrapolation (MARSHALL et al., 2004; LI; CHEN, 2006). Pantel and Wolf (PANTEL; WOLF, 2002) have assessed the usefulness of dead reckoning techniques for games, considering different types of games.

Finally, it should be noted that this extrapolation is employed for remote objects and not for the player avatar, which is (generally) the only local object in an avatar-based on-

line game. In a following section we will add client-side input prediction to the protocol. Input prediction extrapolates the local player avatar's state, but it does so using the local player's commands as basis which guarantees that wrong guesses are much less common. This is necessary since correcting the position of the local avatar is much more disturbing to the player than correcting the position of a remotely-controlled object.

### 2.4.6 Smoothing movement with client-side interpolation

In our current protocol, the clients always perceives the world with delay. When the server computes a new version of the game state, which is definitive and authoritative, clients must still wait for the server update packet to see what really happened, since the server's simulation step function is the official source of simulation time and state advancement for all clients. To keep a player's view feeling like a 'current' version of the world, its client module has to resort to showing guesswork while a new server update doesn't arrive.

Interpolation is a technique that **adds** a constant amount of delay (the *interpolation delay*) to this loop, obtaining in return an increase in visual smoothness of remote object movement. It is especially useful for objects such as humanoid player avatars whose movement patterns, considering the typical 3D shooter game, are not well suited for linear extrapolation for long periods of time.

In essence, interpolation is buffering incoming server updates at the client. Consider a scenario where the client is showing an object on screen at position $P1$, and then an update packet arrives with the new position $P2$ for that object. With interpolation enabled, instead of showing the object at $P2$ immediately, the client instead postpones moving the object to $P2$ for the duration of the interpolation delay. At the end of the interpolation delay, the object is shown at $P2$. In the mean time, the client **interpolates** between $P1$ and $P2$ and draws the interpolated positions to the client.

The interpolation delay can never be too large or else the benefits are negated by the added visual delay. Usually, interpolation is employed only to the point where the effects of infrequent packet loss and network jitter (late packets) are sufficiently mitigated. As a reference, Valve Software uses 100ms of interpolation delay as the default in their Source game engine, together with a 50ms default for update packet inter-arrival times (Valve Software, 2009), which is sufficient to cover for any single lost packet in the update stream.

Interpolation and extrapolation (dead reckoning) can be used together. If the client runs out of data to interpolate to, that is, if the last target point in the buffer is reached, extrapolation can kick in until the next update arrives. Together, the constant interpolation delay and the extrapolation limit can cover for small sequences of packet loss in the update stream before the client has to freeze the screen[18].

### 2.4.7 Restoring responsiveness with client-side input prediction

The current protocol has a serious issue, as pointed earlier: the player has to wait at least full network round-trip to the server to perceive the effects of its own avatar commands. If the network distance between participants (or the server) is too great, this can turn the game unplayable. Most client-server action games such as FPS games choose to address this issue with input prediction.

[18]Freezing is employed to avoid misrepresenting a possible connectivity loss. If that's not an issue, extrapolation can be kept running forever with no client screen freezing.

Input prediction is above all a concept: that of extrapolating the movement of a local avatar based on the privileged information of its own input. Thus, if implemented properly, the end result is an extrapolation of the local avatar state that is almost always right. It is extrapolation because, ultimately, the server will have to validate it. Our goal here is not to provide a detailed explanation of input prediction techniques, but rather provide an overview. Bernier (BERNIER, 2001) gives a good explanation of input prediction as implemented in the Half-Life game engine. Sanglard (SANGLARD, 2009) has examined the source code of QuakeWorld, one of the main ancestors of FPS online game engines, including its prediction code.

With input prediction, the client runs a simulation loop similar to the server loop, and it moves the local player's avatar by an amount of simulation time that corresponds to the amount of real time that elapses at the client. Here, it makes more sense to let the client run that client-side simulation function before each graphics frame is drawn, instead of dictating a fixed rate. But, since the real time it takes for the client to process each frame may vary, this means that each simulation step at the client will advance simulation time by a variable amount.

As for our example protocol, input prediction requires a new client input packet and new server logic. Now, instead of just sending the state of input devices, the client sends a stream of 'commands', where one command is generated for each simulation frame at the client (which, as discussed above, now has variable duration). Each command is comprised of input device state and the frame time for the command (its duration). When the server receives a command, it can either execute it immediately over the respective avatar (the solution adopted by the Half-Life and QuakeWorld games) or queue them for execution on the next server tick, which now does not make much sense since each avatar is now moving by a specific, reported quantum of simulation time. The protocol can also be modified to have the server send an update right after receiving a client input is received. This further reduces the interaction latency loop and minimizes the negative effect of a wrong input prediction at the client since the latter will have less time to execute prediction in a wrong direction.

Notice that, to support input prediction, the notion of a 'dumb terminal' client has to be dropped entirely. This also helps to show that the notion of simulation time advancing uniformly at a simulator is, after all, just a notion. Simulation time and world state are just concepts, and there is nothing preventing a simulator from simulating objects individually by arbitrary amounts of time, from moving objects to arbitrary positions, or from predicting some attributes and not others.

As an example of selective attribute prediction, shooter games usually do predict future movement but they do not predict the effects of hits. A client may predict, during its local frame simulation, that a flying rocket from an enemy moves inside (or through) its own avatar, but it does not execute the animations of rocket explosion or avatar death. This is because if it's predicted wrongly, the effects of undoing an avatar death and a rocket explosion cause too much of a negative impact to the user to compensate for the positive effects of the prediction.

### 2.4.8 Lag compensation for instant-hit weapons

According to Bernier (BERNIER, 2001), *'Lag compensation is a method of normalizing server-side the state of the world for each player as that player's user commands are executed. You can think of lag compensation as taking a step back in time, on the server, and looking at the state of the world at the exact instant that the user performed some*

*action'*.

Lag compensation is a solution for the following problem. If a player shoots at what he is seeing, he won't hit it at the server because he is seeing the server state in the past. Even if the server executes the shoot command with compensation for the client-to-server hop that this command travels (what could be part of input prediction support – or not), this would still not cover for the other half of the interaction loop's delay, that is, the delay for the server-to-client update that fed the version of the world that the client was seeing when he shot.

The Lag compensation algorithm further breaks the notion of an unified simulation time advancing uniformly at all nodes. The algorithm works as follows (as described in Bernier (BERNIER, 2001)). Notice that it assumes a protocol with input prediction where the client sends a stream of commands that it has already executed locally with a specific duration for each:

- Before executing a player's current user command, the server:
  - Computes a fairly accurate latency for the player;
  - Searches the server history (for the current player) for the world update that was sent to the player and received by the player just before the player would have issued the movement command;
  - From that update (and the one following it based on the exact target time being used), for each player in the update, move the other players backwards in time to exactly where they were when the current player's user command was created. This moving backwards must account for both connection latency and the interpolation amount the client was using that frame;
- Allow the user command to execute (including any weapon firing commands, etc., that will run ray casts against all of the other players in their 'old' positions);
- Move all of the moved/time-warped players back to their correct/current positions.

Also notice that the client has to send more information to the server. It sends the ID of the two world updates (update packets) that it is currently using for interpolation, and the current point in time in which it is relative to the first update, that is, where in between the two updates the client is interpolating, or whether it has already moved past the second update and into extrapolation. The server then moves the world back in time taking into account an estimate of both major factors of latency, which are the network latency and the latency of the client simulator.

Though lag compensation increases consistency for players that are shooting at other player avatars, it has a tradeoff. Figure 2.18, which also considers that input prediction is in effect (that is, the server sees player movements with delay), shows what is happening at two clients and the server in two sections of wall-clock time. In the first row of world states, Black shoots at White and hits. In the second row of world states, all nodes have exchanged one round of packets: Black's shoot action arrives at the server, together with White's movement even further into a cover area. The server compensates for lag in Black's shoot action, resulting in White being hit. Later, White will receive a world state where it has been hit, and this will look inconsistent to the player which think it is safely covered from shots. Bernier (BERNIER, 2001) argues that this issue is not so serious

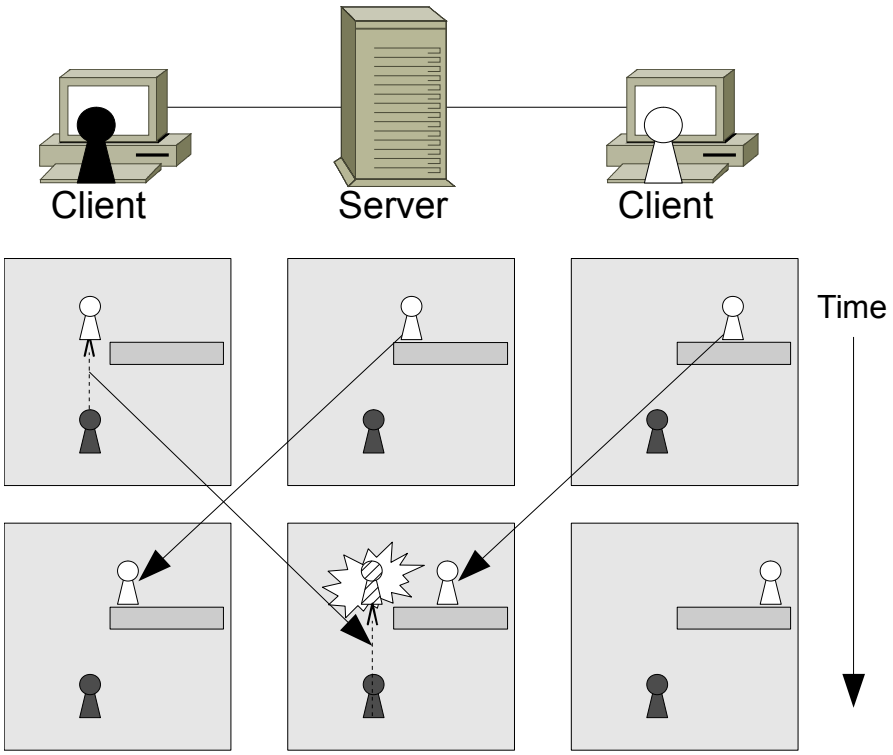


Figure 2.18: Possible execution of lag compensation illustrating its consistency tradeoff. Black sees the hit, the server later detects the hit at the lag compensated target position (hashed avatar). Later, White will perceive its avatar being shot behind cover (not shown in the figure).

because players usually run towards each other when shooting and 'shot behind cover' situations are rare in actual matches of 3D FPS games.

Besides Lag compensation, other solutions to this include just doing nothing and letting players learn how to lead their targets, and extrapolating remote avatars at the shooting client to instead match what the server is seeing when the update is received at the client. The latter is a sort of reversal over interpolation and is comprised of never actually displaying incoming position updates as-is but instead considering them always late and immediately adding the server-to-client (one hop) latency to them and visually extrapolating. This way, the player is always seeing an extrapolated world view. We have performed some limited experiments with that solution for a peer-to-peer shooter game support and found that it is not visually acceptable (CECIN et al., 2006).

### 2.4.9 Closing remarks

This section gives some insight on how client-server action games counter latency. The presented solutions were designed for avatar-based and client-server action games. We can argue that so far no applications have exercised advanced latency compensation techniques in practice more than commercial online 3D FPS games have. Thus, in this section, we have actually went far into what can be done to compensate for latency, though we have only provided a short overview of each presented technique.

As we have seen, as more latency compensating techniques are added, the simulators and the protocol become more complex. Notably, the assumption of uniform simulation

time advancement is broken. This can be a problem because sometimes uniform time advancement is desirable. Specifically, in FreeMMG (CECIN, 2005) and FreeMMG 2, we require, for anti-cheating and fault tolerance, that the state of the simulation is replicated among several peers, and by replication we mean peers having bitwise identical copies of world state. For instance, if we take snapshots at two peers, we want to be able to hash them and compare hash values to confirm that synchrony is being maintained. If simulation time is not advancing uniformly at each peer or if the size of simulation steps on each peer are not the same, then different peer snapshots are not comparable.

Thus, marrying good latency compensation techniques with some new architectures (for instance, a peer-to-peer architecture that relies on replication and snapshot comparison...) may be a problem. This can be a show-stopper issue if, in addition, support for action games is being considered for the new architecture. This was a problem in the original FreeMMG, which ended up being limited to Real-Time Strategy game support. In FreeMMG 2 we tried to address this issue, allowing latency compensation techniques to be employed together with a replication layer. This will be explained in the next chapters.

## 2.5 Cheating in online games

One of the major concerns of online game designers is the issue of players that attempt to cheat at the game. The fundamental problem with cheating in online games is that as more cheating is performed against a game, the less meaningful becomes the idea of putting effort into playing the game by its rules. With enough cheating actions being performed in instances of an online game service, it can become a pointless activity and frustrated players quit. Since any form of revenue (monetary or not) obtained from an online game is proportional to the amount of players interested in it, game designers therefore attempt to prevent or reduce cheating in their games. This is also why cheating in single-player, offline games doesn't matter: only the cheater itself is affected[19].

The ways in which games can be cheated are as numerous as the types of games that exist. A cheat may be a serious threat to a game type while for another game type it may be a minor annoyance[20]. For instance, in a first-person shooter (FPS) game, if a player cheats the game to have automatic, perfect aiming capabilities, then the game is virtually ruined for the opponent players. However, such a cheat would be just an annoyance in an MMORPG game such as World of Warcraft, where aiming is a secondary activity performed to select enemy targets for attacks – the attacks themselves being automatic and a result of server-side dice rolls.

There are several works that attempt to create classification schemes for cheats or that list and explain the possible cheats in online games (WEBB; SOH, 2007; YAN; RANDELL, 2005; MCGRAW; HOGLUND, 2007; LAURENS et al., 2007; JHA et al., 2007; CRONIN; FILSTRUP; JAMIN, 2003). The classifications usually organize cheats in several dimensions, such as the methods (technique) used to cheat, the advantages gained, or the architectural level where it happens (application, protocol, infrastructure, etc.). In this section we explain some cheats that are not specific to peer-to-peer MMOGs but that

[19] A counter-example would be a single-player game which submits high-scores computed locally to a shared online scoreboard. If the scoreboard is meaningful and not just a side feature, we consider the game to be a MOG however and thus cheating in the single-player game matters to the multi-player scoreboard meta-game.

[20] If cooperative virtual reality applications are considered as games, then one can argue that cheating in general is only a minor annoyance to that type of game. In fact, cheating is not considered an issue in virtual reality or collaborative virtual environments works.

can be significant in that context. We split the selected cheats in the following broad categories: time cheats, information exposure cheats, automation (bot) cheats and state cheats.

In this section we also discuss what can be done to counter cheats. There are two main approaches to combat cheating (NEUMANN et al., 2007): detection with later punishment and the preventive approach. With prevention, the architecture is designed to be resistant (or sometimes immune) to cheats. With detection, the cheat is allowed to happen sometimes, but it can be detected immediately, or later, with detection guarantee or with a probability. After detection, the cheat can be corrected or at least partially compensated for and the cheating player punished, all this with automation or sometimes with human intervention (e.g., a game administrator manually identifies and bans a cheater from the game or compensates the victims).

### 2.5.1 Damage level of cheats

In this thesis we introduce the concept of the *damage level* of an online game cheat. We assert that it is impossible to design online games that provide the best possible quality of service (consistency, fairness, responsiveness), scalability and immunity to all types of cheats at the same time. Thus, security tradeoffs will have to be made. We also assert that an online game developer should attempt to design the system in a way such that the damage done from security vulnerabilities should be minimized. So, to help ourselves in the process of choosing which security vulnerabilities would be addressed and which ones wouldn't, we came up with the following damage levels to categorize cheats in:

- Imperceptible damage: the cheat is not performed in the presence of other players and enhances the player character for future encounters. This damage level is specific for bots that play MMORPG characters automatically against AI-controlled opponents to enhance the character. These cheats cause minimum damage for other players because the bots in question do not duel with other human players. When other players later encounter the cheated avatar, now controlled by a human, it is then no different than encountering a powered-up character that did so honestly. Players cannot tell the difference among the several thousands of other players in the server, thus the frustration is also minimal. The advantage is also bounded by the time the robot runs and by the game rules, so this imperceptible damage level is actually a special case of the *bounded damage* level (see below);

- Bounded damage: the cheat can be performed in the presence of other players and confers a direct advantage against other players. In shooter games this includes cheats that allow players to have perfect aiming or dodging reflexes, to teleport to arbitrary positions, to run faster than other players or to see through walls. In a MMORPG, these would also apply, with the aggravating factor that their effects would be persisted instead of erased once a new match begins. These cheats have limited damage because, despite their potential of having a high impact on the game, they directly affect only nearby players and the impact is proportional to the effort, that is, the amount of persistent advantage obtained is still bound by game rules even if the rules for obtaining the advantage are broken. For instance, even if a player in a MMORPG cheats others by teleporting first to a treasure chest's location, the contents of the chest (the player's reward for the cheat) is still bound by game rules. Thus, if a good detection strategy is used, the game provider may not even bother with a recovery (undo) strategy for the effects of the cheats;

- Unbounded damage: cheats in this category, once performed, can confer an arbitrary amount of advantage to the cheating player. An example of this would be a MMOG player being able to create any character or any game objects by changing the client program or by changing the messages that are sent to the server. This level is specific for MMOGs where there is a persistent economy to be protected. An unbounded cheat is thus a cheat which allows a single player or a small group of players to wreck the economy of the game. Since the economy is a main factor in any MMOG, an unbounded cheat causes maximum damage that potentially affects thousands of players at once . Client-server MMOGs, for all practical purposes, are immune to unbounded cheats since clients cannot directly write to the authoritative version of the game state at the server. Currently, we consider that only state cheats against a peer-to-peer MMOG can cause unbounded damage.

The goal of this damage level classification is to assess what are the cheats that should be addressed when designing a peer-to-peer MMOG architecture, with the understanding that each problem solved in an architecture will inevitably complicate other issues (FERRETTI et al., 2005). As an example, consider that a design that starts by solving any and all security issues will inevitably end up with higher communication cost and increased interaction delay (GOODMAN; VERBRUGGE, 2008). This is true for almost any distributed system imaginable. Thus, we found out that some cheats should be addressed in a definitive manner by the architecture itself, others should have their solution facilitated by the way the architecture is designed but left for the application to solve, and still others have to be ignored by the architecture design and left for the application to solve entirely.

We hold a strong position that, when designing MMOG middleware, any unbounded damage, economy-wrecking cheats cannot be left to be addressed by future work if the MMOG model in question is actually intended for implementation and deployment. We also argue that using detection for state cheating in peer-to-peer MMOGs, which can cause unbounded damage, is risky. For detection to be viable for state cheating there must be also a reliable way to undo the damage done by the cheat prior to the detection. Since undoing arbitrary state changes can be too troublesome to implement, at least transparently in a MMOG middleware, for FreeMMG 2 we have chosen instead to employ the preventive approach against unbounded damage cheats. In a following section we will describe the state cheat and show that it can cause unbounded damage. In later chapters, we will show how FreeMMG 2 prevents state cheats from happening with high probability.

### 2.5.2 Common cheat implementation techniques

Though there are many ways to cheat in an online game, there are some common techniques used to perform cheating, and it is useful to discuss them beforehand. These techniques are listed below:

- Cheat by modifying the client executable. This is a straight-forward replacement of the client-side code (or peer code in a pure peer-to-peer architecture) with arbitrary code. This is generally easy to perform since determining the authenticity of code in an unsecured machine is hard. Once the client code is compromised, it can generate arbitrary network messages to other nodes, it can modify its own memory, it can present hidden information to the player, etc. To counter this general technique, Monch et al. (MÖNCH; GRIMEN; MIDTSTRAUM, 2006) propose Mobile Guards, a code generation scheme where the client code has to be always

up-to-date to connect the service. These code updates are both automatically generated and they contain code that verifies the integrity of the executable. Kabus et al. (KABUS et al., 2005) reminds that Trusted Computing (TC) may be an alternative for securing the client executable;

- Cheat by modifying the memory of the client process or files read by the client process. This is relatively easier to accomplish than modifying binary code. Basically, the memory of the client process is altered on-the-fly by an auxiliary process akin to a debugger, or the files read by the process are altered before the process is launched. As a simple example, files containing 3D models of avatars can be modified to have brighter colors and thus help the player to spot them and aim at them. In theory, by modifying the memory similar effects of changing the client code can be achieved, such as auto-aiming;

- Cheat by modifying packets through a proxy. By employing an arbitrary proxy process between the game client and its peers (game server or other clients), the cheating player can modify the contents of outgoing and incoming packets at will. If clients are allowed to, for instance, arbitrate over the current position of a player avatar, then a cheater can simply modify outgoing packets to move its avatar to any location (a speed or teleport cheat);

- Cheat by modifying video drivers. This technique is employed to allow players to see through walls by modifying the video drivers in a way that occluded objects blend with the obstacles instead of being obscured by them. This exploits game systems that don't perform relevance filtering when sending updates (or that can't do so) and that send all known objects all the way down to the rendering pipeline, letting the video card do the job of hiding them.

These general techniques cover most common cheats. However, there are many others ways in which cheats can be realized. For instance, a denial of service (DoS) attack can be employed against peers if there is some advantage to be gained from it.

### 2.5.3 Automation cheats (bots)

In many online games, several tasks performed by the human players would be better handled by a computer agent. For instance, aiming a weapon in a 3D shooter game is relatively hard for a human, which has to possess good hand-eye coordination skills. However, it is trivial to program the client process to automatically follow the heads of the player's enemy avatars – a single line of code would probably do the trick. However, the point of the game is to have humans performing the hard tasks and having fun competing with other humans while attempting such tasks. Automation cheats thus are any kind of client-side modifications where hard tasks such as aiming in a 3D shooter, or repetitive tasks such as accumulating virtual wealth or character experience in a MMORPG, are automated by a *bot* (short for robot) program.

Auto-aiming is possible because most 3D games let the clients dictate the viewing angles of the player to the server. The protocols have to allow clients to change their own angles by arbitrary amounts due to the fact that there are really good players that can manage large and precise changes in viewing angles. With this, it is only a matter of adding a malicious proxy between the client and the server that corrects the angles on outgoing client packets in order to aim perfectly at other avatars.

In MMORPGs such as World of Warcraft and even in non-massive RPG online games such as Diablo II, bots are employed to automate repetitive tasks. In RPGs it is often the case that the player is thrown against wave after wave of similar computer-controlled opponents (the *monsters*) and, every once in a while, the character *levels up* (becomes stronger) due to the *experience points* accrued from defeating foes. Needless to say, these tasks are often viewed as chores when they don't present enough variation to the player. Assume that the player, however, does not want to quit the game altogether for some reason. The result is the player employing a program to do the chore for him. Once his character levels up, he is able to face different monsters or to duel against stronger players and, presumably, will resume normal play with the now stronger character.

Bots are also used in MMORPGs for accumulating in-game wealth. When a character performs repetitive tasks such as defeating waves of monsters or completing *quests*[21], it is constantly rewarded with in-game wealth such as currency or valuable items (which are usually interchangeable). Since possessing stronger items such as magical weapons is also a way to build a stronger avatar, a bot can be employed to acquire any desired amount of in-game wealth which can be later dumped at merchants and traded for whatever item is desired by the player.

An example of MMORPG bot is the Glider bot (MDY Industries, 2009). The Glider bot plays characters automatically for World of Warcraft, making them both stronger and richer (with in-game wealth) at the same time.

Several works propose solutions for detecting bots in real-time online games. Yeung et al. (YEUNG et al., 2006) propose a scheme that could detect aim-bots through statistical analysis. Chen et al. (CHEN et al., 2006) attempt to detect bots by analyzing the traffic between a player machine and its server. Thawonmas et al. (THAWONMAS; KASHIFUJI; CHEN, 2008) are able to detect MMORPG bots by analyzing logs of player actions by classifying the behavior, based on perceived patterns, as either belonging to a typical human player or to a typical automated player. CAPTCHA tests have also been cited in several works as a way to detect automated play (THOMAS; ESSAAIDI, 2006; ROBLES et al., 2008; CHAMBERS; FENG; FENG, 2006).

In any case, we believe that MMOG bots cause limited (bounded) damage to the game world since their actions are bounded to what is theoretically possible for a player, human or otherwise, to perform. Again, this is not to say that this is a minor issue, since bots can and do ruin the game experience (THAWONMAS; KASHIFUJI; CHEN, 2008).

As for auto-aiming bots in shooter-like MMOGs (a MMOFPS), we believe that the persistent-state nature of the game and a centralized player database can be leveraged into solutions. For instance, a detection scheme with a very low false-positive rate, but that takes very long to establish a player as an aim-bot cheater, would be sufficient for an MMOFPS. This is because being banned from a MMOFPS imposes a much higher cost to the cheating player than being banned from a single server of a small-scale FPS game such as Counter-Strike. Thus, cheaters will tend to cheat less as their characters accumulate power and wealth, in order to avoid losing it all to a cheat detection algorithm.

### 2.5.4 Information exposure cheats

Information exposure cheats, also called secret revealing cheats (WEBB; SOH, 2007) consist of a cheating player using some technique to gain access to information that it shouldn't have access to. Below are three prime examples of information exposure cheats:

[21] Quests are waves of monsters plus specific interactions with non-playing characters and the environment that resemble storytelling.

- *Wall-hack:* This cheat is most prominent in 3D FPS games such as Counter-Strike, where it allows a player to see enemies through walls. As discussed earlier, this can be achieved if a player's client process is receiving updates for enemy avatars even when such avatars are not supposed to be visible. If that is the case, the cheater can, for instance, modify the client program to render enemy players over the walls. Laurens et al. (LAURENS et al., 2007) show that players with wall-hack enabled follow their targets behind walls and that this behavior is very difficult to avoid. In other words, the crosshairs of a wall-hacking player 'touch' hidden enemies significantly more, a pattern that can be checked by the server;

- *Map-hack:* This cheat is executed mostly against RTS games such as Age of Empires where the protocol requires all players to know about the full game state. Since, in a strategy game, knowing about the whereabouts of enemy troops is crucial, the game client of an RTS usually hides the contents of terrain that the client in question is not actively scouting. This feature is called the *fog of war* in RTS games. The map-hack consists of disabling the fog of war at the client and thus revealing the full game state to the player, which is already available at the client's memory due to the way the protocol works. Buro proposes a solution for map-hack that is specific to client-server RTS games (BURO, 2003), while Chambers et al. (CHAMBERS et al., 2005) mitigate the impact of information exposure in peer-to-peer RTS games;

- *Extra-Sensory Perception (ESP)*: This is used to refer to the exposure of several information items not covered by wall-hacks and map-hacks, such as the exact health value of enemy troops or avatars, and the position of enemy avatars as drawn in an on-screen map in an FPS game (ROBLES et al., 2008; LAURENS et al., 2007);

- *Collusion cheat:* In the context of *information exposure*, the *collusion cheat* consists of players sharing information among themselves where such information was intended to remain a secret from the receiving players (GAUTHIERDICKEY et al., 2004a; WEBB; SOH, 2007; KABUS et al., 2005)[22]. The communication of such information is usually performed in a side channel that doesn't touch the game system, which makes collusion difficult to tackle (WEBB; SOH, 2007). An example of collusion would be a team-based game where a 'traitor' player sends secret information to the other team.

Information exposure cheats are an issue in RTS games, but they are not a significant threat against MMORPG games (KABUS et al., 2005). We extend this to state that all avatar-based MMOGs, including both MMORPGs and MMOFPSs, should have limited impact from information exposure cheats. We back this assertion with Laurens et al.'s work on wall-hack detection on small-scale FPS games (LAURENS et al., 2007). Additionally, a MMOFPS would benefit from much longer player logs, if compared to a small-scale FPS MOG, before having to reach a decision of whether a player is seeing through walls or not. And finally, information exposure cheats cause bounded (limited) damage against MMOGs. Thus, we disregard information exposure cheats for FreeMMG 2, which is geared towards avatar-based MMOGs. We leave such cheats to be mitigated by the application or by future work.

[22] Kabus et al. (KABUS et al., 2005) describe the collusion cheat in the context of peer-to-peer games, but they don't name it.

### 2.5.5 Time cheats

As we have hinted at previously, online game experience factors such as consistency and fairness can be maximized for all players involved if clients inform the server about the time in which their actions occur. For instance, the lag compensation mechanism of Half-Life for instant-hit weapons allows clients to hit delayed targets on their machines by informing the server about the time in which their actions occur. Also in the Half-Life protocol, clients are allowed to inform, for each command, a quantity of simulation time elapsed locally for the command (the frame duration), which is also a kind of time information.

The problem with using time-related parameters sampled at clients is that clients can lie about them to the server in order to cheat. This also goes for pure peer-to-peer architectures, where all nodes are client (read: untrustworthy) nodes, and they can lie about time-related information to each other and exploit weaknesses in distributed consistency protocols that depend on a client-side source of time.

Cronin et al. (CRONIN; FILSTRUP; JAMIN, 2003) define time cheats as cheats that *'give the cheater an unfair advantage by allowing it to see into the future, giving the cheater additional time to react to the other player's moves'*. Cronin et al. addresses a common time cheat which is the *look-ahead cheat* for peer-to-peer games. Other works address what they call the *timestamp cheat* (WEBB; SOH, 2007). Though there are other time cheats such as the suppress-correct cheat (CRONIN; FILSTRUP; JAMIN, 2003) and the fixed-delay cheat (GAUTHIERDICKEY et al., 2004a), we will focus on the look-ahead and timestamp cheats only, which we describe below.

#### *2.5.5.1 Look-ahead and timestamp cheats*

The look-ahead cheat consist of waiting for other players' moves before committing your own, as in 'peeking into the future' to see what the opponents will do so that you can make a better decision in the present. In a peer-to-peer, round-based protocol, where player moves or commands are ordered with a sequence number – the round number – instead of with specific time values sampled from machine clocks (timestamps), the look-ahead cheat is implemented simply by waiting for other players' moves for the current turn, optimizing your decision based on them, and then sending out your own (cheated) move.

In a timestamp protocol, where players may act at any time by sending a command with a simulation time value, the look-ahead cheat is implemented by simply waiting for an opponent's command and then submitting a counter-move with an earlier timestamp. In this case, it is called a *timestamp cheat*. This cheat is possible in such protocols because they have to be coupled with some mechanism to re-execute straggler (late) events in the correct order or to compensate for them. Chen and Maheswaran (DI CHEN; MAHESWARAN, 2004) offer more details on timestamp cheating.

So, look-ahead and timestamp cheats are, in essence, the same cheat. In both cases, the cheater gains an advantage over opponents due to its ability to commit its action after knowing about other actions that should, but aren't, simultaneous to his own, and thus increasing his reaction time or fabricating a reaction time that didn't exist (in the case of moves that should be simultaneous). The difference is that look-ahead cheats are committed against round-based protocols and timestamp cheats are committed against proper timestamp-based protocols.

There are several works that address the look-ahead cheat for peer-to-peer proto-

cols. The Asynchronous Synchronization (AS) protocol (BAUGHMAN; LIBERATORE; LEVINE, 2007) is an hybrid solution that forces players to synchronize in a round-based fashion whenever their avatars are close enough, and otherwise allows asynchronous (unordered) execution of commands by unrelated (distant) avatars. Whenever executing in rounds, AS forces all players to cypher their moves for each turn and provide the key for reading it on the next turn. This prevents players from using the knowledge of other player's actions since the key (the next turn) is only sent by the owner of the move when it has received all cyphered moves for the current turn from all other players (BAUGHMAN; LIBERATORE; LEVINE, 2007). This basic principle of cyphering moves to prevent look-ahead cheats is employed by other protocols such as NEO (GAUTHIERDICKEY et al., 2004a) and SEA (CORMAN et al., 2006). The Sliding Pipeline (SP) protocol (CRONIN; FILSTRUP; JAMIN, 2003) is an attempt at relaxing the rigidness imposed by the round-based nature of that solution. In SP, missing rounds can be dead-reckoned (extrapolated) and timestamp cheating is prevented by monitoring player delays and trying to determine the maximum allowable delay for a late command (WEBB; SOH, 2007). The main critique of these solutions is that they can impose large delays that would be otherwise avoided by the simpler (and vulnerable) protocols (WEBB; SOH, 2007).

The Referee Anti-Cheat Scheme (RACS) (WEBB et al., 2007) requires a trusted referee which performs sensitive tasks such as assigning events generated by peers (players) a 'round' to execute in all peers. The referee in RACS keeps most game traffic peer-to-peer and it avoids several known cheats, including the timestamp (look-ahead) cheat. The time cheat is avoided by scheduling late events to the *next* round, where the *current* round is whatever the referee considers as current. In other words, timestamping occurs at the receiver which, in this case, is a centralized, trusted party. Thus the cheat is avoided by trading off by departing from a pure peer-to-peer solution. In this context, the authors of RACS followed their work with ways to elect trusted referees out of a pool of anonymous peers (WEBB; SOH; TRAHAN, 2008). This path is being pursued by other works on reputation management for peer-to-peer games (HUANG; HU; JIANG, 2008; SHI; GAO; DU, 2008; SHI et al., 2007).

There are works that attempt to solve the look-ahead cheat (or timestamp cheat) without resorting to a round-based protocol. The AC/DC algorithm (FERRETTI; ROCCETTI, 2006) avoids forcing the simulation into rounds and allows events to be reported with any timestamp. It detects time cheats after they occur by, first, deciding if a node is a potential cheater – that is, it exhibits patterns that could be due either to network congestion or to false client-side timestamping. After that, the server (or remote peer) artificially inflates its own timestamps to see if the client will do the same to 'keep up'. In other words, the server (or peer) 'cheats' on purpose now and then to detect a remote logic that is modifying its own timing to maintain a constant reaction time advantage.

#### *2.5.5.2 Our stance on peer-to-peer MMOGs and time cheats*

The general principle when designing an online game protocol is to never trust the client since it is in the hands of the enemy. However, sometimes it is necessary to do so, and the case of client-side timing information is emblematic. There is so much to gain in fairness, consistency and responsiveness from using peer latency measures and input device read times, and there are so many techniques that use that information, that it is difficult to conceive an online game protocol that doesn't use client-side timing. In addition, there are already several works that counter time cheats. And finally, what

cheaters gain with time cheats, at least before their cheating attempts are detected, is of limited impact for MOGs and MMOGs – time cheats cause *bounded damage*.

In FreeMMG 2 we leave protocol design partially to the application. The FreeMMG 2 protocol provides some basic services such as state replication, overlay construction and routing, and event dissemination, but it doesn't go as far as specifying the actual meaning of events, which are left for the application. The interaction protocol – what clients request of the MMOG 'server' with their command messages – can thus be anything discussed in Section 2.4, with a dumb terminal client or with client-side prediction, lag compensation, etc. Thus, the peer-to-peer overlay that manages game state in a FreeMMG 2 game network (the 'server') can either use timing information from playing clients or not, depending on the game developer's choice. In case it does, any cheating arising from that will cause bounded damage, and existing anti-time cheating work can be applied by the application developer to counter these time-based cheats.

### 2.5.6 State cheats

As far as we can tell, the term *state cheat* is not present in the literature. Its characteristics are described in related works and it is definitely recognized as an issue, but it is not named in any consistent fashion[23]. Therefore, the name *state cheat* and the definition we present of it are ours[24].

A state cheat consists of a cheating player rewriting a significant portion of the authoritative game state. Emphasis should be placed in *significant*: the cheater must be able to cause *unbounded damage* to the game state, as we have discussed previously. Exploiting server-side bugs or race conditions (MCGRAW; HOGLUND, 2007) will only cause damage proportional to the bug in a limited window of time and thus do not qualify. Also, we consider that sending fake absolute position updates to a server or a peer also doesn't qualify as most MMOG economies cannot be exploited purely by position cheating (also known as *telehacks*) for the same reason that time cheats cannot. Thus, in a client-server MOG or MMOG, since the authoritative game state is kept at a trusted and secure server machine, state cheating is only practical if performed by the server administrator itself, not the players.

#### *2.5.6.1 Example: state cheat against a peer-to-peer MMOG with partitioned state*

Figure 2.19 depicts a state cheating scenario. The figure shows the global game state of a MMOG partitioned among four peers. This is one kind of *cell-based MMOG architecture*, since the game world's virtual space is divided into contiguous *cells*. Adjacent cells are supposed to connect transparently so that players should not, ideally, be able to perceive cell borders during play. The figure does not assume anything from the peers other than them being client resources, that is, machines not controlled by a central game operator and thus untrustworthy. The peers in the figure could either be members of a peer-to-peer overlay dedicated only to running the virtual world simulation, or they could

[23] We have erroneously named it *collusion cheat* in previous work (CECIN et al., 2004, 2006), but the term 'collusion cheat' is being used to denote a kind of information exposure cheat as seen in Section 2.5.4. What we meant in our previous work, and which still holds for FreeMMG 2, is that state cheats can be performed against replication-based peer-to-peer MMOGs when a significant portion of the assigned replicas *collude* to perform illegal state alterations in concert to their part of the game world's state.

[24] What we mean here is that our definition of the state cheat has no definitive backing in the literature and thus should have its accompanying argumentation examined by the reader, instead of being dismissed as established (background) information.

be also the players of the game. In the current context, this is not important and can be abstracted away.

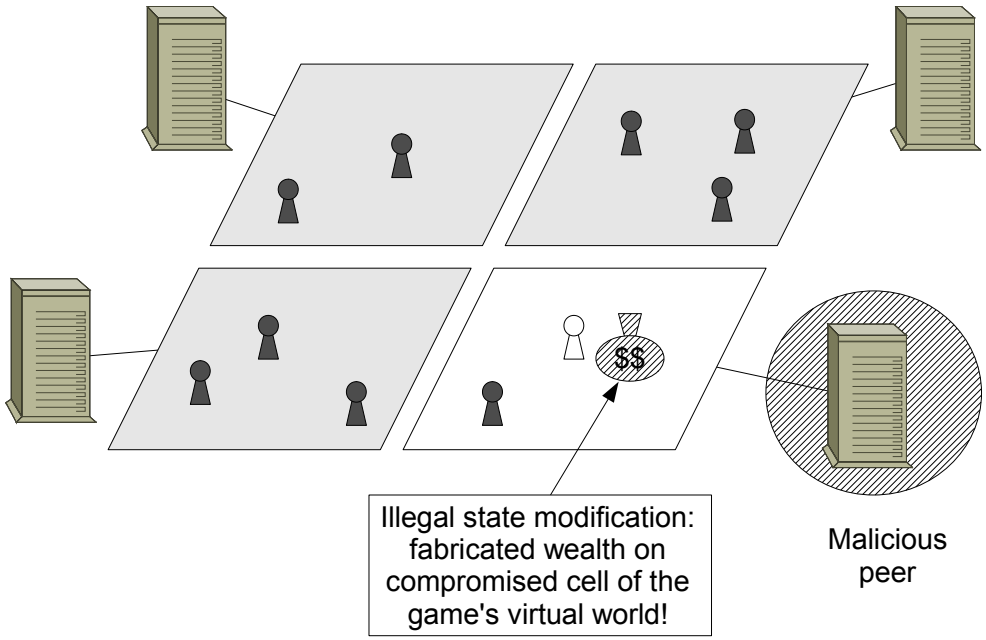


Figure 2.19: Simplified example of a state cheat being performed in a peer-to-peer MMOG architecture where the game state is partitioned among four peers.

In Figure 2.19, one of the peers to which part of the virtual world (a cell) is delegated exclusively is malicious. That peer decides to favor the white avatar in the figure and, to this end, instantly modifies its own local memory to include a substantial amount of game wealth next to the white avatar. In most MMORPGs it is common to encounter objects such as 'piles of gold' that can be picked up by avatars next to them. Such objects usually have a numeric attribute that designates the amount of gold in the pile in question. Thus, a state cheat by a malicious peer could be executed simply by inserting one such pile into the game state that has an arbitrarily high 'amount' value, one that, for instance, is higher than the total amount of circulation in the entire game economy. The problem becomes apparent if we consider that the white avatar can pick up that wealth and move to other non-compromised cells carrying it – an unregulated amount of game wealth can ruin the most well-engineered MMOG economy (THOMPSON, 2000). Thus, we show that this simple cheat can cause an *unbounded amount of damage* to the game state.

While for a particular game it might be easy to establish a ceiling on the wealth that an avatar may carry into a non-compromised cell, this is difficult to solve in any generic way. Also, this is by far not the only example possible. The malicious peer can also cause other avatars that it happens to dislike to die instantly, or it can 'level up' (make permanently stronger) friendly avatars. These examples are already much more difficult to detect by other cell peers and to compensate for, even for a solution that is specific to one well-crafted game design. Each game would require its own checks and it would become hard to detect cheaters that smuggle fabricate wealth to non-compromised cells in a rate that is just below the detection threshold – that it knows about precisely since it should be running the same logic.

#### 2.5.6.2 *Impact of state cheats against MOGs and MMOGs*

State cheats can be at least as serious as other forms of cheating in small-scale MOGs, be them peer-to-peer or client-server. However, in a non-massive MOG that doesn't have persistent state, the damage is bounded by the limited duration of each instance or match of the game. Moreover, the player can always switch servers or peer groups freely in a non-massive game; there is no long-term game world being abandoned.

However, we assert that state cheats are promoted to a different, higher degree of seriousness when we consider *peer-to-peer MMOGs* specifically. A peer-to-peer MMOG, by definition, is relying on what are, by default, *untrustworthy and anonymous client machines for storing parts of a long-term game state*. This is, again by default, the same as letting clients store part of the game state or arbitrate over the game state in a client-server MMOG. We say *by default* because the research trend on peer-to-peer MMOGs was, at first, to just partition the game state among peer machines, letting each peer arbitrate alone over some part of the game state. These solutions, as demonstrated by the earlier example, are very vulnerable to state cheats since any peer can change the contents of its part of the game world arbitrarily.

#### 2.5.6.3 *Dealing with state cheats in peer-to-peer MMOGs*

Later in this thesis, we review other works that propose peer-to-peer MMOG architectures, and we classify them as either addressing state cheats or not. For these architectures, we consider the following to be general valid directions for addressing state cheats:

- *Game design:* The game is somehow immune or at least not fatally affected by the ability of peers that manage game state to modify it arbitrarily;

- *Detection with undo:* The architecture is able to eventually detect state cheats and recover fully from them or at least significantly mitigate their effects;

- *Game design and detection without undo:* The game is not fatally affected by the ability of peers that manage game state to modify it arbitrarily. Additionally, cheaters are eventually detected and banned in a way that state cheats are always bounded by the detection interval (this replaces the undo mechanism). And finally, cheaters are somehow prevented from creating new identities without cost to start cheating immediately after being banned;

- *Bounded state modification:* The total amount of in-game wealth (game-specific) in each part of the game state is somehow bounded. This would apply to solution that verify the amount of wealth over time in each part of the game world and ban peers that are enriching their game state partitions in a manner incompatible with game rules. Another way to tackle this would be through use of cryptography to restrict the minting of in-game items to a trusted party such as in the PSMMO model (CHAMBERS; FENG; FENG, 2006);

- *Trust or reputation management:* trustworthy nodes are somehow correctly elected among peers that are candidates for serving in the peer-to-peer overlay as keepers of authoritative game state. An example of such solution would be one that required a community of 'peer server' administrators that implicitly trust each other to make up the peer-to-peer MMOG overlay. Other solutions would involve reputation management or web-of-trust models for electing trustworthy peers (HUANG;

HU; JIANG, 2008; ZHOU; HWANG, 2007; KAMVAR; SCHLOSSER; GARCIA-MOLINA, 2003);

- *Replication:* The architecture replicates each part of the game state that is delegated to client machines and employs some form of byzantine or quorum-based consensus to filter out deviant versions of that part of the game state. This is the approach of FreeMMG 2 and several others.

These are the ways that we know of and consider as viable solutions to the problem of state cheating against peer-to-peer MMOGs. This list is derived from our current review of the literature, and future works could certainly come up with new ways to address the issue of state cheats. For instance, Kabus et al. (KABUS et al., 2005) point out that a working Trusted Computing (TC) platform could be used as a cheat deterrent.

### 2.5.7 Closing remarks

The persisted game state of MMORPGs such as World of Warcraft contains significant value as perceived by players of these games (MANNINEN; KUJANPÄÄ, 2007). Since there is perceived value in MMOG state and transfer of state items is possible between player avatars, the fact that players of MMOGs trade items between themselves for real money (GUO; BARNES, 2007) is a natural and an ultimately unavoidable consequence, though a significant portion of these players protest against this so-called Real Money Trading (RMT) activity in MMOGs (YOON, 2008; LEHDONVIRTA, 2005). The significant volume of RMT in MMOGs, which is a way of living for some (DIBBELL, 2007), has even caused governments to start pondering whether this commerce of virtual items should be taxed (CHUNG, 2008). As a reference, the worldwide volume of RMT transactions has been estimated to be around US$ 2 billion in 2007 (LEHTINIEMI; LEHDONVIRTA, 2007).

As of now it is unclear if peer-to-peer MMOG architectures, if ever deployed, will face the same issues of law, property, taxation and involvement with real money that client-server MMOG services currently face. Nevertheless, as long as MMOG models based on peer-to-peer are intended to contribute in this direction of large-scale deployment, they should consider these issues.

In this context, it becomes clear that securing game state becomes important. In the real world, it is certainly possible to defraud the systems of work and reward by e.g. robbery, but creation of concrete wealth such as goods or money are either limited by physical laws or limited to centralized (trusted) entities such as central banks which issue 'real' currency. The mechanisms that protect against outright fabrication of wealth in the real world are much stronger than the mechanisms, such as policing and courts, that protect against spurious violations (oftentimes with false positives). In virtual worlds, we assert that this distinction also holds, and thus we have differentiated state cheating, which is arbitrary fabrication of wealth analogous to owning a bank or being able to materialize goods without effort, from other types of cheats that accelerate the acquisition of wealth at a bounded pace.

And finally, this explains our design decisions in FreeMMG 2. Integrating resistance against state cheating at a fundamental level in the architecture was thus one of the top-level priorities, and several trade-offs were made to meet this demand. And, to avoid making too much of a concession in terms of scalability, latency and bandwidth efficiency, we have left other types of cheats to be addressed by the application, which will then be responsible for assessing the cost of the additional anti-cheat solutions employed, if any.

## 2.6 Network-level attacks versus peer-to-peer MMOGs

Alongside state cheats, one of our top-level concerns is with network-level attacks, such as a Denial-of-Service attack (DoS) or a Distributed DoS (DDoS), against selected nodes in the peer-to-peer overlay that is to maintain the game state. Network-level attacks are more of an issue against peer-to-peer MMOGs than against client-server MMOGs. The reason is that, in the case of a network attack against MMOG servers, the server becomes unresponsive or dies, but once the attack ceases, it is certain that clients will keep trusting the game state backed up by the server. Moreover, we can assume that the internal network used by servers to synchronize between themselves is unaffected by external network attacks. On the other hand, peer-to-peer MMOG architectures, which are distributed computing systems, run into trouble when dealing with *partial failure* and the lack of a *central coordinator* to recover from such failures, which are major issues in distributed computing systems design (WALDO et al., 1997).

Though any peer-to-peer system, MMOG overlays included, have to automatically deal with regular *peer churn* (STUTZBACH; REJAIE, 2006), the issue of network-level attacks is more serious – it can be thought of as a form of directed and intentional (selective) peer churn. As an example, consider the following scenario: a peer-to-peer MMOG replicates parts (cells) of the virtual world among several, randomly-selected anonymous peers to make it resilient against state cheating. Let's assume that each cell is replicated in eight nodes. Now, let's assume that one of these peers is malicious, and that it has at its disposal a botnet (RAJAB et al., 2006; COOKE; JAHANIAN; MCPHERSON, 2005) of sufficient size that then disables the remaining seven replicas from sending and receiving packets. Unless otherwise noted by the architecture's description, we should assume that there is now a single copy of that world cell being managed by a malicious peer that is now free to arbitrarily change the state of the cell.

In the above scenario, the honest game clients (player machines) and any server-side, trusted supervisors are prevented from synchronizing with the honest cell replicas that were taken down[25]. Now, one solution would be for the players or server-side coordinators to *wait* for the honest replicas to come back. This is not practical since that can take a long time and the game must proceed in real-time, and also there is no guarantee that the peers will ever return to the system, causing the entire cell state to be permanently lost. Another solution would be to replace the missing replicas with fresh ones, but those would feed into whatever data is reported by the remaining (malicious) replica and thus the window for state cheating remains.

We know that *preventing* a network-level attack such as a DDoS is impossible. Some service outage is to be expected from it; this is the price of considering peers with limited bandwidth as components of a distributed MMOG service. However, we do believe that reducing the likelihood of state loss due to regular churn and network attacks is of paramount importance for the same reasons that cheating is to be prevented. That is, a MMOG's state has value and it should be protected from illegal alterations and loss. Chapter 4 shows how we address network-level attacks in FreeMMG 2 by employing secret back-up replicas in each cell.

[25]This kind of 'cut-off' attack against peer-to-peer overlays in general is known as the *Sybil attack* (DOUCEUR, 2002).

## 2.7 Peer-to-peer MMOG support

In 2001, Fitch (FITCH, 2001) stated that *'cyberspace is simply the logical evolution of peer to peer systems'*, where MMOGs are one possible instantiation of the cyberspace concept. Since around that time, a body of research work around peer-to-peer support for MMOGs started to form. Guo and Norden (NORDEN; GUO, 2007) stated that *'P2P overlays are a natural fit for MMOGs, owing to their scalability and distributed nature of processing'*.

Schiele et al., in *Requirements of peer-to-peer-based massively multiplayer online gaming* (SCHIELE et al., 2007), have identified ten key qualities that must or should be provided by peer-to-peer MMOG solutions:

- Distribution: Schiele et al. argue that peer-to-peer MMOGs are completely decentralized systems and thus require both *distributed data management* and *distributed computation*, and that this decentralization (Distribution) requirement is what primarily differentiates them from client-server MMOGs. We increment this definition by considering that hybrid MMOG distribution models, which combine client-server and peer-to-peer, should also be classified as peer-to-peer MMOG models that trade-off Distribution for some other requirement;

- Consistency: as we have focused on earlier in Section 2.3, once a distributed simulation scenario is considered, where the state of the simulation is to be kept by different LPs communicating asynchronously with each other, the issue of maintaining the global simulation state consistent arises. In addition, even in centralized DVEs, there is the issue of maintaining temporal consistency between a central simulator and the consumers of its authoritative updates, as seen in Section 2.4. Thus, maintaining consistency in a peer-to-peer MMOG is a challenge and, as noted by Schiele et al., some compromise in this area is inevitable. The main tools for managing consistency are controlling network latency and peer bandwidth requirements. Narrowing the model to support a specific MMOG genre (RPG, FPS, RTS or other) is a common strategy when designing for consistency. Other effective strategy is limiting the geographical coverage or the supported network distance between any two peers;

- Self-Organization: as in any peer-to-peer system, a peer-to-peer MMOG cannot rely on any live peer to be available in the future, thus game state or processing tasks must be either replicated in advance or compensated for when the peer that hosts them becomes suddenly unreachable[26]. It also cannot cannot rely on a particular peer's resources to be continually available, thus either dynamically balancing load between peers becomes a necessity or the system must be able to dynamically scale down running tasks (e.g.: graceful degradation of services) to adjust for the changing capacity;

- Persistency: Schiele et al. note that, in contrast to the typical peer-to-peer file sharing network, content (game state items, or the value of game state variables) cannot simply disappear from the system once the last peer that has it leaves the system. They also note that state evolution and storage should be independent of

[26] An alternative is to design for super-peers (YANG; GARCIA-MOLINA, 2003). A system based on super-peers faces the same trust and security issues of other decentralized systems, but each node of such network is assumed to have the same resilience of a server node in a server-based system.

the time that player machines stay connected – what is easily achieved by server-centric models. To satisfy the persistency requirement, peer-to-peer MMOG models either employ replication or employ servers or super-peers (churn-free peers) for persistence, resulting in a partially centralized (hybrid) architecture.

- Availability: the game service must be available for players for a maximum amount of time. For instance, a peer-to-peer MMOG where departing peers routinely take parts of the virtual world's state with them on disconnections is a solution with low availability. Also, solutions which require large amounts of volunteer resources may run out of peers to provide the service. The latter is an availability issue in our model (FreeMMG 2) and is one of our main trade-offs (discussed later in this thesis);

- Interactivity: referred in this text also as *responsivity*, this is a constraint on the interaction latency imposed by the architecture: the amount of time between a human player issuing a command and perceiving its effects on its own machine. Like with consistency, managing interactivity is mostly about dealing with network issues such as delay, bandwidth and jitter. And, solutions that increase interactivity (or responsivity) may decrease consistency (ROBERTS; WOLFF, 2004), and vice-versa.

- Scalability: Though peer-to-peer is touted as being more scalable than client-server, this cannot be taken for granted (SCHIELE et al., 2007). The idea of peer-to-peer, in its way to becoming a concrete solution, may result in a system that scales worse than the client-server alternative. For instance, a fully connected network (full mesh network) is peer-to-peer but is also not scalable. By definition, a massive gaming architecture is scalable, since it must support thousands of simultaneous users to be called a MMOG. However, some architectures may scale better than others, that is, degrade more gracefully than others as the amount of nodes in the system grows;

- Security: besides the phenomenon of player cheating, which currently dominates security concerns in virtual worlds, MMOGs can present other security-related challenges such as offensive language, monetary fraud, identity theft, virtual crime and others (BARDZELL et al., 2007; KU et al., 2007; ALEMI, 2007; BAROSSO et al., 2008; WHITE, 2008). Peer-to-peer MMOG architectures are, however, still not mature enough to be able to afford tackling these problems. Also, by addressing cheating first, the amount of 'virtual crime' that will need investigation later will be naturally reduced;

- Efficiency: Schiele et al. note that, by departing from the client-server paradigm, MMOGs must cope with limited CPU, memory and bandwidth resources typical of player-owned nodes to not only render the game to the player, but to run the simulation tasks that were being performed by the servers. In a server-based MMOG, this can be solved by adding more servers, an option generally not available in a peer-to-peer scenario. Thus, the architecture must make efficient use of the available peer resources;

- Maintainability: some peer-to-peer MMOG models may impose more maintenance burden than others. Schiele et al. note that cheat-proof solutions that depend on cheat detection mechanisms may need such mechanisms to be regularly updated

to counter new cheat techniques that avoid detection. Also, we remind that hybrid client-server and peer-to-peer architectures present the burden of maintaining the server-side infrastructure, which is absent from a pure peer-to-peer approach.

Considering the current Internet infrastructure, it is hard to design a peer-to-peer MMOG model that flawlessly passes all these requirement checks. Aiming for such a design that should, in addition, work for any conceivable MMOG design or deployment scenario is infeasible. Thus, each MMOG support model will be based on a particular approach to meeting MMOG support requirements, with its own particular requirement trade-offs, supported game designs and specific deployment scenarios envisioned.

In this section, we review other proposed peer-to-peer architectures for MMOG support, presenting the key features of each. With the following review work, we intend to show that there is a shortage of peer-to-peer MMOG architectures that adequately address both the problems of *state cheating* and *network-level attacks*, which are particular MMOG security requirements that we have raised as top priority due to reasons discussed in earlier sections. By focusing on protection against state cheating and resilience to network-level attacks, we can quickly establish the originality of FreeMMG 2 among the bulk of the literature. In Chapter 6 we perform a more detailed comparison with selected works which measure better against FreeMMG 2 under our own security-centric approach, with the other requirements being used more often as evaluation criteria.

The sub-sections below separate architectures in a few broad categories: player mesh, single arbiter, multiple arbiter, and peer-to-peer with fat server-side. These categories correspond roughly to how we view other works in light of our specific motivation. After that, we review other works that don't fall into the previous categories under the *other models* sub-section.

### 2.7.1 Player mesh

Peer-to-peer MMOG support models of the *player mesh* type focus on establishing connections directly between player machines and letting players exchange events related to their avatars with each other directly. Player mesh contrasts with *cell-based* architectures, where the virtual world terrain is split into cells, and the overlay, weaved with playing (interactive) and non-playing (*daemon* volunteer) peer nodes, is designed mainly to perform management of avatars and other game objects within each cell.

Player mesh architectures often result in the game state being limited to being a collection of avatars, where each avatar's state is, generally, to be controlled (arbitrated) by the respective player machine. Players establish point-to-point socket connections to one another and exchange events (commands or updates) directly between each other whenever their respective avatars are close to each other in the virtual world. Thus, connectivity in the peer-to-peer overlay changes as avatars move in the virtual world. Cell-based architectures, on the other hand, are more stable in this regard.

Yu and Vuong (YU; VUONG, 2005) refer to this type of architecture as *unstructured P2P approaches* in contrast to structured network overlays such as DHTs (DARLAGIANNIS; HECKMANN; STEINMETZ, 2006). They have also identified three major proposals that fitted into this category: Kawahara et al.'s *A Peer to Peer Message Exchange Scheme* (KAWAHARA; AOYAMA; MORIKAWA, 2004), Keller et al.'s *Solipsis* (KELLER; SIMON, 2003, 2002) and Hu et al.'s VAST (Voronoi-based Adaptive Scalable Transfer) (HU; LIAO, 2004). Voronoi-based approaches have been further developed by recent works (GENOVALI; RICCI, 2009; JIANG; CHIOU; HU, 2007; BACKHAUS;

KRAUSE, 2007) which tackle pending issues such as maintaining global connectivity at all times.

However, we see two main problems with player mesh approaches that still persist:

- It is likely that each player will hold part of the game state without supervision from other peers. These works often focus on messaging scalability, interaction timeliness and efficiency, and they revolve around the details on how players can send events to interested peers directly with low latency and low overlay management overhead. Though this is not generally discussed, we can only assume that these outbound events are authoritative updates. In other words, each player will be able to decide the state of its own avatar. As discussed earlier, this is highly problematic due to cheating in a competitive MMOG. We haven't seen a player mesh approach that attempts to address this;

- Since avatar management is the focus, it is generally unclear who will be responsible for non-avatar objects such as non-player characters and other dynamic objects which aren't part of the static game terrain. We haven't seen a player mesh approach that addresses this in a satisfactory way, such as replicating avatar and non-avatar objects;

Thus, though scalability, bandwidth efficiency, consistency and interactivity are maximized, these works in their current maturity result in desert-like, static virtual worlds (TYCHSEN; HITCHENS, 2006) where avatars can notify each other about their arbitrary, unsupervised state. Thus, it becomes difficult to visualize such models supporting a MMORPG with virtual economies and a persisted game state that can evolve in meaningful, complex ways through years of play.

These schemes could be made to support virtual economies if the game objects were somehow partitioned among players. However, the fact that these overlays are designed to match proximity of avatars in the virtual world makes matters much worse for security solutions. That is because peers that would be supervising each other or replicating data together would be suddenly connecting and disconnecting due to avatar movement.

This would not be a problem if these overlays were supposed to be just communication substrates for a distributed state replication scheme, which is what structured peer-to-peer overlays (such as DHTs) provide. In the case of a DHT, the overlay would provide a virtual address where objects can be stored. Each game object such as an NPC would hash to a key, and that key would route to a peer which would be responsible for that game object. Security could then be achieved by using DHT replication meachanisms (KTARI et al., 2007). However, this only moves the security problem to the DHT layer. Securing the DHTs themselves is an active research topic (SANCHEZ-ARTIGAS; LOPEZ; SKARMETA, 2008; DANEZIS et al., 2005; SRIVATSA; LIU, 2004).

An unique model that seems to also fit into the 'player mesh' category is Ahmed and Shirmohammadi's convex hull AoI (AHMED; SHIRMOHAMMADI, 2008). Their idea is to create a peer-to-peer network dedicated for each cluster of nearby avatars. So, whenever two packs of avatars approach each other, the underlying connection overlays merge, and when a pack of avatar splits in two, the overlay likewise splits. To detect (calculate) a group of avatars, they employ a well-known algorithm that attempts to fit avatars inside convex hulls, where candidates for vertices of the hull are player avatars at their current positions. This zoneless (not cell-based) solution does not require any synchronization between the player groups. This contrasts with most zone-based (cell-based)

models, which typically require cells (zones) to synchronize with each other. However, the security concerns raised earlier also apply to this model.

Finally, Krause (KRAUSE, 2008) compares *mutual notification* (avatar AoI-based, player mesh approach) with ALM (Application Layer Multicast) protocols[27] and super-node protocols[28]. Though a case is made that player mesh performs better in many aspects such as responsivity and bandwidth efficiency, the problem of cheating is ignored.

### 2.7.2 Peer-to-peer with fat server-side

Several peer-to-peer MMOG models are hybrids which depend on some trusted server machines to perform some tasks, but which also lets clients participate in the simulation instead of relegating them to the role of consumers of simulation results computed by the servers. The more tasks are delegated from servers to clients, the more decentralized the model is, and smaller are the CPU and communication costs at the server-side infrastructure.

We consider *fat server-side* models to be any hybrid MMOG proposals which do not perform enough decentralization. Though these generally have a smaller server-side footprint if compared with a pure client-server approach, the reduction is not sufficient. We don't compare these models with FreeMMG 2 later on Chapter 6 because the comparison wouldn't be fair: it is much easier to solve several problems in MMOG support if significant computing and communication resources are available at reliable machines. For instance, we have previously extended FreeMMG to require receiving all player commands, and this allowed us to achieve immunity against state cheats (CECIN; BARBOSA; GEYER, 2005). Also, this goes against one of our core motivations in using peer-to-peer for MMOG support, which is to lower the barrier of entry to MMOG hosting.

In the list below we review a selection of such models: the ones which delegate some network or CPU cost to client machines. We point out what causes them to be framed as 'fat server' solutions, in spite of their partial CPU and communication decentralization efforts:

- ACORN (NORDEN; GUO, 2007) is a model that was designed to run on top of a DHT such as Pastry or Chord. It mitigates DoS (network) attacks and state cheating (called *coordinator compromise* which they regard as being *'akin to the server cheating (in a client-server architecture)'*) by moving the coordinator functionality around the DHT frequently and in unpredictable ways. Also, though they use a DHT, they recognize that DHTs must have their routing secured in some way to be used. We see two main problems with ACORN. First, all player nodes have to send all of their commands, in real time, to a CAP (Coordinator Access Point) which is a server-side process (similar to the PP-CA architecture (PELLEGRINO; DOVROLIS, 2003)), resulting in a relatively significant communications cost for the game operator. And second, coordinators are still unsupervised: any node, upon detecting to be a coordinator, can modify the managed state to anything it pleases. However, they do provide a cheat detection and a cheat undo mechanism (based on checkpointing) for this kind of cheat. These mechanisms work by executing all player commands at the trusted CAP and verifying the outcome;

[27] Application Layer Multicast (ALM) is any technique that compensates for the absence of IP multicast support. Basically, it means emulating IP multicast through a logical connection overlay, such as in a DHT.

[28] *Super-node protocol* here means the same as *single arbiter*, which is an approach that we'll look into later. The use of *super-node* here is unrelated with the capacity or reliability of the nodes in question.

- RACS (Referee Anti-Cheat Scheme) (WEBB et al., 2007) is another extension of the PP-CA architecture by Pellegrino et al. (PELLEGRINO; DOVROLIS, 2003). It provides protection to a wide array of cheats such as information exposure cheats, suppress-correct cheat, fixed delay cheat, inconsistency cheat (a player sending two commands with same timestamp and different content to different peers), timestamp cheat and others. The authors note that it also integrates easily with other anti-cheat schemes such as bot detectors. It provides two modes of operation, and both require receiving and executing all player commands at the server-side. This results in the CPU cost at the servers being the same as in a client-server setting since the server must replay the whole virtual world simulation. It also results in a somewhat significant client-server communication cost, much like in ACORN (NORDEN; GUO, 2007) and other extensions of the PP-CA model to massive scale;

- Ito et al. (ITO et al., 2006) leave the whole virtual world simulation to be performed by servers, resulting in the same CPU cost at the server-side as in a client-server solution. However, they use ALM to significantly reduce server communication. Instead of having the servers communicate directly with each client, servers communicate with reduced number of player-provided proxy nodes which, in turn, communicate with player machines. Thus, server world update messages travel multiple hops through a multicast tree (the overlay) in order to reach the players, thus increasing server-to-client communication latency to save server bandwidth. To keep latency acceptable and avoid impact in responsivity, the maximum height of the multicast trees are always three: servers (level 1, roots), proxy helpers (level 2, intermediaries) and player machines (level 3, leaves). The system elects player machines to act as additional proxies as more bandwidth is required. Vik et al. (VIK; GRIWODZ; HALVORSEN, 2007, 2006; VIK, 2005) propose a similar system, but they approach ALM tree construction with Steiner Minimal Tree (SMT) or Minimum Spanning Tree (MST) algorithms instead of using a predefined tree structure;

- Rooney et al. (ROONEY; BAUER; DEYDIER, 2004) propose a *federated* architecture. They found out that, while both CPU and network (communication) cost at a client-server MMOG grow quadratically as the number of players increases, the predominant cost in the client-server architecture is the processing of game state (ROONEY; BAUER; DEYDIER, 2004; BAUER et al., 2004). Thus, they focus on reducing CPU cost by offloading it to client machines, but the full network cost is left to the game operator. They propose the use of *multicast reflector* processes at the server-side which are responsible for receiving unicast messages from players and multicasting these to all players interested in it. To achieve this, multicast reflectors may need to exchange packets, which is done using an unreliable protocol to reduce overhead. This is acceptable since they envision a scenario where multicast reflectors are allocated in several different administrative domains (thus forming the *federation* in question), all rented by the central game operator and all having sufficient bandwidth (dedicated hosting), thus making packet loss between reflectors unlikely. The idea is that the actual providers of hosting for the multicast reflectors will have economy-of-scale gains by selling such service to other MMOG operators or other clients of application hosting and thus achieving cost savings. Their model is currently not applicable to a scenario where the multicast reflectors have to be provided by home users with 'Internet leaf', consumer-grade broadband links;

- DaCAP (LIU; LO, 2008) is resistant to state cheating and achieves a significant level of decentralization. DaCAP divides the game world into cells, and cells can run in either peer-to-peer or client-server mode. The peer-to-peer mode is implemented by a full mesh (all-to-all) network between all peers, which are players *interested* (as in IM) in the cell. All clients (peers) in a cell supervise each other, and if any inconsistency is found, the server rejects integrating the update informed by the peer-to-peer cell, such as increasing an avatar's amount of virtual currency at the server-side character account. Additionally, random peers (non-interested) are allocated to each cell to prevent undetectable collusion groups that could perform state cheats. When the number of player peers interested in a cell exceeds some threshold, this 'hot-spot' cell is converted to client-server mode, being served by a H&R (Hotspot & Raid[29]) server, which is a trusted machine that synchronizes with players using some client-server game protocol as seen in Section 2.4. Since it is to be expected that, as the game world's population grows, the number of hotsposts will grow accordingly, this could result in a significant communication cost for the game provider, but also significantly lower than that of a pure client-server solution (LIU; LO, 2008). Finally, the DaCAP paper is short on details such as how to allow peers to join and leave a full replica simulation group dynamically, what is the exact protocol adopted for intra-cell synchronization in a P2P-mode cell, and how to allow for inter-cell avatar interaction.

Besides these works, there are others that retain the full processing and networking costs at the server-side, and thus do not match at all our motivation of cost reduction and lowering the barrier of entry to MMOG hosting. Below we list some of these:

- Peer-to-peer at the server-side: Rieche et al.'s *Peer-to-Peer-based Infrastructure Support for Massively Multiplayer Online Games* (RIECHE et al., 2007) employs a structured P2P overlay (a DHT) at the server-side. That is, the member nodes of the DHT are trusted server nodes that are all maintained by a central game operator. Their gain is in building a server infrastructure out of low-cost nodes instead of expensive machines, but otherwise the server infrastructure operator pays for the full CPU and networking costs;

- Load-balancing: Vleeschauwer et al. (VLEESCHAUWER et al., 2005) and Bossche et al. (VAN DEN BOSSCHE et al., 2006) divide the game world into *microcells* which are dynamically passed around server machines to regulate load. The idea is that once a large number of avatars moves to an area, the microcells of such area will have to be handled by several server machines, thus solving the *hot-spot* problem of virtual worlds. The *2Layer-Cell Method* (JANG; YOO, 2007) also solves hotspots by dividing the game world dynamically in a quad-tree fashion and moving the resulting smaller world pieces to different server machines as needed;

- MMOG development facilitation: Balan et al. (BALAN et al., 2005) see a problem with peer-to-peer approaches to MMOG support because they have to be *'intimately designed with the P2P network in mind'*, and thus they developed a scalable

[29] As the name implies, DaCAP's H&R server also serves *raids*. 'Raid' is another name for a private instance. These are insulated game spaces that are dynamically added to the game world when demanded by closed groups of players. Examples are World of Warcraft's raids and the PvP instances in Guild Wars. The difference between raid cells (or instanced spaces) and contiguous cells is that raid cells do not require inter-cell synchronization, thus being much simpler to support.

distributed system called Matrix, which isolates the MMOG developer from the intricacies of map partitioning, consistency and reliability mechanisms. Glinka et al. (GLINKA et al., 2008) and Gorlatch et al. (GORLATCH et al., 2008) propose the Real Time Framework (RTF), a middleware which cooperates with the game application layer on determining what goes into the 'main loop' of the game process. This contrasts with communication middleware, which leave the game loop and related tasks entirely to the application (maximum flexibility and more work to the developer), and full game engines, which take over the main loop and only call back the application when specific events occur (minimum flexibility and less work to the developer[30]). Finally, Project Darkstar is a *'research effort attempting to build a server-side infrastructure that will exploit the multithreaded, multicore chips being produced and scaled over a large group of machines while presenting the illusion that the game programmer is developing in a single-threaded, single-machine environment'* (WALDO, 2008);

- World state replication: In a mirrored-server model (CRONIN et al., 2004a), each MMOG server manages a full replica (copy) of the virtual world, and thus the servers mirror each other's full contents. The Enhanced Mirrored Server (EMS) architecture (WEBB; SOH; LAU, 2007) enhances Cronin et al.'s mirrored server (MS) architecture (CRONIN et al., 2004a) by optimizing client-to-mirror assignment (player latency reduction) and by reducing mirror processing (CPU) and bandwidth requirements, which are issues in MS and also in RACS (WEBB et al., 2007). The main issue we see in EMS is that its enhanced mirrors maintain replication through a Bucket Synchronization protocol (GAUTIER; DIOT, 1998a) which does not guarantee deterministic execution under arbitrary communication latencies (GAUTIER; DIOT, 1998a; CRONIN et al., 2004b)[31]. Finally, Ferreti et al. (FERRETTI; ROCCETTI; PALAZZI, 2007; FERRETTI et al., 2005) attempt to mostly minimize inter-mirror communication and maximize player command responsivity and fairness by employing Active Queue Management (AQM) techniques;

- World state partitioning (or ghost-based replication): Müller et al. (MÜLLER; GÖSSLING; GORLATCH, 2006; MULLER et al., 2004) distribute authoritative control of game objects among several *proxy servers* in such a way that each game object is *owned* by a single server. However, a *ghost* or cached copy of each object is kept at each server that is not the owner of that object. Thus, players can connect to the nearest proxy server, to minimize interaction latency, and any server is still able to inform the client about a fairly current state of any game object, regardless whether the object in question is owned by that proxy server or not. There is, of course, a synchronization protocol between the proxies that synchronizes object master copies and their ghosts. Assiotis and Tzanov (ASSIOTIS; TZANOV, 2006) propose a similar model which is capable of transferring ownership of objects between servers, and also allows servers to own regions (region locks) in addition to owning objects (object locks), which increases the granularity and reduces the object management overhead;

[30]Provided that the game developer doesn't want to achieve something unexpected by the developers of the engine he is using, which ties back into the inflexibility aspect of engines.

[31]It should be noted that the TSS protocol of MS also doesn't guarantee consistency between replicas at all times (CECIN et al., 2006). We will discuss this in detail when we present our tweak to TSS, called BSS, in Chaper 3.

In short, there are other key motivations adopted by peer-to-peer MMOG researchers beyond aggressive offloading of CPU and communication cost to outside of the game service provider. Between the pure client-server MMOGs such as World of Warcraft and fully decentralized peer-to-peer MMOGs, there is a large gap being explored by many hybrid-architecture works. Naturally, each work is able to offload different kinds of tasks to player or volunteer machines, and in different proportions, and some do not delegate any tasks to clients.

### 2.7.3 Single arbiter

We have classified as *single arbiter* all peer-to-peer MMOG models which are both decentralized and that can support MMOGs with virtual economies, but that are also significantly vulnerable to state cheating due to *letting single client-side machines arbitrate over state*. As we have discussed during our review on cheating, if a single anonymous client machine is left with a piece of the virtual world to arbitrate over, that machine is able to break the game's virtual economy.

We have split single arbiter models further into four groups. First, there are *single zone owner* models that let a single client machine be the sole coordinator[32] of a virtual world cell[33]. Second, there are *single object owner* models that let a single client machine to be the sole coordinator of one game object or set of objects. Third, there are *single master coordinator models* that employ multiple coordinators but one of them is fully authoritative over the others. And finally, there is SimMud (KNUTSSON et al., 2004) and a derived work which we classify as single arbiter for different reasons.

#### *2.7.3.1 Single zone owner models*

*Load-balancing for Peer-to-peer Networked Virtual Environment* (LEE; SUN, 2006) divides the game world into a grid of rectangular subspaces (cells) and assigns each to a *subserver*, which is a client-side machine. Each subserver knows in real-time the contents of each one of its eight neighbor subspaces since it actively synchronizes with each one of its eight neighbor subservers. Hot-spots are handled by allowing a subserver to optionally delegate its functions to *area servers* (also client-side machines), but otherwise the chosen subserver remains the final arbiter of that cell's state. The authors acknowledge that cheating occurs if the subserver is malicious and leave cheat-proofing for future work.

The Time Prisoners works (EL RHALIBI; MERABTI, 2005; MERABTI; EL RHALIBI, 2004) are among several works that form a tree of client nodes for each cell. The tree is constructed from player nodes that are interested in such cells. They mention shortly that the multiple members of such trees could perform checks against each other to prevent cheating. However, the way in which such trees are built is vulnerable to state cheating. This is because cells with no interested players have no coordinators and the first player to become interested in a cell becomes its sole coordinator until another player becomes interested. That window of time in which a player is the sole coordinator of a cell is enough for state cheating and virtual world economy-breaking to happen. Moreover, the Time Prisoners papers do not specify how the multiple coordinators would perform the checks against each other.

[32] Coordinators are also referred to as *owners*, *managers*, *servers*, *masters*, *controllers*, *arbiters*, etc.

[33] Cells, which are sections of contiguous virtual world land, are also referred to as *zones*, *regions* and other terms. Cell have many advantages, such as providing a way to partition game objects while exploiting intra-cell locality, a way to divide CPU and communication load, a way to shape an otherwise unstructured peer-to-peer overlay, an indirect way for peers to find each other, etc.

Using DHTs to MMOG event delivery or state partitioning seems to cause single-arbiter designs to emerge among peer-to-peer MMOG architectures. We suspect that this is due to the nature of DHTs: a resource (key) maps to exactly one 'nearest' node which can then conveniently become the holder of the master copy of that resource. The following works on this section all use DHTs to perform one or more core functions.

Iimura et al. (IIMURA; HAZEYAMA; KADOBAYASHI, 2004) does not prevent state cheating due to malicious *zone owners*, but rathers chooses to detect malicious zone owners and then punish them later. It does not describe how arbitrary modifications to game state mandated by malicious zone owners are to be undone or compensated for.

MOPAR (YU; VUONG, 2005) divides the game world into hexagonal cells and then employs a DHT that maps each cell to a *home node* – a hash over the cell ID is used as key. The home node, however, does not hold state: it is only a kind of directory service which can be used to locate the *master node* of a cell, which is its sole coordinator and the definitive arbiter for the cell's current state. The state cheating vulnerability is the same as the one in the Time Prisoners works (EL RHALIBI; MERABTI, 2005; MERABTI; EL RHALIBI, 2004): the first player to become interested in a cell is assigned the role of *master node*. Finally, MOPAR does not have a mechanism to guarantee long-term state persistence of cell data, since once the last interested player in a cell leaves it, the state is lost since the home node does not hold cell state.

Jiang et al. (JIANG; SAFAEI; BOUSTEAD, 2007) also build a multicast tree out of each game world cell. They argue that the typical 'peer' node of peer-to-peer networks, with the typical low upstream bandwidth, is not able to service many interested players in a cell. Thus, they make it so that the coordinator of a cell only has to send (large) state update data to a few player nodes, which then help to disseminate it further to other players. Their focus is on communication scalability and they only employ one coordinator (called a *locale server*) per cell, resulting again in a state cheating vulnerability.

#### *2.7.3.2 Single object owner models*

In Colyseus (BHARAMBE; PANG; SESHAN, 2006), *'all updates to an object are serialized though exactly one primary copy in the system'*. It does not seem that objects are to be grouped based on an arbitrary cell division of the virtual world. Instead, they support *range queries* in their DHT which allows the game developer to retrieve arbitrary groups of objects. They also perform replication, but such replication is made in a master-slave way where exactly one master node can decide the state and the slave replicas must accept what the primary owner announces as the object's state. As before, this is vulnerable to state cheating.

In Hydra (CHAN et al., 2007), the focus is on facilitating development by leveraging client-server programming patterns. This is achieved by an architecture composed of Hydra client and an Hydra server node APIs which hide the details of peer-to-peer operation from the game developer. The game world is divided into *slices*, and each world slice is hosted by a *primary slice* (now the term is used to denote a node, not a piece data) and also by *backup slices*. The state of the backup slices is obtained from the primary slice, thus a malicious primary slice can at least 'boot' some part of the game state to be anything it wants.

Mediator (FAN; TAYLOR; TRINDER, 2007) combines three different styles of peer connectivity. First, it uses structured P2P overlay (DHT) for bootstrapping, a task where problems from DHT multihop routing are not an issue. Second, it uses Application Layer Multicast (ALM) to achieve efficient zoning structure maintenance. And third, it connects

players directly for dissemination of time-critical events. Mediator supports load balancing (BEZERRA; GEYER, 2009) and avoids hot-spots by dynamic splitting of zones as needed, as in other words, and it assigns a group of super-peers (also called *mediators*) to handle each zone. In Mediator, zone coordination tasks can be given to several different super-peers, instead of relying on one node to perform all of them. However, whichever node wins the *Zone Mediator* role becomes responsible for selecting which nodes will perform the other roles, so this scheme does not warrant any added protection to single-coordinator abuse. They also mention that single client machines can volunteer to host individual non-avatar game objects such as NPCs, which is why we classify Mediator in the *single object owner* sub-category of models that are vulnerable to state cheating. However, it probably presents a similar state cheat vulnerability at its zoning layer, as per the above discussion.

#### *2.7.3.3 Single master coordinator models*

Yamamoto et al. (YAMAMOTO et al., 2005) hands each cell to one coordinator and, when one coordinator is unable to handle the load, it can split its cell into sub-cells which are then handled given to subordinate coordinators, resulting in a tree overlay for each cell where the coordinator of the parent cell is the root. The increased delay of having to multicast through a tree of coordinators is addressed through a technique that expedites the process. They employ *back-up coordinators* in each cell. The back-up nodes help with event dissemination and they also serve as a replica should the master coordinator fail. However, it is unclear whether the back-up nodes can prevent the master coordinator from performing state cheating, as the paper doesn't mention what happens if master and back-up coordinators for the same cell diverge.

Proximity (MALIK, 2005) is a scalable P2P architecture for latency-sensitive MMOGs. Each player keeps the state of its own avatar, and updates other nodes regarding the state of the avatar. Non-avatar objects are hosted in game world zones, where each zone's state (collection of objects) is dictated by a zone coordinator which is elected out of player nodes. Each zone coordinator maintains a chain of back-up replicas. This is vulnerable to state cheating since the back-ups are just silent mirrors of the master coordinator. Should the master coordinator be compromised, the state of non-avatar objects can be set to anything it desires.

Peer Clustering (CHEN; MUNTZ, 2006) is an hybrid architecture that combines several techniques. It is based on a cell division of the game world terrain as usual, and it solves hot-spot cells by simulating these at the server-side. Each region is properly replicated in a *region manager* and in several *backup region managers*. The paper states that region manager *'maintains official control of a given region'*, thus we assume that the region manager has ultimate authority in determining the contents of a region. Additionally, the region manager is the only replica that receives commands from the players. The backup region managers instead receive the player commands from the region manager. Thus, even if a majority check is added to the model, the region manager is able provide its own bogus commands to the backup region managers to force the region to hold any state that it wants.

#### *2.7.3.4 SimMud and related work*

Knutsson et al.'s SimMud (KNUTSSON et al., 2004) employs a hybrid of client-server and DHTs, separating game state into *persistent* and *transient*. Persistent state is to be managed securely at trusted servers, while transient state is the state in each region (cell)

of the game world. The transient state of each region is further classified in either *player state* and *object state*. Player state is the state of each avatar, and players are authoritative on their avatar's transient state – we believe this doesn't include the wealth of an avatar since this is probably part of what they consider to be *persistent state*. Object state are all non-avatar objects of the game world which are shared between avatars. Object state is also maintained by a single *coordinator* of the region where the object resides. The single region coordinator *'not only coordinates all shared objects in the region, but also serves as the root of the multicast tree'* (KNUTSSON et al., 2004), that is, object updates are disseminated to interested players through a tree which works around bandwidth limitations on the coordinator. Additionally, objects are not only grouped by region but also by *object type*. Object types are to be defined by application, and each region can have one coordinator for each object type defined, which further helps with load balancing.

The SimMud model employs replication over the single coordinator of each region. However, that replication is a primary-backup mechanism that is built to *'tolerate fail-stop failures of the network and nodes'* (KNUTSSON et al., 2004). The paper suggests that the system continues to work fine even if only one coordinator replica remains, and that replication carries on in the background. This means that new replica data is obtained from the replica that was the only one remaining earlier, which allows any last replica of a region to perform state cheating. The paper argues that failures are sufficiently uncommon, but in the face of highly motivated attackers with DoS capability and the high payoff of state cheating, this cannot be assumed.

However, we must recall now that SimMud's *persistent state* is kept securely at the server-side. However, we do not believe that this avoids state cheating. Any persistent state will necessarily be derived from transformations in the transient state. For instance, if a coordinator is able to fabricate arbitrary items in the transient state, and it is able to resolve the action of an avatar picking up such items in a cell, then we must assume that the coordinator is able to send messages to the servers that add arbitrary amounts of in-game wealth to the persisted account of the player in question. Thus, the problem of state cheating is only obscured and not solved. However, this separation was probably done in SimMud to achieve fault tolerance, since they acknowledge that their scheme does not guarantee persistence of cell-hosted state (transient state) in all circumstances. The separation was probably not intended as an anti-cheating measure. Thus, we conclude that SimMud is vulnerable to state cheating, which is potentially too damaging to virtual economies.

Wierzbicki and Kaszuba (WIERZBICKI; KASZUBA, 2007) is a significant extension of Knutsson et al.'s SimMud. It employs a single coordinator in each cell which must be trusted by all players. It also allows two players to interact directly, in which case a trusted arbiter is elected to oversee the transaction. Each player keeps its own state, but it must be able to *prove* the legality of its current state. This is achieved by having players accumulate a chain (an history) of signed modifications and testimonials from zone coordinators and arbiters. The model is much more complex than the ones reviewed so far. It may be secure against state cheating, but accumulating an ever-growing history of proof of the legality of avatar-held state seems infeasible in the long run. Their results show an interaction response time which in the order of one second and that seems to be mostly CPU-bound.

### 2.7.4 Multiple arbiter

The single arbiter models discussed earlier either keep cell or object state in one peer, or they replicate them in such a way that fault tolerance against non-intentional failure is achieved. However, in a threat model that considers adversarial behavior, the state has to be replicated in such a way that multiple replicas of state are authoritative. In other words, if a subset of a replication group tries to perform state cheating, the honest replicas have sufficient authority to either reliably eject the cheaters from the system or at least prevent illegal modification to occur by forcing both the honest and the cheated versions of the state to be dropped.

We found nine peer-to-peer MMOG models in the literature that fit the 'multiple arbiter' criteria and thus are resistant to state cheating. However, they do not consider network-level attacks. In all these remaining works, either *state cheating can be performed against the model with the help of network-level attacks*, or *part of the game state can be forced to be lost with the help of network-level attacks*.

Initially, one might argue that network-level attacks are sufficiently uncommon. There are studies that report worldwide monthly incidences of around 100 to 1,000 attacks per month (GILES; MARCHETTE; PRIEBE, 2004) to around 10,000 attacks per month or more (MOORE et al., 2006)[34]. Also, as most peer-to-peer systems, peer-to-peer MMOGs tolerate attacks in the sense that failed peers are eventually replaced and cells eventually resume processing and disseminating events to players. Thus, we can assume that peer-to-peer networks cannot be neutralized by any dedicated attacker or group of attackers performing network attacks.

However, if we assume that guaranteeing the correctness of the state of each single cell in a MMOG is crucial to maintaining the virtual economy as a whole in a healthy state, then these models do not actually tolerate network attacks. The system as a whole is made out of thousands of peers, but each cell or object is perhaps replicated among ten, twenty or maybe even thirty live replicas, at most. If the IP address of all these replicas is known, and if they have low networking capacity, a DoS attack against a few replicas becomes much more feasible. So, even if maybe only one replica of a cell is malicious, that replica can become the only copy should a dedicated attacker decide to remove the other few replicas through network attacks. So, ultimately, it doesn't matter which cell is compromised. If any cell is compromised, and cells are able to hold and generate any amount of in-game virtual wealth, then the economy can be dominated by cheating players.

Actually, most works that replicate cell state do not describe what happens should all but one or a few replicas of a cell remain. That is, they consider that mass failure in a cell is unlikely. That is true of non-intentional failure. The problem of a single compromised cell wreaking havoc in the global game economy is not considered in any other work. The only exception is our eariler model, FreeMMG (CECIN, 2005; CECIN et al., 2004), which explicitly prevented state cheating from occuring. However, FreeMMG paid a high price to achieve this. The main problem with FreeMMG is that cells attacked as described above would simply lose their state. That can be as damaging to the game economy as state cheating, not to mention that loss can break the global consistency of the game. For example, state loss in a cell may remove an essential unique avatar or

[34]These works vary on several details such as the time when the measurements were made and what constitutes an 'attack' in each work's count of attacks. We have only checked these counts superficially as we were interested in an approximate order of magnitude for world-wide DoS and DDoS attacks only – thousands vs. millions.

item from the global game. Also, FreeMMG featured multiple negative game QoS issues such as low responsivity (this was why we limited it to MMORTS support), no inter-cell interaction support besides simple object transfers, and it also presented excessive server-side involvement in the game due to object transfers being brokered by the server. We attempt to address all these issues in FreeMMG 2 while also trying not to raise new issues.

Besides FreeMMG, the other eight works in our multiple arbiter category are reviewed below. We only review them briefly here since most of them will be compared in greater detail with FreeMMG 2 in Chapter 6.

#### *2.7.4.1 Non-architectures: NEO and SEA protocols*

Among other works that suggested that employing replicas with equal standing in determining the correct value of a shared piece of state within a small group of peers, the NEO protocol (GAUTHIERDICKEY et al., 2004b) seems to be one of the first together with FreeMMG. NEO also proposed the use of randomly chosen *witness nodes*, the same as FreeMMG's randomly chosen replica peers, to prevent players from manipulating group composition by simply moving their avatars around the virtual world. NEO proposed a novel voting algorithm which prevented several time cheats and also the *collusion cheat* (state cheat) from happening. In NEO, the majority always determines which events are input for each round (step) of the replicated simulation. The state of each replica evolves deterministically since the set of events of each round is always the same at all honest replicas. The SEA protocol (CORMAN et al., 2006) is an enhancement over NEO which fixes problems identified in NEO. Both are not actual MMOG support architectures. The authors of NEO mentioned an architecture in the works that would use NEO as the synchronization algorithm for each world cell.

#### *2.7.4.2 Models that are vulnerable to state cheating if network attacks are considered*

In these works the vulnerability to state cheating in the presence of network attacks can be more definitely established. This is not always the case since, as we noted earlier, most works do not provide enough details about their replica failure detection and recovery process to assess this.

RAP (ENDO; YANG; ZHANG, 2006) assigns exactly three peers to replicate the state of each cell. A consensus of two out of the three nodes is required to make any decisions that affect cell state, which is used to filter cheating attempts. State cheating in RAP requires only a substantial amount of nodes to serve as volunteer replicas. Once two such colluding cell replica maintainers land in the same cell by chance, the game's economy is compromised with blessing from the architecture. That is, the nodes that would be performing network attacks against peers can instead simply volunteer as overlay maintainers or players and wait.

Izaiku et al. (IZAIKU et al., 2006) propose an architecture whose cell synchronization protocol is very similar to the one in FreeMMG 2. Their idea is that each cell's state is maintained by multiple replicating nodes. One of these replica nodes is the *responsible node*, while the others are the *monitor nodes*. However, all nodes evolve the cell state deterministically, and the role of responsible node continually changes hands among all replicas to minimize the effects of minor cheating. The responsible node is the only one that sends updates to players interested in the cell, while both the responsible node and the monitor nodes receive commands from players. Since players deliver their commands to all replicas and the deterministic simulation rules are static and previously known, the

majority of honest replicas is able to have an agreeing state in spite of interference from a minority of malicious replicas. Izaiku et al. does not consider network-level attacks, so we can assume that the mutual checking between replicas would be vulnerable to state cheating once there is only one replica left. Izaiku et al. also does not *prevent* state cheating from happening, but instead relies on *detection*. This has the added issue of requiring undoing of illegal state modifications or at least compensation. That problem is not addressed.

Hampel et al. (HAMPEL; BOPP; HINN, 2006) employs a Region Controller (RC) plus a number Backup Controller (BC) peers for each region. At least the communication between RCs and BCs is performed through a DHT. The RC sends state update packets to players interested in the region, like discussed in Section 2.4. Additionally, all BCs of a region send *hashes* calculated over the state. So, the players can confirm that all controllers agree on the exact state of the cell by comparing the hashes they receive between each other. Since only the RC sends large update packets to players, their potentially low download bandwidths are not saturated. Also, as with Izaiku et al. (IZAIKU et al., 2006) and several other multiple arbiter works, both the RC and the BCs receive all player commands directly from each player interested in the cell. Hampel et al. do not consider the effect of network attacks, that is, of mass-failure of replicas.

#### *2.7.4.3 Models that do not address network attacks and that peform excessive messaging*

In addition to not dealing with network-level attacks, the three remaining multiple arbiter models require all or a majority of replicas of a cell to send a stream of large state update packets to all players currently interested in the cell. These are Kabus and Buchmann's *Cheat-Resistant P2P Online Gaming System* (KABUS; BUCHMANN, 2007), Webb et al.'s *Secure Referee Selection* algorithms (WEBB; SOH; TRAHAN, 2008) and Endo et al. (ENDO; KAWAHARA; TAKAHASHI, 2005).

The *Cheat-Resistant P2P Online Gaming System* (KABUS; BUCHMANN, 2007) employs a rather unique approach to maintaining the replicas of a region synchronized. Each replica keeper of a region, referred to as a Region Controller (RC), receives a command from a player, computes the next state, and sends back an update. Thus, players receive one update from each RC, and they can detect faulty RCs. Since updates are digitally signed, a player is thus able to prove that it has received a couple faulty or malicious updates out of a large set of otherwise agreeing updates. Thus, synchronization among RCs becomes unnecessary. However, since update messages can be large and the architecture intends to draw RCs out of users that are also currently playing the game, this may end up depleting the upload bandwidth of player machines. That would not be a problem if the deployment scenario considered that, for some reason, the upload bandwidth of the average player node is in the order of Mbps instead of a few hundred Kbps.

Webb et al.'s *Secure Referee Selection* (SRS) algorithms (WEBB; SOH; TRAHAN, 2008) draws inspiration both from RACS (WEBB et al., 2007) and Kabus and Buchmann's *Cheat-Resistant P2P Online Gaming System* (KABUS; BUCHMANN, 2007) and its multiple RCs technique. Actually, the SRS model carries the 'referee' terminology from RACS (which in turn comes from PP-CA (PELLEGRINO; DOVROLIS, 2003)) but the referees of SRS are converted into RCs as in Kabus and Buchmann's model. Thus it is easier to think of SRS as a way to securely select region controllers, not referees, since they don't work as referees like in RACS, EMS (WEBB; SOH; LAU, 2007) or PP-CA.

SRS is composed of two parts. First, it mimics Kabus and Buchmann's RCs by making all RCs send full updates to all players interested in a region. Second, it attempts to

select a reduced number of RCs that are less likely to collude (for instance, to perform state cheating) instead of blindingly choosing large amounts of RCs randomly hoping to minimize the likelihood of collusion groups forming. They attempt to achieve this secure selection by searching for RC candidates that are sufficiently far apart from each other on the Internet. This does not prevent collusion if an attacker group controls machines in several parts of the Internet, which would be the case if the system is being Sybil-attacked by a botnet for example. The reduced RC count also aggravates the problem of network-level attacks, which can either be used to facilitate collusion or at least force the state of a region to be dropped.

Endo et al. (ENDO; KAWAHARA; TAKAHASHI, 2005) propose an adaptive system where the simulation of regions is initially performed at the server-side and, whenever the server detects that it exceeds some processing or communication load threshold, it dynamically offloads game world regions to groups of peers. In this 'peer-to-peer mode', each region is replicated among several peers which are referred to as *site servers*. Players send their commands to all site servers of a region simultaneously. Site servers order the incoming player events based on the median arrival time among all site servers, and the event is not timestamped (and therefore executed) before that median value is known. Execution of events must be deterministic as usual to maintain the replicas synchronized. Each user receives $(N-1)/2$ full update messages and $(N+1)/2$ update hashes for each command it broadcasts to site servers, where $N$ is the (odd) amount of site servers managing the region. The players only accept an update once a majority of agreeing locally-computed update hashes (from the full updates received) or site server update hashes match. Though the amount of full update messages sent by each site server (RC) is reduced to only half of the site servers (RCs), we consider this cost to be still rather significant. One of the main purposes of these multiple full updates is to cover for congestion: some updates may be delayed for several seconds due to temporary congestion but at least one may get through to the player.

Endo et al. assert that probability that a large number of colluding malicious replicas will end up being randomly assigned to the same cell is small, which is an assertion that our own work also depends on. However, they do not address the scenario where replicas with public IP addresses can be selectively taken down by a highly-motivated attacker that has enough resources to perform network-level attacks.

### 2.7.5 Other models

In this section we review two approaches that didn't quite fit in the above categories. The first is the PSMMO model which, instead of building a peer-to-peer MMOG architecture with virtual economy support, lays a virtual economy on top of the existing online populations of thousands of unrelated servers of non-massive MOGs. The second is the Asynchronous Synchronization (AS) protocol and related work which warrants dedicated analysis.

#### *2.7.5.1 The PSMMO model*

The PSMMO model (CHAMBERS; FENG; FENG, 2006) leverages the *Public Server* deployment model of small-scale client-server MOGs, where anyone is able to start and stop their own server independently from others. The PSMMO model inserts long-term game state persistence by adding support to shared 'loot' (items) that players can acquire at one game server and later carry to any other server. For instance, in a 3D FPS game such as Counter-Strike, players could accumulate more powerful or exclusive weapons

as they play the game for years in hundreds of different servers. To prevent servers from simply fabricating arbitrary amounts of loot and thus ruin the game economy, all pieces of loot have to be cryptographically signed by a central loot authority which is the only party that is able to mint loot. The loot server trades loot for *authenticated player minutes* with each anonymous game server. Simply put, a server receives loot proportional to the raw amount of time that human players spend at that server. To prevent servers from fabricating minutes or from using bots to generate minutes, those have to be *authenticated* by CAPTCHA tests that, using creativity, can be blended into the game environment to avoid game disruption. Finally, game servers that distribute loot unfairly to its players will naturally drive away real human players and cease receiving loot, which forces them to be sufficiently fair in loot distribution.

#### *2.7.5.2 Asynchronous Synchronization protocol and related work*

Baughman and Levine (BAUGHMAN; LEVINE, 2001) introduced cheat-proof protocols for real-time peer-to-peer games: the Lockstep protocol, the Asynchronous Synchronization (AS) protocol (an improvement over Lockstep) and the Cell-Based Hidden Positions protocol (CBHP)[35] for cell-based MMOG architectures. In a follow-on work, Baughman et al. (BAUGHMAN; LIBERATORE; LEVINE, 2007) also propose the Sub-Cell Hidden Positions (SCHP) protocol, which fixes problems in CBHP. Lockstep and AS guarantee both cheat-proofing and consistency between player simulators (LPs) which may be potentially affecting each other's outcome, that is, which are sharing or can potentially come to soon share an AoI over the virtual world due to increasing proximity. CBHP and SCHP in turn avoid information exposure by allowing players to conceal their exact position from each other and still detect whether there is AoI proximity or not. SCHP does so by obfuscating player's relative location into crypto hashes which can be sent to nearby players and compared. Matches on such obfuscated hashes indicate that players are approaching AoI reach and should start running the AS protocol.

Though in all these protocols the players have to connect to each other and send authoritative updates to each other, they achieve more in terms of cheat-proofing than the other 'player mesh' models. AS with SCHP detects time cheats (e.g., vulnerabilities in dead-reckoning) and information exposure cheats.

However, Baughman et al.'s work doesn't address state cheating explicitly. This is probably because their work is geared towards action games where a virtual economy is secondary. That is, in AS and the games envisioned for it, the generation and flow of virtual wealth could be trivial to be handled by a central third party. Or maybe the generation of virtual wealth could be limited to one-time cryptographic minting such as in the PSMMO scheme.

Baughman et al.(BAUGHMAN; LEVINE, 2001) mentions a central *observer service*. It is described as a cheat detection service that would monitor players in real-time to check whether the *secret information* stored authoritatively at each avatar is legal. However, it is unclear whether such service would guarantee the ultimate legal origins of secret and public information which is stored only at each avatar. So, it is possible that avatars could fabricate virtual items of arbitrary types and in arbitrary amounts. This is crucial if play would be centered around a virtual economy. Baughman et al.'s work seems geared towards a scenario where this problem could be solved adequately by add-on work due to

[35]The CBHP protocol is described by Baughman and Levine (BAUGHMAN; LEVINE, 2001) but it is not named in the original paper. The CBHP name is given by the authors retroactively in Baughman et al. (BAUGHMAN; LIBERATORE; LEVINE, 2007) which is an update on the original work.

it not being its focus. Instead, the focus of AS and related protocols is clearly in cheat-proofing against time and information exposure cheats, and in guaranteeing consistency and timely resolution of conflicts.

Ghost (JOHN; LEVINE, 2005) is a peer-to-peer MMOG architecture that also extends AS to massive scale. In Ghost, if two players are in Sphere of Influece (SoI) range, one may unilaterally opt out from interacting with the other if the communication delay between them is too large. Alternatively, Ghost also allows players to be grouped into *teams*, where any player P either plays with an entire team T or plays with no member of team T at all. The description of Ghost doesn't contain any considerations towards securing the state that is dictated by each avatar and thus fits into our 'player mesh' classification.

## 2.8 Closing remarks

In this chapter, we have reviewed some diverse background information. In Section 2.3 we have discussed how distributed simulation techniques can maintain consistency among several simulator processes (LPs) which typically have equal status in a simulation hierarchy. In Section 2.4, we have shown that online games employ a simpler model where one simulator, running as the *server*, has no such synchronization issues. Thus, client-server protocol developers were are able to ignore replication issues and actually experiment on several advanced techniques to improve responsivity, consistency, etc. In Section 2.5 we discussed cheating in online games, attempting to show that for MMOGs, *state cheating* dominates other forms of cheating in a scale of potential damage to long-term game play. In Section 2.6 we introduced the issue of dealing with network-level attacks in peer-to-peer MMOG architectures.

Finally, in Section 2.7, we performed short reviews of any and all existing peer-to-peer MMOG architectures that we could find. To characterize a niche for our FreeMMG 2 model, presented in the upcoming chapters, we have split these works into a few broad categories which helps us make our case. The first category was of *player mesh* models, which either do not support MMOGs with relevant game economies or do so with security holes. The second category was of *fat server-side* models which either leave too much CPU-bound tasks or communication-intensive tasks to the server-side, sometimes both. The third category was of *single arbiter* models, where state cheating is trivial. The fourth category was of *multiple arbiter* models, which is where FreeMMG 2 fits in. We have shown that existing *multiple arbiter* models do not consider the effects of network-level attacks.

In our FreeMMG works, the whole design revolves around providing strong resistance against illegal modifications to the game state. By *illegal modification* we mean any transformation or update to the game state (e.g., a cell's state) that does not abide to the programmed rules of the game. By *strong resistance* we mean guaranteeing that it is extremely unlikely that state cheating can be performed. In FreeMMG and FreeMMG 2, we have elected one way to guarantee such strong resistance together with significant decentralization:

- Replicate state (e.g., cell state) among a significant number of volunteer nodes;
- Choose part or all replicas (of each state partition) randomly out of a sizable pool of volunteers. That makes it unlikely that a majority of colluders for a state partition is ever assembled out of the architectures' own overlay construction algorithm;

- Any observer of the replicated state must employ a consensus algorithm to determine if the cell's state can be determined. However, if too few of the randomly-chosen replicas remain, that algorithm should not trust the resulting consensus state, if any. Thus, employing any consensus criteria such as Byzantine, Crusader, or arbitrary quorums is not enough: the nature (legal origins) of each replica is also important. This is because, in our view, one compromised cell defeats the purpose of thousands of non-compromised cells in the global game.

In other words, we'd rather have the state of a cell be dropped, replaced by an old snapshot (CECIN, 2005), or reconstructed from coarse-grained light-weight server-side data (CHEN; MUNTZ, 2006) than allow a window for state cheating. We believe that MMOGs have, at their heart, the motivation to compete in a long-term dispute with other players in an economical meta-game that feeds from other subordinate game activities such as dueling, hunting monsters, completing quests and performing social interactions. That is our vision of what MMOGs are and we acknowledge that other peer-to-peer MMOG designers may and do disagree.

Some may argue that our state integrity focus and our level of concern with network-level attacks are premature or exaggerated. One can certainly make the case that peer-to-peer MMOGs designers currently have more pressing issues to attend to, such as guaranteeing scalability, responsivity and efficient use of limited peer resources. However, if and when peer-to-peer MMOGs (as in actual implemented and deployed massive *games*) become popular, we believe that state cheating vulnerabilities will kill them regardless of how scalable, responsive and efficient those architectures may be. No player will put effort in evolving a shared state according to harsh rules, 'grinding' for character power and status, when one can game the system to instantly apply arbitrary modifications to the shared state. Thus, we have decided to focus on securing the game economy from the start, even if that results in blatant QoS issues and other problems which is the case with the original FreeMMG model (CECIN, 2005). The stance that protection against arbitrary state manipulation should be uncompromisable during peer-to-peer MMOG model design is a small but original contribution of the FreeMMG works, as far as we can tell. Even if we fail to reach real-world deployment, we hope that we have at least enriched the discussion around peer-to-peer MMOG support.

There may be other ways to achieve both significant reduction of server-side infrastructure and protection against state cheating besides replication of state in multiple arbiter nodes (replicas with equal status). A few works attempt to leverage cryptography to guarantee the integrity of distributed computation and state. Endo et al. (ENDO; KAWAHARA; TAKAHASHI, 2007, 2006) investigates the use of homomorphic encryption, which could let peers act as proxy servers for some state while not being able to make chosen (meaningful) modifications to that state. However, in this thesis we have not evaluated works that use cryptography as the main source of cheat mitigation.

# 3 INTRA-CELL REPLICA SYNCHRONIZATION

As seen in the previous chapter, there are several peer-to-peer MMOG architectures that are both cell-based and replication-based, be the replication in a single arbiter setup (one master and multiple ghost copies) or multiple arbiter one (consensus based on some quorum of voting replicas). In these cases, each cell in the MMOG architecture has multiple replicas, and these replicas have to be constantly kept in synchrony. For example, in FreeMMG 2, players will connect to one of the replicas of a cell to play the game. If player commands are being issued individually to different replicas of the cell, the cell replicas will have to somehow synchronize with each other to compute a unified result out of executing the events submitted simultaneously by all players. That synchronization has to be performed in a timely fashion, as players are interacting in a real-time virtual environment and thus are expecting to see the results of their actions and that of their peers sooner rather than later.

That general issue of coordinating the multiple replicas of a cell is what we call *intra-cell synchronization*. In this chapter we describe the intra-cell synchronization approach, called Baseline State Synchronization (BSS), that will be used in the FreeMMG 2 peer-to-peer MMOG architecture, which will be presented in the next chapter.

The chapter is organized as follows. We begin the chapter by introducing the problem of intra-cell synchronization for cell-based, replication-based MMOG architectures in Section 3.1. In that section we also explain that BSS is based on the Trailing State Synchronization (TSS) mechanism. In Section 3.2 we explain the motivation behind TSS, which is that of maintaining the multiple servers of a Mirrored Server (MS) system synchronized. In Section 3.3 we explain how the TSS algorithm works. In Section 3.4 we explain how BSS works, by showing the difference between BSS and TSS. Finally, In Section 3.5 we show what the FreeMMG 2 cell looks like and how BSS fits as a synchronization solution for it, and Section 3.6 concludes.

## 3.1 Introduction

A game world instance can be entirely simulated by one machine, but the task of running a single instance can also be distributed among several machines. Typically, game simulation is distributed for one of the following reasons:

- Scalability: Simulating the game instance becomes a more costly task, especially in terms of CPU cost, as more players and more game objects and complexity are added. To increase the amount of supported game complexity beyond what a single machine can handle it becomes necessary to distribute the load across several machines. Some perform a static distribution of tasks to machines, while others

dynamically balance the load among server machines by moving tasks away from overloaded nodes (CHEN et al., 2005);

- Latency reduction: If a game world is being simulated by several server machines distributed across the globe, players can connect to the nearest machine to have a reduced communication latency from and to the game world (BESKOW et al., 2008). This works only if the connectivity between server machines is of high quality. In this way, the cooperating servers are able to communicate more effectively across significant geographical distances than the players. In other words, distant players experience lower average latencies by taking the 'server highway' than if they had to communicate with a distant party such as the other player directly or a single server;

- Preventing state loss and state cheats: If a persistent-state game world is to be simulated by consumer-grade, untrustworthy nodes that can either cheat, fail or leave the system at any time, then distributing the game world simulation among several machines is essential to prevent the whole game state from being lost and to prevent players from being able to alter the game state illegally (CECIN; BARBOSA; GEYER, 2003).

The first two reasons are specific to server-centric (or 'fat server-side') architectures. As discussed in Chapter 2, scalability of a client-server MMOG design is achieved by *partitioning*: each server machine runs part of the game world and is responsible for part of the game objects. The partitioning can be either completely visible to players, such as in several commercial MMOGs that are instanced, or transparent to players as in some commercial MMOGs such as EVE Online (BRANDT, 2005). Latency reduction, on the other hand, is achieved by *replicating* the state of the virtual world among trustworthy and well-provisioned servers which can be geographically distant from each other but that experience no congestion, no bandwidth limitation problems and near minimum latency when synchronizing with each other. In addition, the dedicated network that the mirrored servers have at their disposal can be more easily arranged to support IP multicast. IP multicast reduces the latency even further and also the cost of one server broadcasting the updates or events that are supposed to be executed (mirrored) at all the other servers (CRONIN et al., 2004a).

For peer-to-peer games, the state of a game instance can be *replicated* among untrustworthy and fault-prone nodes to make the simulation tolerant to a broad range of intentional and non-intentional faults. As an example, the commercial RTS game Age of Empires is a serverless (peer-to-peer) game where the game state is fully replicated at each player machine (BETTNER; TERRANO, 2001). The player replicas in Age of Empires synchronize event execution conservatively using a protocol similar in spirit, but more advanced than the stop-and-wait protocol outlined in Section 2.3.6. Age of Empires does that to achieve a serverless support for RTS games, but they mention that the cheat-proofing is welcome side-effect of replicating the game state at each player (BETTNER; TERRANO, 2001).

The replication approach alone doesn't scale to massively-populated virtual worlds. Even if thousands of players could somehow synchronize their replicas in real-time, we would be left with the problem of storing and simulating a massive virtual world at each consumer-grade node. Replication can however be combined with partitioning to build a peer-to-peer MMOG architecture that is feasible. Thus, all replication-based approaches

for peer-to-peer MMOGs reviewed in Chapter 2 also employ a partitioning scheme where the game world is divided into *cells* (also called *regions*, *zones*, and others) which are, in principle, separate game simulators. Each partition (cell) then replicates its contents among several peers. After that, some architectures may optionally tie together the cells in some way to try and assemble a unified game play instance with a large and contiguous virtual world space. That latter issue (inter-cell synchronization) will be investigated in Chapter 4 and in Chapter 6.

The original FreeMMG model, which was among the first works to outline the cell-based replication approach for peer-to-peer MMOGs, was basically an extension of the peer-to-peer instance replication idea of Age of Empires to massive scale. The cells in FreeMMG were replicated among active player nodes, that is, players that owned at least one unit (object) in that cell. So, a cell in FreeMMG was the same as one Age of Empires game instance. And, similarly to Age of Empires, the replicas (which were also players of the game) inside the same FreeMMG cell synchronized with each other conservatively to avoid out-of-order event execution and thus inconsistencies among them. This resulted in a high-latency interaction which was suited to MMORTS games but that wasn't adequate to support avatar-based or action games.

In this thesis, we propose the FreeMMG 2 architecture which, in a sense, is an evolution of FreeMMG. What we carry over from the predecessor model is the basic idea of joining replication and partitioning to achieve fault tolerance, security and scalability in a peer-to-peer MMOG. However, in FreeMMG 2 we wanted to support avatar-based games such as MMORPGs and, if possible, avatar-based *action* games such as a MMOFPS. To achieve this, the intra-cell conservative synchronization algorithm had to be replaced with an algorithm that would both guarantee consistency among replicas and allow low-latency player interaction. Even if that algorithm didn't itself minimize player latency, we wanted it to be compatible with client-server latency compensation techniques such as the ones presented in Section 2.4.

The Baseline State Synchronization (BSS), which we present in this chapter, fills that gap for FreeMMG 2. BSS is a small modification of the Trailing State Synchronization (TSS) algorithm (CRONIN et al., 2004a, 2002). The main difference between BSS and TSS lies not in the algorithm but on the different purposes of each. TSS is a replica synchronization algorithm created to synchronize *Mirror Servers*; it was created to achieve player latency reduction in a server-centric game architecture. BSS on the other hand was created to support replication among *client machines*. Thus, BSS focuses on fault tolerance: resisting intentional faults (from cheaters, griefers and other adversaries) as well as non-intentional faults that can range from unplanned disconnections to fatal crashes. To achieve that, only minor changes had to be made to TSS, resulting in the BSS algorithm. The new name is mostly to avoid confusion; we published BSS first without giving it a name (CECIN et al., 2006).

In the next section, we will explain the TSS algorithm. Next, we describe the tweaked TSS, which we called BSS. Next, we show that BSS execution by nodes that aren't server-grade is feasible in terms of CPU cost (CECIN et al., 2006). We finish the chapter showing how BSS fits as the synchronization solution in a FreeMMG 2 cell. By addressing the problem of intra-cell synchronization in this chapter, which is the central and by far the most critical component of FreeMMG 2, we will be able to concentrate on assembling the overall architecture and in describing its many other parts in Chapter 4.

## 3.2 The problem of keeping mirrored game servers synchronized

The motivation behind the creation of the Trailing State Synchronization (TSS) algorithm was that *'None of the existing distributed game or military simulation synchronization algorithms introduced above are entirely suited to a game such as Quake*[1] *that has frequent updates, has a need for strong consistency and is very latency sensitive.'* (CRONIN et al., 2004a). To put that in context, the 'mentioned above' synchronization algorithms evaluated in the TSS paper were conservative protocols such as Fixed Time Bucket Synchronization (STEINMAN, 1995), the optimistic Time Warp algorithm (or family of algorithms (LOBATO; ULSON; SANTANA, 2004)), optimistic 'Breathing' algorithms (STEINMAN, 1993) and the optimistic Bucket Synchronization algorithm used in the MiMaze (Multicast Internet Maze) prototype (GAUTIER; DIOT, 1998b).

TSS was developed as a means to maintain servers synchronized in a Mirrored Server (MS) architecture (see Figure 3.1). With a MS architecture, one can lower the overall interaction latency and thus increase the visual consistency experienced by the players, as well as balance the communication load across server machines. That is achieved by distributing the mirror servers across a significant geographical area (e.g., country, continent or world-wide), interconnecting the servers with a well-provisioned and multicast-enabled backbone and having players connect to the nearest server.

Since the goal of the MS distribution scheme is to achieve latency reduction and increased consistency first and foremost, it makes no sense to partition the simulation load across servers. Instead, each server maintains a full copy (replica) of the whole game instance that is kept up-to-date. The focus of the synchronization strategy is in maintaining the differences among mirrors to a minimum and in always correcting any differences (inconsistencies) in a timely manner. That rules out a purely conservative approach to synchronization.

Thus, player commands have to be executed optimistically at each Mirror Server. Each mirror forwards the commands they receive from players to the other mirrors, resulting that all mirrors will eventually have the same input and thus it becomes possible to maintain mirrored states. However, if each mirror just executes each command immediately upon receipt, the different ordering and the different time of execution of each command can make the copies diverge.

## 3.3 Trailing State Synchronization (TSS)

To synchronize the mirrors, TSS works as follows. All mirrors share a network-synchronized clock, whose current value we will refer to as $t$. Each mirror maintains, in RAM, $N$ *states*, which are copies of the state of the whole game world, where $N$ is a fixed number. The several *states* stored in each mirror will represent the state of the world at different simulation times and thus they are not supposed to be identical to each other. Each state runs with a different and fixed event execution delay, called the *synchronization delay* of that state. A state with a synchronization delay of $d$ only executes an incoming event with timestamp $t_e$ when, locally, $t_e \leq t - d$. As discussed in Chapter 2, the events, which are typically avatar commands such as 'fire' or 'move' issued by players, can either be timestamped when they are received by the mirror server (minimizing cheat opportunities) or when they are generated at the player node (maximizing fairness) – it doesn't

[1] Quake (ID SOFTWARE, 2008) was the first online First Person Shooter (FPS) for the PC platform with true 3D rendering. It was released in 1996.

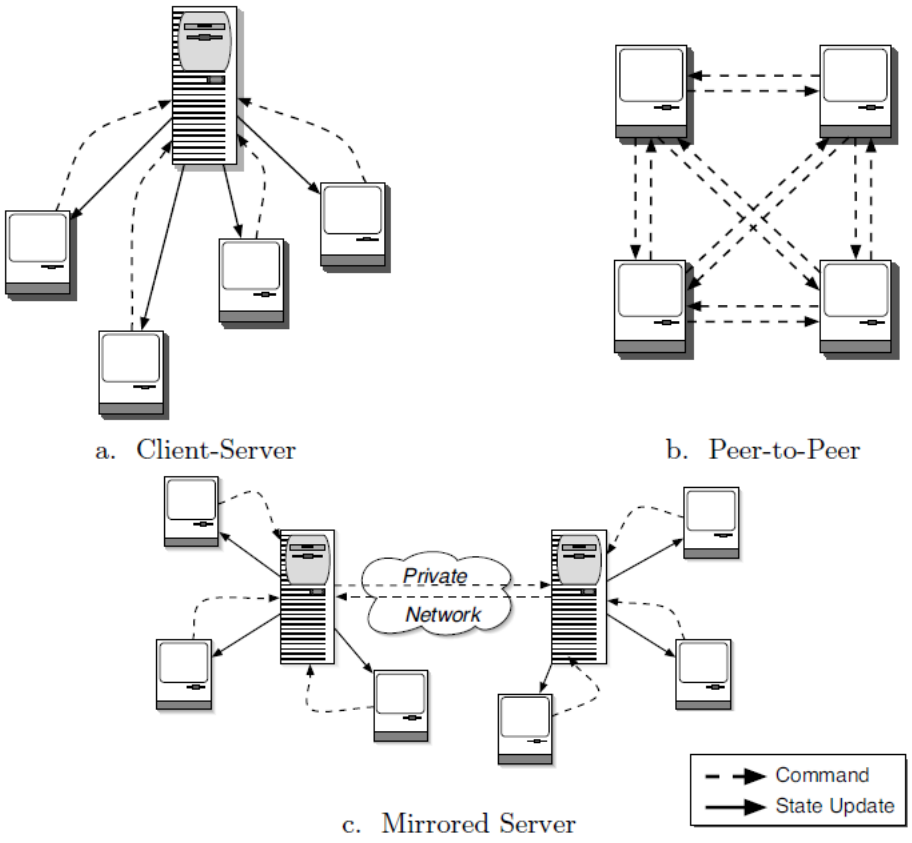


Figure 3.1: Mirrored Server architecture compared with pure Client-Server and Peer-to-Peer architectures. Copied and adapted from Cronin et al. (CRONIN et al., 2004a).

really matter for now.

Thus, each state is running a delta $d$ of simulation time 'in the past' and thus 'waits for $d$ to elapse' since an event 'occurred' to execute that event. For example, a state that has $d = 50ms$, at $t = 1000ms$ will have only executed events with timestamps earlier than $950ms$. Thus, at $t = 1000ms$, if the mirror receives a 'late' command (relative to $t$) timestamped with $t_e = 970ms$, it won't execute that event – not until $t \geq 1020ms$, when the simulation time of the state reaches $970ms$ (remember that it is running $d = 50ms$ behind the 'current' time $t$).

The reasoning behind this is that the chances of a state executing the incoming events out of order, out of their intended round or time bucket, or out of their precise timestamp (or any criteria for maintaining consistency) is minimized when $d$ increases. If communication latency between mirrors can be bounded to a maximum, then by assigning $d$ to that maximum we can obtain states at each mirror that only execute events 'correctly'. However, with a large enough $d$ the game becomes unresponsive to players, since their commands (such as avatar weapon-fire and movement) have to wait for $d$ to elapse at each mirror before being executed.

So, instead of simulating just one copy of the world at each mirror and trying to come up with a good value for $d$, TSS maintains $N$ states at each mirror, each state with its own $d$. For the following discussion, let $i$ be a positive integer in the interval $[0, N-1]$ which is used to refer to one of the states of a mirror. Thus, let $S_i$ denote the i-th *state* of a mirror. Let $d_i$ denote the synchronization delay of $S_i$. Let $S_0$ be the *leading state*,

and states $S_1$ through $S_{N-1}$ be the *trailing states*. Finally, $d_b > d_a$ if and only if $b > a$. That is, the states are sorted in an ascending order of synchronization delay. The leading state ($S_0$) will always have the smallest synchronization delay, while the last trailing state ($S_{N-1}$) will have the largest synchronization delay.

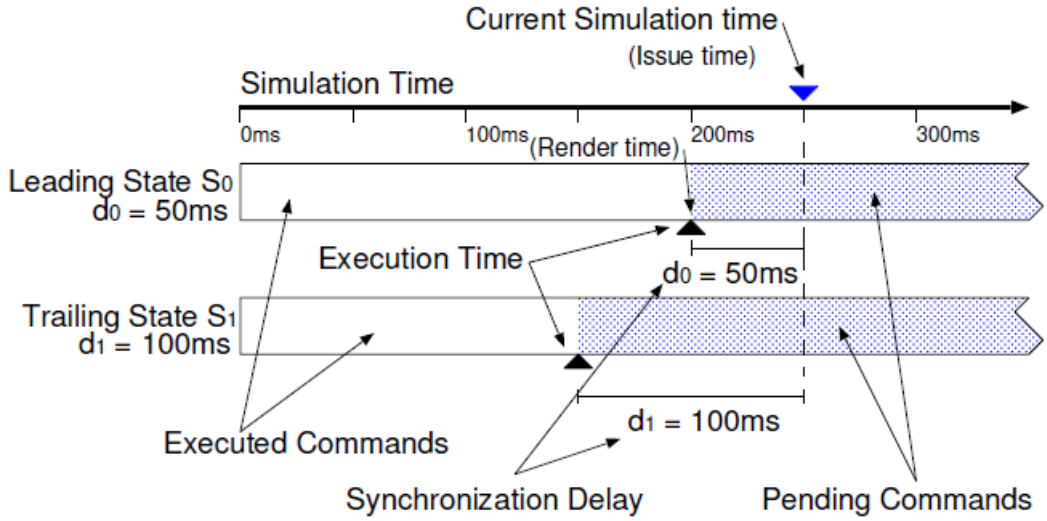


Figure 3.2: Example scenario that illustrates TSS terminology. Copied from Cronin et al. (CRONIN et al., 2004a).

When an event is received at a mirror, it is placed in an event list. The event will only be removed when it is integrated into all states in a definitive manner, that is, at the correct simulation time and order at all states. Figure 3.2, taken from a TSS paper (CRONIN et al., 2004a), shows an example TSS node running two states: $S_0$ and $S_1$. In the figure, the current simulation time is $t = 250ms$. The $S_0$ state (leading state) has a delay of $d_0 = 50ms$, thus its state is the result of executing all known events generated before $t = 200ms$, and any known events with timestamps greater than that have to wait on the event list before they can be executed at $S_0$. $S_1$ is running with a delay of $d_1 = 100ms$ behind $t$, thus it has only executed events up to $t = 150ms$. The leading state's time is said to be the *render time* since mirrors will send update packets which are constructed from data in the leading state. Thus, the leading state is what is *rendered* at player screens.

### 3.3.1 Detecting inconsistencies between local states

Besides being placed in the event list, it is possible that an incoming event is executed immediately by one or more states if it is already late for execution at that state. That is, a state executes immediately an incoming event if that event falls in the 'Executed Commands' region of that state as illustrated in Figure 3.2. However, in that case, it is possible that the late event has introduced inconsistencies. That is to say, late event executions may cause the *local* states of a mirror to diverge significantly. If an inconsistency is detected in a state $S_i$, then that state can restore consistency by performing a rollback to its following (trailing) state $S_{i+1}$ and re-executing the updated event list. But, before delving into the TSS rollback mechanism, we will dissect how inconsistencies can be detected in TSS in this section.

Detecting a potential inconsistency may be as simple as checking whether an incoming command is late or not. However, the original TSS algorithm compares states to check for

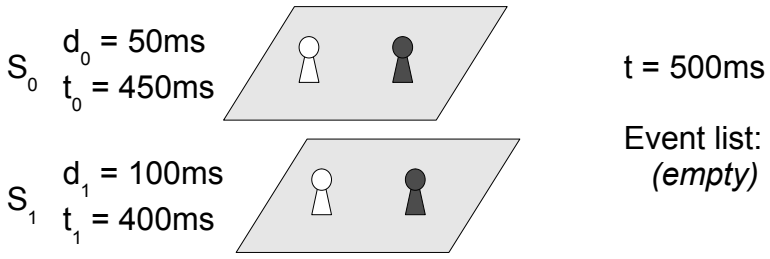


Figure 3.3: TSS inconsistency detection example (1 of 4): the example mirror server starts with all its states consistent.

divergences. We now provide a sample example execution of TSS in a mirror which keeps $N = 2$ states to illustrate the alternatives to inconsistency detection. Figure 3.3 shows the mirror at an arbitrary simulation time $t = 500ms$ where both $S_0$ and $S_1$ are in a consistent state (e.g., the initial game state) but running with their separate synchronization delays of $d_0 = 50ms$ and $d_1 = 100ms$, thus the simulation time of $S_0$, which we call $t_0$ is at $t_0 = t - d_0 = 450ms$. Similarly, $t_1 = 400ms$.

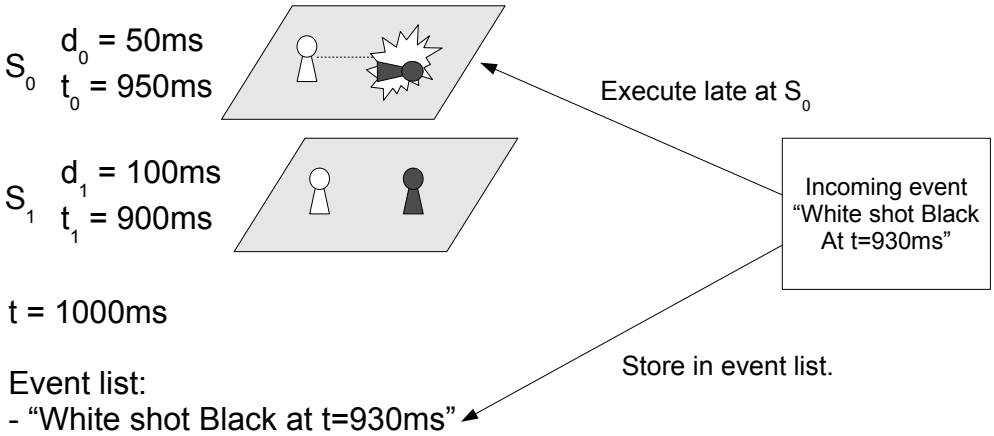


Figure 3.4: TSS inconsistency detection example (2 of 4): incoming event that is late at the leading state but early at the trailing state.

Figure 3.4 shows the mirror's state at simulation time $t = 1000ms$, that is, approximately 500ms of wall-clock time should also have elapsed. At that point in time, the mirror receives an event with a timestamp of 930ms which informs that White has shot a weapon at the Black avatar. Since $t_0 = 950ms$ the event is already late for execution at $S_0$, so it is executed immediately. In our example game rules, this results in a dead Black avatar. And, since $t_1 = 900ms$, $S_1$ cannot execute the event yet. And, since the event may still be relevant to the algorithm, it is placed in the event list of the mirror.

Figure 3.4 could be detected as *potentially inconsistent* by a simple implementation of the general TSS technique. And, actually, that is what we did in our extension of TSS. In Figure 3.4, the mere fact that an event is being executed *late* at $S_0$ indicates that $S_0$ may have potentially diverged from timestamp order execution of events since, in

principle, late events are a source of inconsistency across a distributed simulation. The only difference here is that, in TSS, all the 'distributed' states are local, that is, running at the same node (e.g., a mirror server).

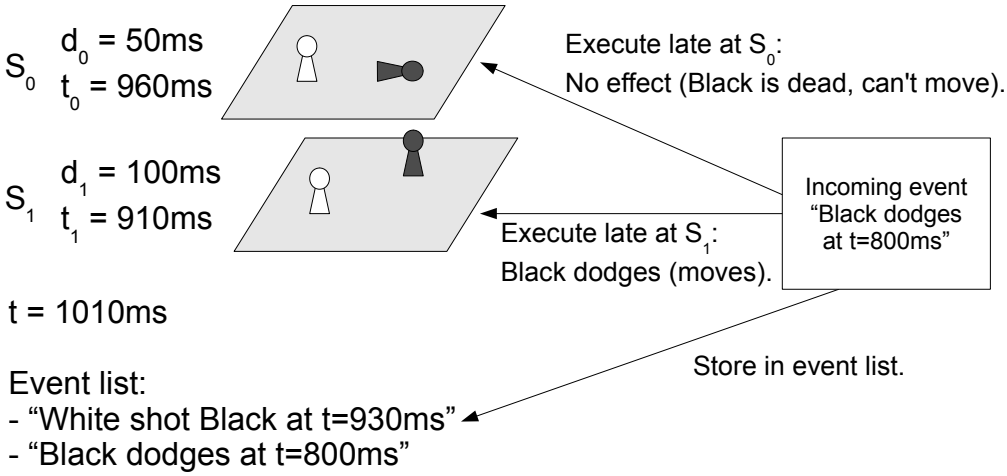


Figure 3.5: TSS inconsistency detection example (3 of 4): detectable inconsistency.

Figure 3.5 shows the system just 10ms later, at $t = 1010ms$. At that time, a (very) late event has arrived, which tells that the Black avatar's player has issued a 'dodge' command prior to the White avatar issuing its shot command. As discussed before, the '800ms' timestamp of the event can either be the time the event is generated at the player machine or the time in which it is received by the player's mirror server. Let's assume that, in any case, the event will be late at both states when received by the mirror server of our example. Since the event is late for both states, they both execute it immediately. At $S_0$, this results in the dead Black avatar doing nothing (remaining dead), while at $S_1$ this results in the still live Black avatar 'dodging'. For simplicity, let's assume that the 'dodge' command grants permanent immunity from shots to any live avatar upon which it is executed.

Figure 3.5 shows a second situation where a potential inconsistency may have been detected. Considering only the object affected by the event, at $S_0$ the Black avatar's state isn't modified by the event, while at $S_1$ the Black avatar's state is modified to 'dodged'. With a bit of insight into application event and state semantics, we could conclude that an avatar being able to 'dodge' or not is very likely to lead to significant divergences between states. This criteria is more refined that simply considering that any late event execution is a source of inconsistencies that can lead the states to diverge.

Finally, in Figure 3.6 the system reaches $t = 1030ms$, causing $S_1$ to reach the point where it can finally execute the event where White shoots at Black. This time, an event is executed on time at a state; thus, that would not, in principle, cause an inconsistency. However, with that execution it is possible to detect that $S_0$ is inconsistent. At $S_1$, executing the shot event on time results in White missing the shot while, at $S_0$, the execution had resulted in the Black avatar being shot. Again, this is detectable if we compare the effects of the event over game objects.

This example shows that there are several alternatives to determining whether a state should be corrected or not. Simpler ones need less or no insight into game data structures

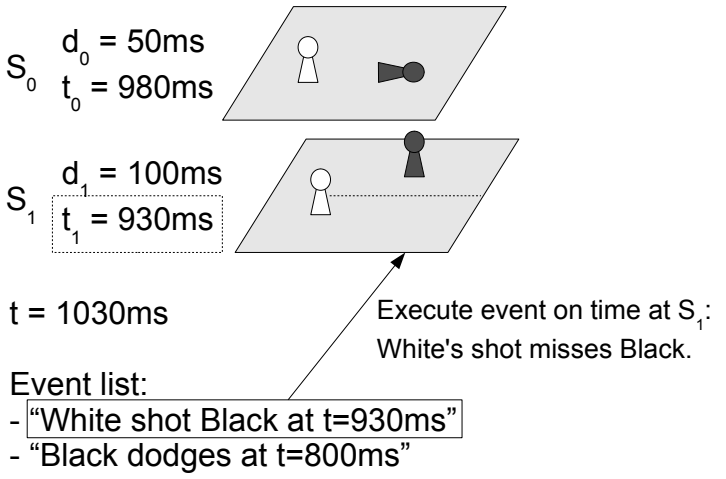


Figure 3.6: TSS inconsistency detection example (4 of 4): another detectable inconsistent situation.

and event semantics, while more sophisticated (and efficient) ones may need cooperation between a generic implementation of TSS and application code. In the next section we show how TSS corrects a state through a *rollback* procedure, regardless of the criteria chosen by the TSS or game implementor for determining when a state must be corrected through a rollback.

### 3.3.2 Fixing local state inconsistencies with rollbacks

As discussed in the previous section, a state may need correction. To correct a state, the TSS algorithm performs rollbacks, like other optimistic algorithms such as Time Warp. However, Time Warp rolls back to snapshots that are taken dynamically from 'the' local simulation state, while TSS keeps several 'live snapshots', which are versions of the simulation, running at a node and uses them as source of rollbacks instead.

Rolling back a state $S_i$ only makes sense if its immediately trailing state, $S_{i+1}$ is considered consistent. Also, the last trailing state can never be corrected by a rollback since it has no state trailing its execution. To correct $S_i$, the state of $S_{i+1}$ is first copied over $S_i$ ($S_i = S_{i+1}$). After that operation, $S_i$ will be representing the same simulation time of $S_{i+1}$, which is not intended. That is, the simulation time of $S_i$ is now $t_i = t_{i+1} = t - d_{i+1}$. To complete the correction procedure, $t_i$ must be brought back to $t_i = t - d_i$. To do so, all events in the event list whose timestamps are in the $[t - d_{i+1}, t - d_i]$ range are re-executed, in order, at $S_i$. The end result of this procedure is that $S_i$ *remains* representing the same point in simulation time as before, but now it has re-evaluated its state to more accurately represent the intended order and timing of event execution. This guarantees that all states in a TSS simulation are able to sustain a consistent simulation in the long run, at least as long as the last optimistic trailing state is never inconsistent.

### 3.3.3 Cascading rollbacks and the CPU cost of rollbacking

An incoming event may invalidate several states. As a result, the rollbacks can 'cascade'. Suppose that $N = 4$ and both $S_0$ and $S_1$ are detected as inconsistent after an event arrives and is executed (so far) at $S_0$, $S_1$ and $S_2$, while $S_3$ has still not reached the time

when that specific event 'happened'. The inconsistency is detected because, while $S_0$ and $S_1$ are consistent, $S_1$ and $S_2$ aren't ($S_0$ and $S_2$ aren't either, of course). Thus, the contents of $S_2$ are copied over $S_1$, and $S_1$ re-executes all events in the event list whose timestamps are in the interval $[t - d_2, t - d_1]$. After that, $S_0$, the leading state, has to be corrected as well. The rollback procedure then 'cascades': the contents of $S_1$, now corrected, are copied to $S_0$, followed by $S_0$ re-executing all events in the $[t - d_1, t - d_0]$ interval. Thus, each event is re-executed only once in a rollback, though more state copies are required.

In an ideal world, these rollback tasks should happen instantly. In that ideal setting, each mirror would always have, in its leading state, a real-time version of the game world which would be adequate for feeding updates of a real-time game such as Quake to players for rendering. In any case, the delay of the leading state, $d_0$, must be set to a small amount such as $50ms$, since $d_0$ is the minimum delay that will be applied to any event, considering the time it is issued (its timestamp). Thus, $d_0$ can be set, for instance, to a *minimum* network delay expected for inter-mirror communication, or some other small value close to it. The trailing states are then used solely as 'live snapshots' which can be used to restore consistency to the leading state, should the leading state diverge from the correct simulation outcome due to premature event execution.

But, in the real world, the rollback tasks can take too much CPU time to execute and stall other tasks such as regular command execution and communication. Thus, the TSS algorithm was evaluated by its authors in terms of amount of time taken to execute rollbacks, among other parameters. It was shown that, when implemented over the Quake game, TSS took around 1.4ms of real time to execute a rollback (CRONIN et al., 2004a). The results indicate that the technique is feasible, considering that game code designed and optimized specifically for TSS would perform much better.

The TSS paper also discusses optimizations such as avoiding rollbacks for some types of events which can be integrated into the state by performing some custom correction that is cheaper than the rollback and re-execution mechanism. We don't want these optimizations in FreeMMG 2 as we don't want the middleware to have insight on the meaning of events and their timestamps, nor do we want to call back application code to figure it out. However, if optimizing rollbacks turns out to be a necessity in practice then these can be developed in future works.

## 3.4 Baseline State Synchronization (BSS)

The Baseline State Synchronization (BSS) algorithm which we now describe is our adaptation of TSS to work in an environment where the simulation is to be performed by a group of potentially unreliable nodes which also may have no well-provisioned network to connect them. We will use 'peers' to refer to these nodes. As with the mirrored servers of the MS architecture, the peers that will now run the simulation may not be actually playing the game. The TSS paper also considered a scenario where the nodes running TSS are players. Likewise, BSS does not dictate whether the peers are playing the game or serving as proxies, or 'mirrors', so that other players can play.

The only definitive difference between BSS and TSS is that BSS maintains a *conservative* state on each peer. A minimal instance of the BSS algorithm should have at least two states: the *conservative* state and an *optimistic* state. The *optimistic* state is the leading state and, in principle, the application should set it to have a synchronization delay of zero, though that is not mandatory. The *conservative* state is the only and last trailing state. It orders events *conservatively*, meaning that it is never inconsistent since, in princi-

ple, the ordering and timing of events for the mirrored simulation in question are the only aspects that affect its consistency. By performing conservative ordering, no late events are ever received at the conservative state and thus the peer will never end up diverging permanently from the intended simulation outcomes. Thus, the conservative state runs the trivial stop-and-wait protocol described in Section 2.3.6, and the optimistic state just executes events as TSS does.

In TSS, all states order events optimistically. The guarantee that a mirror server never diverges permanently from the other mirrors comes from assigning a sufficiently high synchronization delay to the last trailing state. Since the mirror servers and the network that connects them are well-provisioned, estimating a sufficiently high synchronization delay such as 900ms may be enough in practice. However, in our new 'peer' scenario of BSS, we can not assume anything about the delay between the simulators. Thus, we enforce replication guarantee by conservatively synchronizing the last trailing state of each peer. Thus, the peers can be said to maintain replicas of each other, like in games such as Age of Empires (BETTNER; TERRANO, 2001) and like the peers in the FreeMMG architecture (CECIN, 2005).

In our original publication of our BSS idea (CECIN et al., 2006), which was an adaptation of TSS to serve peer-to-peer 'instanced games' similar to Guild Wars (ARENANET, 2006), we have also fixed other parameters and introduced other modifications. For instance, we set the synchronization delay of zero for the optimistic state, and we divided the simulation in rounds (steps) of fixed duration. But these are peripheral changes. BSS is thus essentially TSS with a stop-and-wait ordering at the last trailing state. For performance or other reasons, additional optimistic states may be introduced beyond the conservative and the optimistic states, or the leading state may have a synchronization delay, etc. Our earlier work (CECIN et al., 2006) shows that even using BSS to synchronize peers, with only the two default states, with a lossy and high-latency network, with a maximized roll-back gap size (between zero and the maximum possible synchronization delay), and executing rollbacks almost constantly (with no additional insight on application code), resulted in an average CPU load that is manageable. Thus, for this thesis and for FreeMMG 2, we assume that any BSS variant will not be prohibitive in terms of CPU, especially if games are written from scratch to fit well with it.

## 3.5 FreeMMG 2: a MMOG architecture based on BSS/TSS

The BSS algorithm is a central component of FreeMMG 2, and the goal of this chapter is to show where BSS fits into our MMOG architecture. Thus, we must provide an overview of FreeMMG 2. The architecture will be explained in detail in the next chapter.

FreeMMG 2 is an hybrid peer-to-peer and client-server architecture for MMOG support. Following the architecture classification of Chapter 2, it would fit in the *multiple arbiter* category. The focus of FreeMMG 2 is in achieving a significant reduction of server infrastructure while, at the same time, achieving a high degree of resistance against state cheats and state loss even in the presence of high peer churn, network-level attacks and large groups of colluding malicious peers.

FreeMMG 2 is a cell-based architecture which represents the space or terrain of a virtual world as a collection of contiguous cells, as illustrated in Figure 3.7. In the figure, the white pins represent any kind of in-game object. They may be player-controlled avatars, characters controlled by the simulator (or 'NPCs') or other game objects such as treasure and items, terrain features, buildings, etc. In FreeMMG 2, objects can interact across cell

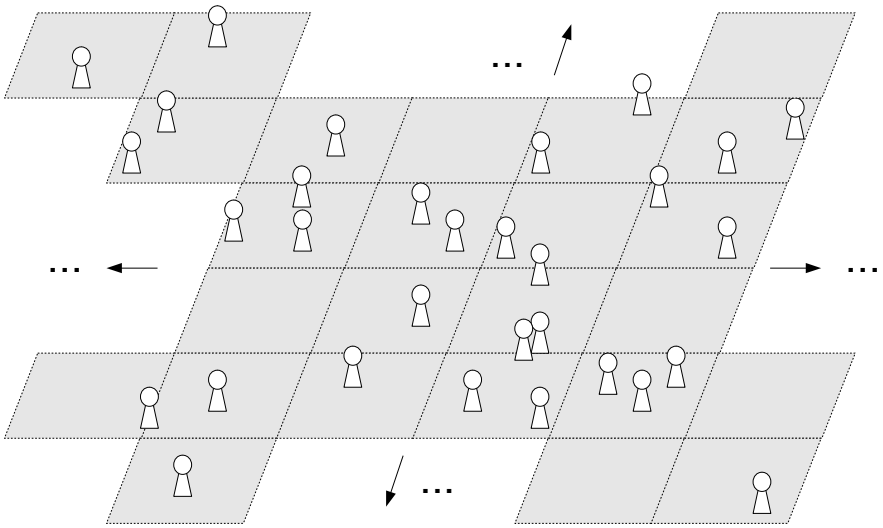

Figure 3.7: Section of a cell-based virtual world in FreeMMG 2.

borders. That is, an object in one cell can affect and be affected by objects owned by other cells. The quality (visual consistency, responsiveness, fairness, etc.) of inter-cell interactions is, unfortunately, lower than the quality of intra-cell interactions, that is, interactions between objects inside the same cell. However, to the game programmer, object interaction is the same whether objects are on the same cell or not; there is no need to write dedicated code to handle inter-cell interactions at the application (game) layer[2]. Finally, objects can be owned by one cell and be located at the terrain partition of an adjacent cell. Thus, each cell also has to hold in RAM any relevant state from its adjacent cells which can affect its own objects. The latter includes all the static terrain data of adjacent cells or, if RAM optimization is necessary, only the relevant portion (the nearest portion) of it.

To achieve decentralization and saving on server processing and communication costs, all cells in FreeMMG 2 are to be maintained (simulated) by client machines. As discussed in Chapter 2, to do that in a secure manner each piece of delegated game state needs to be replicated among several such untrustworthy client machines. In FreeMMG 2, each cell is a replication group and each replication group maintains all game objects or state variables that are currently *owned* by that cell. Thus, the bulk of game state is maintained by the cell replication groups[3].

Each client machine that acts as an active replica with arbitration power at a cell is called a *cell node*. A single client machine cannot maintain more than one cell replica since that would defeat the purpose of having multiple arbiters, though a single client machine may act as cell node at several different cells at once.

In FreeMMG 2, the cell nodes need not be players of the game. More precisely, the cell node process does not encompass game-playing capabilities. This does not rule out a scenario where one player (client) machine is running two processes: one player process, through which the player interacts with the game, and one or more cell node processes. However, if sufficient non-interactive volunteer nodes can be gathered to serve as cell

[2]The only exception is if the game programmer wants to apply special compensation for the lowered quality of inter-cell interactions. In that case, the game programmer can easily check the owner cell of each object at each cell simulator.

[3]Some state can be stored at the server. This is discussed in Chapter 4.

nodes and provide the game to actual players, the player nodes will have less processing and communication tasks to perform and thus the quality of the game will improve. If the network runs out of dedicated volunteer nodes then any idle resources on player machines can be used to provide the missing cell nodes. However, we assume the best case scenario where dedicated, non-playing volunteer clients are available to serve as cell nodes.

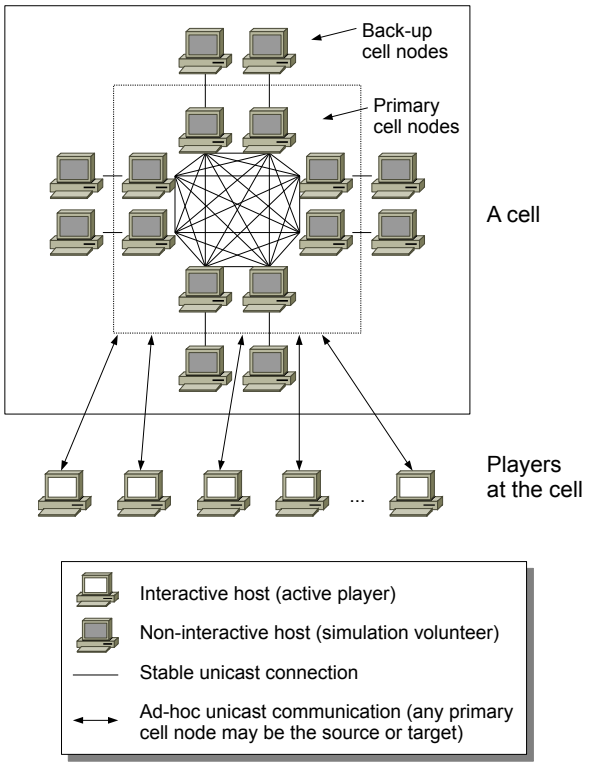


Figure 3.8: An isolated FreeMMG 2 cell simulation network.

Thus, a FreeMMG 2 cell resembles the Mirrored Server (MS) architecture which uses TSS to synchronize mirror servers. In FreeMMG 2, we instead use our adaptation of TSS to synchronize cell nodes. Like in MS, in FreeMMG 2 the cell nodes serve the game to player nodes. Figure 3.8 shows a network of cell nodes and player nodes that are running an isolated FreeMMG 2 cell simulation. The cell simulation works as in the MS architecture with TSS synchronization. The players in a FreeMMG 2 cell are the players in a MS architecture, the cell nodes are the mirror servers of a MS architecture, and BSS is used to synchronize cell nodes.

The main difference between MS and our cell is that the connections between cell nodes are like player connections, thus we cannot rely on any maximum latency to maintain consistency and instead employ a conservative trailing state. Event timestamping, ordering and execution can be as specified in the TSS paper, or it can be an hybrid of round-based and non-round-based execution as described in our earlier BSS work (CECIN et al., 2006). Also, we can no longer rely on IP multicast to optimize dissemination (broadcast) of events between the cell nodes, thus a full mesh of unicast logical connections is needed between cell nodes. Thus, the upload bandwidth requirement of cell nodes becomes an issue. The issue is aggravated by the currently low upload bandwidth available to consumers through popular technologies such as ADSL. That is an issue because we expect that volunteers in our peer-to-peer network will have the same average Internet connection

found among Internet users at large.

For completeness, notice that Figure 3.8 also shows that there are two kinds of cell nodes: *primary cell nodes* and *back-up cell nodes*. The primary cell nodes run BSS and serve the game as in the TSS/MS architecture, and connect to each other forming a full mesh of unicast connections. The back-up cell nodes run events using stop-and-wait (conservative) only since they don't have to serve the game to players, and they only connect to a single primary cell node to which they serve as a back-up. Contrary to primary cell nodes, the back-up cell nodes should impose a much lower load and thus could be more easily provided by the active players of the game. Finally, it should be noted that the number of primary and back-up cell node pairs, which in Figure 3.8 is set to eight, is actually a parameter configurable by the application.

## 3.6 Closing remarks

In this chapter we have produced a synchronization algorithm that is suitable for maintaining the state of the cells of a cell-based, decentralized and cheat-resistant MMOG architecture. That algorithm, BSS, is a tweak of the TSS algorithm (CRONIN et al., 2004a), whose target MS architecture resembles, topologically, our cell simulator. We have focused thus far in explaining how event synchronization is achieved inside a single cell. The next chapter will show how a contiguous cell-based virtual world is achieved, that is, how cells can affect each other's state, how players connect and disconnect from cells, etc. It will also show how players communicate commands and receive updates from cells, how to recover from failures and other essential functionality.

Our adaptation of TSS to intra-cell synchronization in a peer-to-peer MMOG left some open questions. The two main questions left are the following. First, it is unclear how the potentially significant latency and packet loss between cell nodes would affect audiovisual consistency, command responsiveness and fairness for players. Second, the upload bandwidth required by a cell node may be significant for current consumer-grade 'broadband' solutions such as ADSL, aggravated by the lack of support for IP multicast on the Internet at large. In Chapter 5 we will evaluate the bandwidth requirement and show that the bandwidth requirements of FreeMMG 2 are realistic. As for latency issues, these are not the focus of FreeMMG 2, though we tried to keep the number of hops within an acceptable range to support MMORPGs, at least. We discuss interaction latency later in this thesis, and much of latency minimization or compensation is left for game development, deployment decisions (e.g., limit game instance to a country or region) and future works to sort out.

# 4 FREEMMG 2 ARCHITECTURE

This chapter presents FreeMMG 2, a new model for supporting the distributed simulation of MMOGs. A FreeMMG 2 network presents a mix of client-server and peer-to-peer topologies. The system needs trusted servers to be maintained by a centralized game provider, but those servers are free from some important, CPU and network-intensive tasks that are often delegated to them when a pure client-server approach is used. This reduces significantly both the amount of server-side hardware and connection bandwidth that is needed to be maintained by the game service provider.

As shown in Chapter 3, FreeMMG 2 is cell-based. The main motivation behind the multiple-arbiter and conservative replication of cell state was security. By randomly choosing a significant amount of untrustworthy peers to replicate each cell, and taking other precautions elaborated in this chapter, we prevent collusion groups from forming in any cell with high probability. Thus a high resistance (though not immunity) against state cheats is achieved.

Our goal with this chapter is to provide the most important, core functionality that would be required of a peer-to-peer MMOG. The idea is that the functionality described in this chapter can be implemented as a generic 'FreeMMG 2 library' that provides support for peer-to-peer MMOGs. The information provided in this chapter does not explain, however, how a full game engine or a working game is built over the basic functions offered by FreeMMG 2. In Chapter 5 we show how FreeMMG 2 can be complemented by game or engine logic to provide higher-level functions such as reliable object transfers between cells, global consistency guarantees, resolution of conflicting updates and others.

For simplicity, we assume that the virtual world to be supported by a FreeMMG 2 engine can be modeled as a 2D plane and that it can be split into a grid of square cells, with each cell having exactly eight neighbors. Extending the model to other types of virtual world spaces (e.g. 3D) or other types of divisions (e.g. hexagons) and evaluating those extensions is left as future work.

## 4.1 Overview of a running FreeMMG 2 network

Figure 4.1 gives a top-level view of the main components that would be found on a functioning FreeMMG 2 network. Notice that the network is a hybrid of client-server and peer-to-peer topologies. The figure shows three roles filled by physical server machines: *master server*, *cell manager* and *cell filler*. The remaining machines are *clients* which are either machines where human players are playing the game or machines running daemon (non-interactive) helper processes which actually run the virtual world simulation.

There are a set of security-sensitive tasks that are the responsibility of the server-side of the network. This means that only machines trusted implicitly by all players

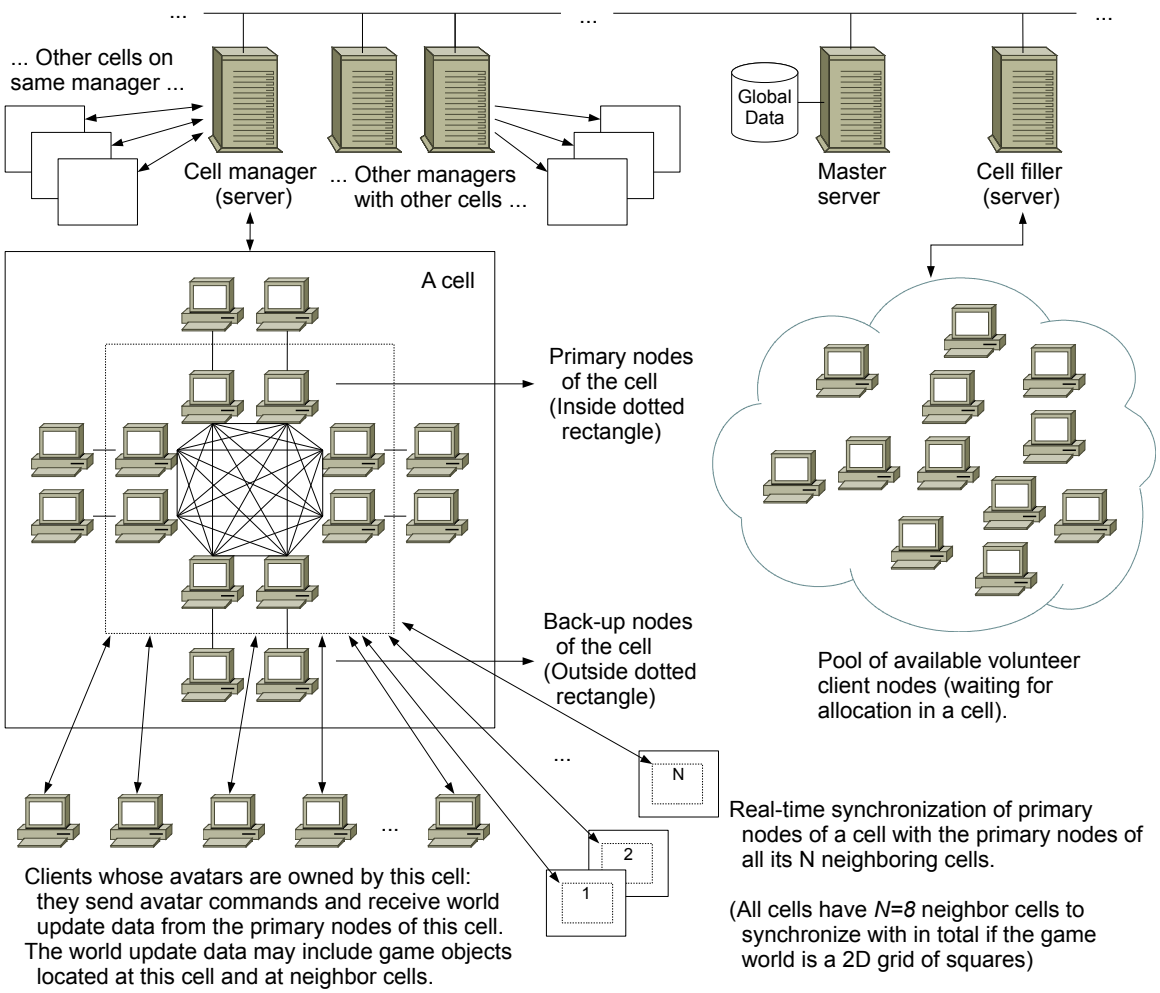


Figure 4.1: Overview of a FreeMMG 2 network

can execute those tasks. These machines are to be maintained by the game developer or publisher or a set of sufficiently trustworthy parties. In contrast, there are other tasks which require use of the client-side of the network. These tasks are CPU and network intensive and are off-loaded to machines outside of the game operator's control so that the network can scale to large player populations without incurring great hardware and communications costs for the game operator.

As can be seen on the figure, each server-side *cell manager* will handle a set of cells. It is not clear how many cells a cell manager could handle, and how this would be implemented exactly. For instance, a program (single process) could implement support for a single cell and accept socket connections to other processes that manage other cells, and a single server machine could run several cell manager processes at once. Another option would be implementing a single process with multiple threads, each thread managing a single virtual world cell. To avoid confusion, from now on we will use *cell manager* to refer to the process, thread or object that handles a single virtual world cell, and *cell manager server* to refer to a machine that handles several virtual world cells by running several *cell manager* processes.

### 4.1.1 Server-side overview

The main tasks carried by the server-side of a FreeMMG 2 network, as depicted in Figure 4.1, are the following:

1. Volunteer management: Schedule volunteer client machines to help in simulating (running) the virtual world cells;
2. Authentication: Authenticate client nodes;
3. Global state management: Store and serve any *global state*;
4. Vital cell state management: Store and serve the *vital cell state* of each cell, which is basically a server-side digest (reduced version) of the cell state that is being kept by the peers (see below);
5. Client tracking: Coordinate client nodes so that they can reach each other;
6. Cell fault handling: Manage the recovery of cells that have failed;
7. Certificate authority: Sign and distribute player certificates if a cryptosystem is used. That is, the server operator should act as the Certificate Authority (CA) of the game network;

Ideally, only one server machine would be employed in the network, and all server-side tasks would be delegated to that machine. Although this will probably work for a number of clients between 1,000 and 5,000, this would certainly not scale to a network with tens or hundreds of thousands of clients.

Volunteer management is achieved by a single cell filler machine depicted in Figure 4.1. This machine just has to accept a few incoming UDP/IP packets from the volunteer simulators when they start up and note their socket addresses. Since there is no persisted communication involved with this machine it is reasonable to assume that it could handle a few million clients easily. This machine is contacted by the cell managers when they need a fresh volunteer to fill positions as simulators on the cells that those machines manage (more on that soon). If there are several cell filler machines (unlikely), then the cell servers can choose one at random to request volunteers from.

Authentication and global state management are ideally handled by the same, single *master server* machine. The best possible example scenario that illustrates this task is when a player tries to connect to the game world. When a player connects to the game, it will obviously first have to authenticate with a server machine, so that real human players will only be able to play with their own avatars and not with other player's avatars. The machine that accepts the client's first connection, before authentication, is the master server. Player credentials information might be stored in a database local to the master server, as depicted in Figure 4.1.

After connecting and authenticating, the master server will have to decide to which cell manager this player will connect to. Let's assume that, when players rejoin the game, their avatars should reappear at the last position in which they were located prior to log-out. In this case, the master server might keep track of the cell ID where each player is currently located. This information might be stored in the same local database as player credentials information. So, the master server just looks up on which cell the avatar should reappear, and issue a command to the appropriate cell manager process to re-insert

the player's avatar object in the cell. At this time, the master server will also arrange a socket connection between the client process and the cell manager, which can then take over from there.

Vital cell state management is performed by cell managers. The vital cell state is essentially a reduced version of the cell's complete state which is kept by client machines (peers). Should the peers fail, the vital cell state can be used by the server to restore the cell. What goes into the vital cell state and what doesn't is decided by the application depending on what's important and what isn't. That is, a running virtual world cell will have lots of different state information, but some pieces of that state will be more important than others. For instance, the fact that cell $(X = 4, Y = 5)$ is currently populated by a specific player's avatar is more important than the exact position of that avatar in the virtual terrain. Avatar positions change rapidly so if positions were part of the 'vital' state of a cell, this would mean that every time the primary nodes of a cell changed the position of an avatar, then these nodes would have to exchange messages with a server-side machine (the cell manager server) in order to update that 'vital state'. Keeping vital cell state at servers allow cells that have failed completely, that is, that have lost their entire state, to recover partially. Vital cell state can also be used during object transfers (between cells) to achieve protection against *global inconsistencies* (CECIN et al., 2004). Object transfers and global consistency will be elaborated later.

Client tracking and fault handling will be presented later in this chapter. These tasks would, ideally, be carried out by the cell managers. However, the master server might also be used for this. The master server will at least forward new player clients connecting to the network to their initial cells and, maybe, the cell managers will be able to take over from there. Currently, this is not defined in the model. That is also true of the *cell filler server*, whose function could be integrated in the master server as well.

### 4.1.2 Client-side overview

The client-side of the FreeMMG 2 network is composed of 'client machines'. Any machine that is not directly owned or secured by the game provider is considered a client machine, meaning that full trust cannot be granted to any message or computation result reported by them. Client machines includes, for instance, the game player's machines, or server-grade computational resources donated to the network but maintained by third parties. In other words, if the central game provider cannot guarantee that the software running on a particular machine is the same genuine game software that was released, then that machine is deemed a 'client' machine.

There are three different kinds of processes that will run in client machines, as illustrated by Figure 4.1:

1. **Primary cell node**: A primary cell node stores a replica (copy) of a cell's current state. It synchronizes constantly with all of the other primary cell nodes of that cell, and with a single 'back-up cell node' (see below). It also synchronizes, less frequently, with the primary nodes of neighboring cells. If it has the bandwidth available, it will also serve one or more players with game updates (see below);

2. **Back-up cell node**: Similarly to a primary cell node, a back-up cell node stores a replica (copy) of a cell's current state. The difference is that the back-up copies the state of a single primary cell node. So, all primary cell nodes, whenever they update their own states, will send messages that bring its corresponding back-up simulator node to the same state. A back-up cell node's IP address is only known

by the server-side cell manager and the primary node that synchronizes it. If the primary node fails (e.g., due to a network-level attack such as a DDoS) then the back-up node can tell, to the cell manager, what was the current state of the primary node that just crashed, helping to avoid the loss of the cell state;

3. **Player node**: The player node is the interactive client-side process that allows human players to connect to the game and play it. It will continually collect user input and present updated versions of the player's view into the game world on the screen. As in any client-server on-line action game, the player (client) will send an almost constant stream of small network packets to the server, which are the command packets, and it will also receive an almost constant stream of larger network packets from the server, which are the world-state update packets. The difference is that in FreeMMG 2, the 'server' node will be a primary node of one of the cells of the game world.

The rationale behind this separation of cell nodes and player nodes is the following. In Chapter 3, we presented a way for a set of nodes to collectively keep a piece of data in such a way that a small set of nodes cannot possibly 'cheat' all nodes in the group in changing that data in an unintended way. In this regard, we have obtained a way for emulating a trustworthy node out of a set of untrustworthy nodes. Those nodes are the 'primary nodes' of a single cell in the virtual world. However, this does not cover network-level attacks, as a group of malicious nodes inside the group could attack the honest nodes and dominate all replicas [1]. To counter network attacks, a single back-up node is given to each primary node. And finally, being the primary nodes non-interactive, a player's machine will have to connect to the primary nodes in some way to inject its commands into the replica simulators and pull out information about the game world that immediately surrounds that player's avatar.

Since the cell nodes maintain the state of the virtual world, the player nodes have to synchronize with cell nodes in some way to play the game. In FreeMMG 2, each player node will be interested in synchronizing with only one cell at any one time, which typically will be the cell that currently *owns* its avatar. The player does not have to synchronize with several adjacent cells, even if its avatar is near the border of a cell, as inter-cell synchronization is handled solely by cell nodes.

To synchronize players with their cells, a FreeMMG 2 implementor has several options at their disposal. In a later section we provide several such options. We did not want to restrict the final game protocol or the semantics of events or updates. Thus, game implementors have several alternatives such as trading off vulnerability to protocol-level cheats for reduced interaction latency.

### 4.1.3 Delegating the simulation to volunteer client machines

To do away with the large bandwidth and computing costs at the MMOG provider's side of the service, the simulation of the virtual world must be distributed to machines that lie outside of the provider's domain. By 'distributed' we mean that this collective of

[1] That attack is addressed by having the server demand a minimum quorum of live replicas, out of the original set, before it will accept any information from a consensus gathered out of the live replicas. For example, if a cell is typically replicated by ten nodes, out of which nine fail (e.g., due to a network attack), no game server, in any context, should trust the result presented by the remaining node for a cell recovery. However, that means the state of the cell is lost. To avoid that, back-up nodes allow attacked primary replicas to still count by having their protected back-ups step in for the recovery quorum.

machines would be responsible for constantly recalculating the state of the shared virtual world and communicating with the game players.

The machines that lie outside the game provider's control will be called 'volunteers'. The presence of a substantial quantity of volunteers is a hard requirement for a game to be deployed on a network that follows the FreeMMG 2 model.

A functioning FreeMMG 2 will require many volunteer nodes, since even unpopulated cells need a minimum amount of cell nodes to maintain its state. The more volunteers the better. Even when the fabric of cells is fully filled with volunteer simulator hosts, it is important that a healthy dose of 'surplus' volunteers is readily available to replace the hosts being used. Surplus volunteers (not allocated in cells) are shown in Figure 4.1 as hosts inside a cloud.

So, it is in our interest to identify all possible sources of volunteering nodes. The possible sources of volunteer hosts that we have identified are listed below:

1. Machines of game players, as they play the game;
2. Machines of game players, while they are not playing the game;
3. Machines of users that don't even play the game and have chosen to volunteer for the game project;
4. Machines of users that don't even play the game but are being paid by the game publisher for their CPU/networking time;
5. Machines of ISPs (Internet Service Providers) or other interested businesses;
6. Some combination of the above;

Assumption number 1 is the most common for most works that need volunteers for a peer-to-peer game simulation network. Many proposals for peer-to-peer online gaming, including the earlier ones such as MiMaze, design a system where player machines connect directly to each other in some sort of overlay network [2] such that each player machine is connected to other player machines whose avatars are near its own. In other words, proximity of users in the virtual world translates more or less to direct, one-hop peer-to-peer socket communication on the resulting overlay network, if possible. Sometimes the use of IP multicast is proposed such that the virtual world is divided into areas and each area maps to at least one multicast group. In this scenario, when a player locally moves its avatar into a new area, it will start to broadcast its status, position updates, and other events to that multicast group so that other players in that area will be able to synchronize their views accordingly.

Assumption 5 has been considered by commercial projects. The Butterfly Grid could be seen as an attempt at bringing MMOG subscription revenue to ISPs: game server machines, running one or more games supported by the grid, would be installed at several ISPs, saving on most of the traffic that needed to be hauled from the ISP to the provider's central server.

Our proposal assumes a combination of items 1, 2 and 3. However, the primary source of simulation nodes for our proposed model (FreeMMG 2) is assumed to come from 2 and

[2] An overlay network is a 'logical' network that is typically formed by a mesh of socket connections between pairs of machines (unicast sockets) or groups of machines (multicast or broadcast sockets). Overlay networks usually feature multi-hop communications between hosts with some sort of routing logic, just like the packet-switched IP network that lays under them.

3: for some reason, users will want to support a particular MMOG project, and they will leave a non-interactive, daemon process running on their machines. This process will consume some of the CPU resources and, probably, most of the upload bandwidth of the machine (if it is a home ADSL connection), and will be analogous to leaving a machine running a P2P filesharing program such as a BitTorrent client that is seeding (serving) a file.

The obvious question that pops up is whether users will have any motivation to leave that daemon running on their machines. To answer this question, first we should note that there are already a number of situations where users are not required in any way to volunteer to a distributed effort but they do so anyway, which is the case with SETI@Home or any Internet computing effort. This would also be the case of most peer-to-peer filesharing that happens after a file is retrieved by a certain peer. And secondly, if these volunteers are also players of the virtual world (the more likely scenario) then it would be trivial to reward those players that contribute with in-game virtual wealth. This would provide extra incentive for users to contribute. Considering that other distributed computing projects get volunteers just by providing them with on-line rankings and such, we believe that the additional in-game wealth reward would potentially bring even more users to a 'distributed game computing' effort. We believe that, in the end, what will matter is whether the implemented game appeals to a large audience of players or not.

### 4.1.4 Software architecture suggestion for a FreeMMG 2 implementation

This thesis provides an abstract model. However, to be useful, the model needs to be implementable. Figure 4.2 shows a suggestion of a software architecture that is suitable for implementing a solution based on the model. What we provide below is an example; FreeMMG 2 can be implemented in several ways.

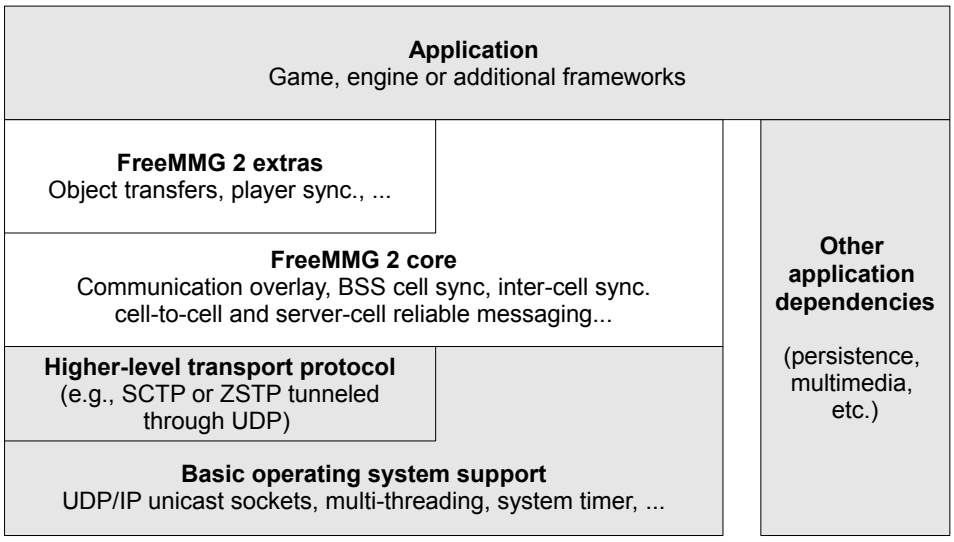


Figure 4.2: Envisioned software architecture for implementing FreeMMG 2 as library.

As the foundation of the FreeMMG 2 library there should be the common operating system support that is present in any networked machine such as a high-resolution timer, threads and UDP/IP sockets. If possible, implementing persistence (disk storage) should be left for the application to do through callbacks. For instance, a FreeMMG 2 master server might receive some state from a cell that is to be persisted. In this case, the

FreeMMG 2 master server code calls back the application equivalent of the master server code, passing the data buffer to be stored and any other relevant parameters such as the ID of the cell that is storing it. Thus, the application (whatever sits on top of FreeMMG 2) becomes free to choose any particular persistence mechanism such as a relational or object database or a simpler engine such as Prevayler (WUESTEFELD, 2003). Also, any multimedia support such as graphics and sound are also not provided through nor needed by FreeMMG 2, and should be managed directly by the application.

A FreeMMG 2 implementation will be much easier if a more sophisticated transport protocol such as SCTP is present to enhance UDP. What we need are mechanisms for delivering messages of arbitrary sizes between hosts reliably (with retransmission) and unreliably (without retransmission), and with our without ordering between different message streams. The main problem with UDP is that it doesn't provide reliable delivery. Also, it is more practical and efficient to encode multiple message ordering streams between two hosts in a single stream of UDP packets, instead of implementing different streams of messages between two hosts as independent flows of UDP packets. That is a good idea because aggregating data saves on packet headers and, additionally, the aggregation generates larger packets which offers more opportunities for gains in packet compression. And finally, flow control can be achieved easily by discovering an adequate packet rate and a maximum outgoing packet size for the UDP packet flow between any two hosts while these are constantly communicating.

Using the stream-based UDP enhancement or UDP itself, the *FreeMMG 2 core* layer implements basic communication and simulation functions. The core master server API has methods to create and remove cells, which causes the master server implementation to draw volunteer peers out of the pool and build the cell simulation overlays or to discard existing cell overlays. The cell nodes in each cell can run the BSS algorithm, synchronizing their simulation clocks and exchanging event packets at every simulation step, whose rate is fixed by server-side game code. The core also offers primitives for passing reliable and unreliable messages between cells and for passing reliable and unreliable messages between player nodes and cell nodes.

On top of the core layer, some higher-level functions that would be useful for many games could be provided in an *FreeMMG 2 extras* layer. For instance, we can assume that a MMOG is composed of relatively autonomous aggregates of state variables that all share virtual coordinates in the game world and thus move together around it: the *game objects*. Thus, it is necessary to have a mechanism for transferring game objects between cells. The object transfer transaction can be implemented using the inter-cell messaging support provided by the core. If the object being transferred can never be lost or duplicated in the virtual world, the transaction can be implemented by, additionally, logging the departure and arrival of the object at the respective server-side cell managers. Also, the extras layer could include some ready-made solutions for player synchronization. For instance, for a MMOFPS game, the layer could provide direct communication between player machines to reduce latency and increase consistency. But this would imply that players send authoritative updates to each other and to the cells. That is the kind of assumption that we want to avoid making at the core layer. We suggest that the core should have only irreductible functionality. Instead, the several options to player-cell synchronization could be implemented and placed in the 'extras' layer, together with the several object transfer options.

Finally, the application layer can be either monolithic game code or it can be broken down into engine and application or other combinations. As mentioned, the application

decides much of the functionality that is needed by the library such as data persistence. Alternatively, an implementation of FreeMMG 2 can be made less generic and provide a specific data persistence engine, a physics engine, a cryptosystem, etc. and thus become a game engine.

## 4.2 Basic protocol definitions

In this section we provide a basic framework for supporting communication between the different nodes that compose our distributed MMOG. These protocols are the base upon which the simulation algorithms are to be implemented. For instance, the BSS protocol described in Chapter 3, in which a replica disseminates events to all other replicas, should use the *primary cell node synchronization protocol* to implement that dissemination.

Each game based on FreeMMG 2 will have its own custom protocols. Thus, the specification provided here is incomplete, and it is intended to be used as basis for actual game protocols. The idea is that an implementor of FreeMMG 2 can provide a communication API using the specifications in this section as basis. Thus, the application should only have to configure parameters such as packet rates and data rate limits, and otherwise just call API functions to send and receive 'messages' (blocks of bytes opaque to FreeMMG 2) that are encoded to and decoded from the low-level transport protocol used (which in practice will be always provided by unicast UDP/IP sockets only).

The first subsection below discusses what type of base transport protocol is required by a FreeMMG 2 implementation. The following subsections discuss the higher-level protocols which enable the different types of nodes in the FreeMMG 2 model to communicate and thus implement all the required algorithms.

### 4.2.1 UDP-based low-level transport protocol

FreeMMG 2 needs a 'low-level' transport protocol, such as SCTP, that is tunneled through UDP and that enhances it, adding support for emulating multiple message streams over a single packet stream between two hosts. Each message stream of an UDP packet connection between two hosts should order messages independently and actually allow for configuring whether reordering at the receiver is required or not, among other stream properties.

Using that low-level, point-to-point protocol, the other higher-level protocols, described in the following sections, are implemented. Some details are abstracted, such as connection management. In FreeMMG 2, like in our previous work in MMOG support that uses only UDP unicast sockets, the typical client machine has to manage several active connections at once. This is an implementation issue which has to be dealt with by the model implementor. Generally, the application can be shielded from most of that complexity.

#### *4.2.1.1 Security-related assumptions over UDP sockets communication*

One of the main sources of security problems in networking lies in the relative ease of forging the source address of a packet. The higher-level cheat-resistant protocols in FreeMMG 2 require (or assume) that an UDP packet's payload and its header, including the source IP address, are authentic. Most protocols that rely on peer consensus in particular can be broken if an attacker can forge a 'vote' or consensus-building message in behalf of other nodes. There are several ways to address this. One would be to assume that most

such attackers will only be able to find out the IP address and port of the source they want to impersonate (since that is generally public information in a peer-to-peer system) and the IP address and port of the target to which they want to send the forged messages. That is, to assume that the attacker is not actually able to intercept traffic. Using this threat model, a simple solution is to have each pair of peers send each other a customized 'session password' (such as a random 64-bit or 128-bit number) which is then required in all subsequent return traffic, at least until a new password is given. That and similar 'weak' or 'security by obscurity' solutions may improve the situation but they can be broken easily if, for instance, the attacker is actually able to intercept packets. For a limited prototype they may be useful as a deterrent against the least motivated and equipped attackers, and one which is trivial to implement.

However, the definitive solution to the above problem is to add a proper cryptosystem to the model. Our general suggestion to an implementor is the following, which uses a combination of public-key and shared-key cryptography. First, a trusted server should act as the Certificate Authority (CA). Each party, CA and clients, generates a public-private key pair. The CA signs the public key of all clients using its private key. All validation of client public keys (or certificates) is done server-side and any client wanting to validate a new public key from other client should contact its server. Once clients have that secure way to validate each other's public keys, they can, pairwise, establish secure point-to-point connections in which they can negotiate shared (symmetric) keys for each individual peer-to-peer communication session. The shared key of each session is then used to efficiently authenticate the payload of outgoing UDP packets using a MAC (Message Authentication Code) algorithm. The MAC guarantees authenticity and integrity of the messages, but not secrecy, which isn't needed. Replay attacks can be prevented by having senders add a sequence number (serial number) to each outgoing UDP packet which is unique for each session. We assume that the computational and networking overhead of this system, compared with the costs of the rest of the functionality in the model, should be negligible.

However, MACs do not offer non-repudiation. So, if a party B in an (A,B) point-to-point channel forwards a message sent from A to B to a third client C, including the MAC in the message, then C cannot rely on B to determine that the message is indeed from A. Fortunately, that property is not necessary in most of FreeMMG 2's higher-level protocols. Our assumption is only that, pairwise, hosts (clients) can know that incoming packets are authentic, that is, the sender IP address really sent that UDP payload. Whenever there is a protocol or functionality which requires multi-hop communication in our overlay, we will explicitly state any further security-related assumptions, such as requiring peers to sign their messages using their public keys, which should be significantly slower than tagging messages with MACs.

### 4.2.2 Basic primary cell node synchronization protocol

The primary cell nodes of each cell form a group that runs the BSS synchronization technique described in the previous chapter. In this section, we define a concrete protocol based on the general BSS algorithm described in Chapter 3. The protocol described here is the same one described in our previous work (CECIN et al., 2006)[3]. The following discussion is in the context of a single cell, and by 'cell node' it is meant a primary cell

[3]This paper delves in more detail than the basic protocol, such as offering suggestions on how to compensate for the interaction latency. We have addressed this in Chapter 2. The reader can refer to the paper for more ideas.

node.

As we have specified earlier, to keep the *conservative state* synchronized, BSS executes the stop-and-wait protocol described in Chapter 2. Also, the optimistic simulator executes events such as, but not limited to, player commands, as they arrive. We have specified that, in a FreeMMG 2 cell, players do not necessarily send their commands to all cell nodes. However, the command of a player has to, eventually, reach all cell nodes, otherwise replication of cell state is impossible. Thus, cell nodes have to disseminate incoming player events to all other cell nodes in some way.

To minimize latency, a cell node broadcasts incoming events directly to all other cell nodes through sending network packets in unicast to each one of them. But, instead of broadcasting each incoming event individually, the cell node buffers all incoming events and dispatches them together to all other cell nodes in regular intervals. The event dispatching mechanism is common for both the conservative and optimistic replicas at each cell node. The only difference will be on how events are applied to each state.

Let $CTICK$ be the duration of a round[4] in the conservative simulator of each cell node. For example, $CTICK = 100ms$. Let $R_C$ be the round number of the current conservative state, starting at zero and increasing by one every time the conservative state is able to advance. Let $R_O$ be the round number of the current optimistic state, starting at zero and incrementing unconditionally by one every time $CTICK$ of real-time elapses.

Every time a cell node receives an *external event*, that is, one that is not sent or timestamped by another cell node, it will do two things with said event:

- Execute it immediately at its optimistic state;
- Place it in a 'pending input' list.

Whenever $CTICK$ of real-time elapses at a cell node, it will:

- Send the current contents of the pending input list, timestamped with the current value of $R_O$, through a reliable message stream, to all cell nodes, including itself. That message is the *round message* for that round and for that cell node;
- Remove all elements in the pending input list;
- Increment $R_O$;
- Generate an outgoing UDP packet to each other cell node (can use some hard-coded loopback mechanism to send to itself).

Whenever a cell node receives a round message from another cell node (or from itself), it:

- Places the message into a bi-dimensional array (let's call it *event array*) which is indexed both by cell node (the sender) and round number (the $R_O$ at the sender);
- Executes the events in the message in its optimistic state.

Whenever a cell node receives all round messages for round number $R_C$ and from all active[5] cell nodes, it will:

[4]This is synonymous with a simulation step or turn.

[5]An active primary cell node is one which has not failed. The mechanism that deals with failures, and which allows us to describe this and other algorithms without worrying about failures, is described in a later section.

- Execute all events in the event array which are scheduled for $R_C$ over the conservative state;
- Increment $R_C$;
- Discard sets of events from the event array pertaining to old rounds. It will be necessary to keep a window of old rounds in this array to allow the fault recovery algorithms to work (see Section 4.3.4). The exact size of this window is not an important definition as it certainly won't impact performance in any way.

With this basic mechanism, events are executed both optimistically and conservatively at all cell nodes. Flow control for this protocol is achieved in the same way that we achieve flow control in most of the other FreeMMG 2 protocols: by setting a fixed rate of outgoing packets at each primary cell node (one outgoing packet every $CTICK$ interval for each other primary cell node) and by setting a maximum size for each outgoing packet. Since UDP packet sizes gain from avoided fragmentation at the IP layer, it is probable that the value for maximum packet size will be limited to the Path Maximum Transmission Unit (PMTU) which, in theory, should be discovered but, in practice, will be set to around 1 kilobyte by a pragmatic implementor. If turn packets end up being larger than the maximum packet size allowed, there are several options to address this. The $CTICK$ can be increased, resulting in smaller packets since the amount of incoming input events remains constant. Also, multiple UDP packets can be generated in a single turn, each respecting the PMTU. Finally, the message retransmission strategy of the reliable streams can be tuned to generate less traffic, resulting in fewer outgoing packet data. What's important is that intra-cell synchronization can use up to a predictable maximum percentage of the node's download and upload bandwidth. This is taken into consideration when checking whether any given node in the volunteer pool will, a priori, be able to handle being a primary cell node.

This protocol defines some of the delay components of event processing. First, once a cell node receives the event, it will wait, on average, half of $CTICK$ to send it to all other cell nodes. After $CTICK$ elapses, it sends it in one logical network hop to all other cell nodes. Assuming that players in the same cell are served by different cell nodes, our scheme only introduce one additional logical network hop into the interaction delay equation. If interacting players are in the same cell and served by the same cell node, the components of interaction delay are the same ones of a client-server system (that is, of a star topology).

We haven't come up with definitive mechanisms for determining when rollbacks are necessary and how to perform them. We suggest the solution we evaluated in our paper (CECIN et al., 2006), which is to simply execute rollbacks continually and spread their execution over multiple rounds, using a third state as a buffer for executing them while they don't finish. Optimizing this is left for future works.

### 4.2.3 Extended primary cell node synchronization protocol

The *basic* protocol described in the previous section is vulnerable to the *inconsistency cheat* (CORMAN et al., 2006; GAUTHIERDICKEY et al., 2004b) and thus needs to be enhanced. Figure 4.3 illustrates the distributed lock-step[6] conservative simulator that runs on each cell being easily defeated by a single malicious primary cell node. The malicious

[6]We have also called it a stop-and-wait protocol. It means the same thing.

node makes the conservative replicas diverge by simply sending different command messages for different primary nodes during the same turn. That is known as the inconsistency cheat.

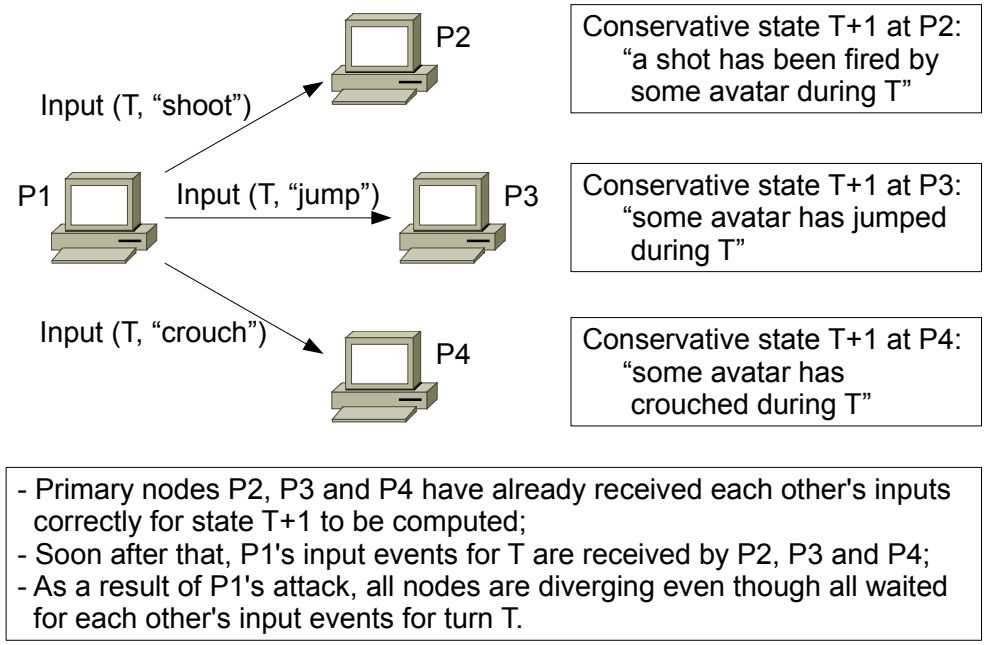


Figure 4.3: A single primary cell node causing the basic primary cell node synchronization protocol to fail

In an instanced game model, this could be addressed by simply discarding the problematic instance upon detecting later that the conservative states have diverged. For example, an infrequent exchange of state hashes is easy to perform and costs almost nothing in terms of bandwidth. However, our fault recovery mechanisms, described later in this chapter, cannot run if the conservative states are allowed to silently diverge from each other. We need to perform an additional tweak to the stop-and-wait simulator so that the attack shown by Figure 4.3 is blocked. This is accomplished by a modification that we called *turn flags*, which is explained below. Other mechanisms to address this *inconsistency cheat* already existed prior to our development of this (CORMAN et al., 2006; GAUTHIERDICKEY et al., 2004b) and our *turn flags* turned out to be quite similar to these. Thus, one should consider this section as an explanation of an existing technique rater than an original contribution of our work.

#### *4.2.3.1 The turn flags mechanism*

In addition to collecting all other node’s turn inputs received so far, each node of the lock-step simulator will also keep a list of turn input collection hashes received from each node, for each turn. A turn input collection is a set of serialized input events, each element from the set being the events sent by one of the nodes for that turn. By ‘input event’ we mean a full set of events that a single node sends with a single timestamp. The hash is calculated over a concatenation of all input events received so far for a turn, the concatenation being performed by sorting the serialized input events by node ID. That concatenation is the serialized form of a collection of all node’s inputs (events) for a given turn.

When a node detects that the basic lock-step criteria has been satisfied locally, that is, the node has received all inputs from all nodes for a given turn $T$, it concatenates the serialized form of all inputs using a deterministic ordering (node IDs), calculates a hash over that concatenation, and sends that hash for all nodes (including itself). It does not immediately use the node inputs it has to advance its 'conservative' state to simulation time $T + 1$.

Eventually, each node will receive one such 'full turn inputs hash' from every other node. Receiving that hash from all nodes is a definitive signal that the turn can advance. Thus we call that hash a 'turn flag', which is easier to visualize. In short, each node will now wait for an additional 'flag' from each node during any given turn, in the same way that it already waits for the 'input' from each node during any given turn. Once all nodes have waived their 'flags' for a turn, and all 'waived' flags match at all nodes, a node knows (locally) that all other nodes have received the same set of inputs that it did, and that updating its replica locally won't cause divergence with any other node, at least not due to malicious or faulty input broadcast (as illustrated by Figure 4.3).

Figure 4.4 shows how the attack illustrated by Figure 4.3 is detected through turn flags. To keep the explanation simple, the figure assumes that the cell has only 3 nodes (P1, P2 and P3), that all nodes have initially synchronized their states for an initial timestamp 'T0', and that the first simulation tick is about to be performed, which should advance the state from time 'T0' to time 'T1'. This example only concerns itself with the conservative state of the cell.

The first step (1) on Figure 4.4 shows P2 and P3 correctly broadcasting their events for turn T0. As an example, imagine that both P2 and P3 are issuing commands to move two different player avatars. As a result of this dissemination, both P2 and P3 have their own input and the input from the other node on their input lists. However, the lock-step simulation cannot proceed since P1 has not answered yet.

Before P2 or P3 decides that P1 is taking too long, which would cause the cell to fail, P1 decides to send its input for turn T0. So, in step (2) of Figure 4.4, P1 sends two different turn T0 inputs for P2 and P3. Imagine that P1 is also controlling a third, different player avatar, that P1 is telling P2 that P1's avatar has shot P2's avatar, and that P1 is telling P3 that P1's avatar has shot P3's avatar instead.

In step (2), P1 is hoping that, by receiving his input for T0, both P2 and P3 will be tricked in unlocking from the basic lock-step criteria and computing their next states for time T1. However, if P2 and P3 did that, both would compute two different outcomes, and their 'conservative' states would no longer be replicas. At P2, P2's avatar was shot by P1 on T1, and at P3, P3's avatar was shot by P1 on T1.

However, because of the additional 'turn flags' matching criteria, P2 and P3 cannot immediately update their states. Instead, on step (3), they compute a hash over their full event lists for T0, that is, what they would be executing if they were running a simple lock-step, and send the hashes to each other. Upon receiving each other's 'turn flag' (the input hashes for turn T0) they immediately detect a mismatch. The mismatch stems from the different inputs sent by P1 at step (1). Even if P1 had sent a turn flag that matched either flag (hash), it wouldn't change the fact that the flags from P2 and P3 don't match. Any hash mismatch during a turn is enough to prevent the next turn from being silently computed: there is no need to wait for all turn flags to arrive before a mismatch is reported. The only way to avoid a turn flag mismatch for a given turn is having all nodes exchange the exact same events (inputs) for the current turn.

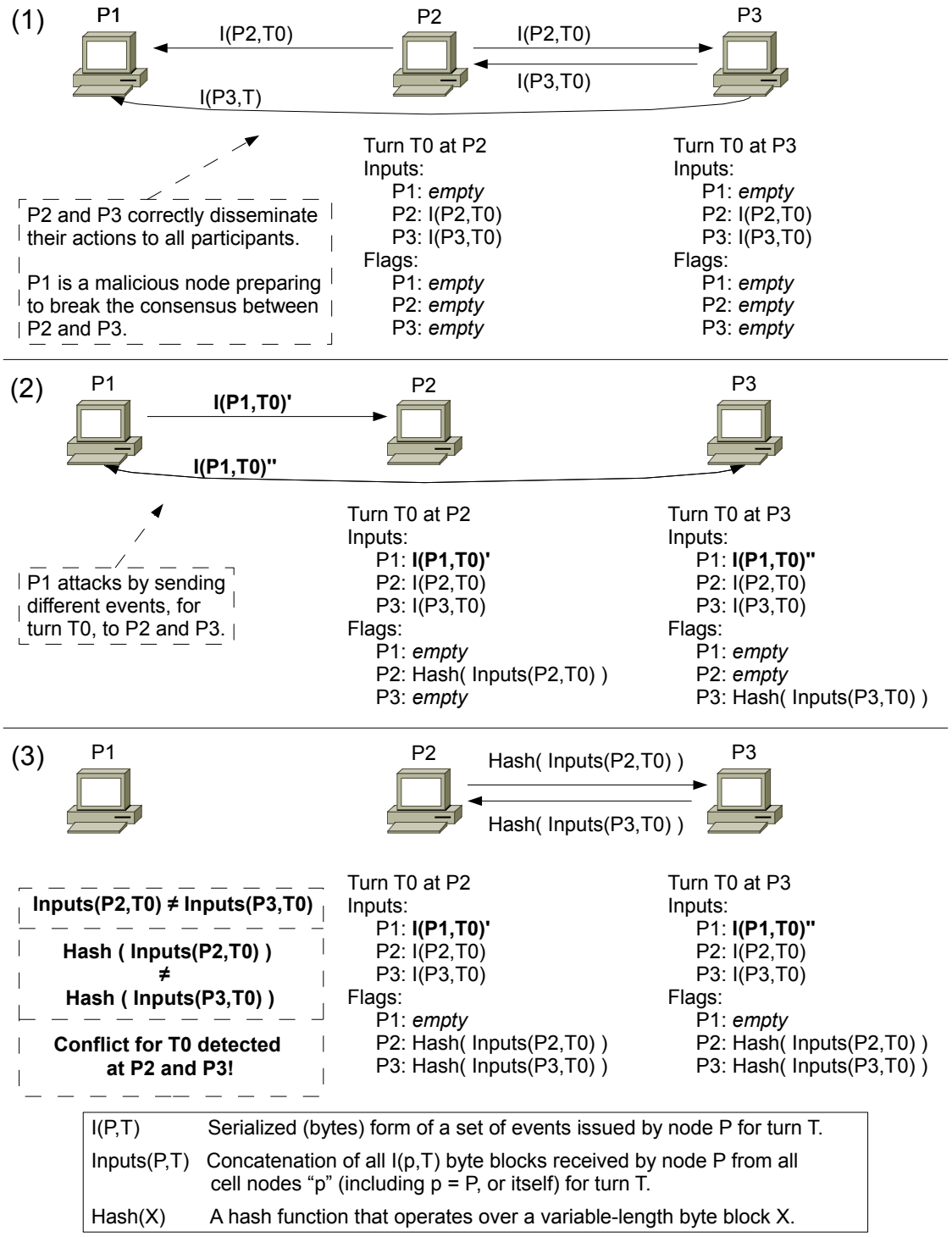


Figure 4.4: Example of turn flags mechanism detecting inconsistent input dissemination on a cell with 3 primary nodes

#### *4.2.3.2 Performance impact of turn flags*

The ‘flags’ mechanism certainly causes a negative impact on simulation rollback and re-execution as described in Chapter 3. By requiring more conditions to be met before the conservative state of each node can advance in simulation time, it will certainly widen

the simulation time gap between each node's optimistic and conservative states. This will increase the CPU cost of rollback and re-execution, and also decrease visual consistency for players due to the more drastic corrections performed by rollback and re-execution.

If this ever becomes a serious problem, splitting the conservative state in two replicas, 'optimistic conservative' and 'true conservative' could mitigate this. The 'optimistic conservative' would do simple lock-step, and the 'true conservative' would perform the additional 'flags' check. So, assuming that the 'flags' criteria is unnecessary most of the time, because almost all nodes are honest almost all of the time, then the rollbacks could be performed against the 'optimistic conservative' state, which uses simple lockstep as described in Chapter 3. However, when performing fault recovery, the 'true conservative' state would be used. The 'true conservative' state would use the 'flags' criteria and maybe something even more strict such as an exchange of hashes of the state after it is computed by the flags-approved inputs. In any case, this would result in three or four different state snapshots being kept on each cell node, instead of the proposed two 'conservative' and 'optimistic'. For now, we assume that the negative impact of the 'flags' mechanism over the optimistic-conservative gap is acceptable for the sake of a more simple cell simulator model.

#### *4.2.3.3 Limited responsibility of the turn flags mechanism*

This add-on 'flags' mechanism ensures that all cell nodes have the same inputs for updating their replica states locally. However, it doesn't guarantee that there isn't a programming error which introduces non-determinism in replica updating, which would cause divergences regardless of perfectly synchronized turn events across all nodes. Furthermore, malicious nodes will always be able to destroy their own replicas (and the ones at their back-up nodes) at will, while never sending discrepant event lists (inputs) during the same turn and thus never violating the turn flags.

So, the possibility that replicas will diverge will always be present. The 'flags' mechanism just ensures that a minority of nodes cannot destroy the majority replica consensus by just manipulating their own events. The fault tolerance mechanisms, described later, solves the remaining problem of dealing with, usually, a minority of divergent nodes and, rarely, a majority of divergent nodes. Perceiving mismatched turn flags locally is a failure situation which, like all failures detectable by a node, are notified to the cell manager through a simple network message (as will be explained in the upcoming 'Fault detection' section), and no further action is required by the cell

### 4.2.4 Backup cell node synchronization protocol

As hinted by the BSS explanation in Chapter 3, each primary cell node has a back-up cell node assigned to it. The back-up node is a mirror of the state kept by the primary node. The back-up node's IP address is only known by server-side machines and the primary node to whom it serves.

The purpose of the back-up node is twofold. First, it provides a means to discourage attacks of whatever nature against the primary nodes. The back-up cannot be attacked, since its IP address is unknown to any potentially rogue nodes. Once the primary node is taken down, the back-up node takes its place, essentially avoiding the loss of the replica state of the primary node. Unless the back-up node is part of a group of rogue nodes that is trying to perform state cheating on that cell, there would be no point in attacking

primary nodes besides vandalism [7]

The second reason for employing back-up nodes is preventing 'normal' node failures from invalidating the state of a cell. Normal failures include power outages, software crashes and any other source of ungraceful disconnection from the network. This is a problem because, whenever replicas are taken out from a cell, the server-side is left with a reduced number of working replicas of the cell state to trust. If that number is small enough, the server-side cell manager is presented with a tough choice between accepting what a small number of cells agree upon (window for state cheating) or scraping the cell state and replacing it with some arbitrary 'recovery' content (and watching disfavored players quit the game after this). If we ignore the existence of large mobs of nodes cooperating to sabotage the network for a moment, we can say that the adoption of back-up nodes pretty much filters out all unintentional failures from reducing the number of copies of the cell state. Even in the rare cases where a primary and its back-up node fail at the same time, this would only reduce the replica count by one, which will hardly be a problem if the original node count for the cell was healthy. This will be discussed in more depth later.

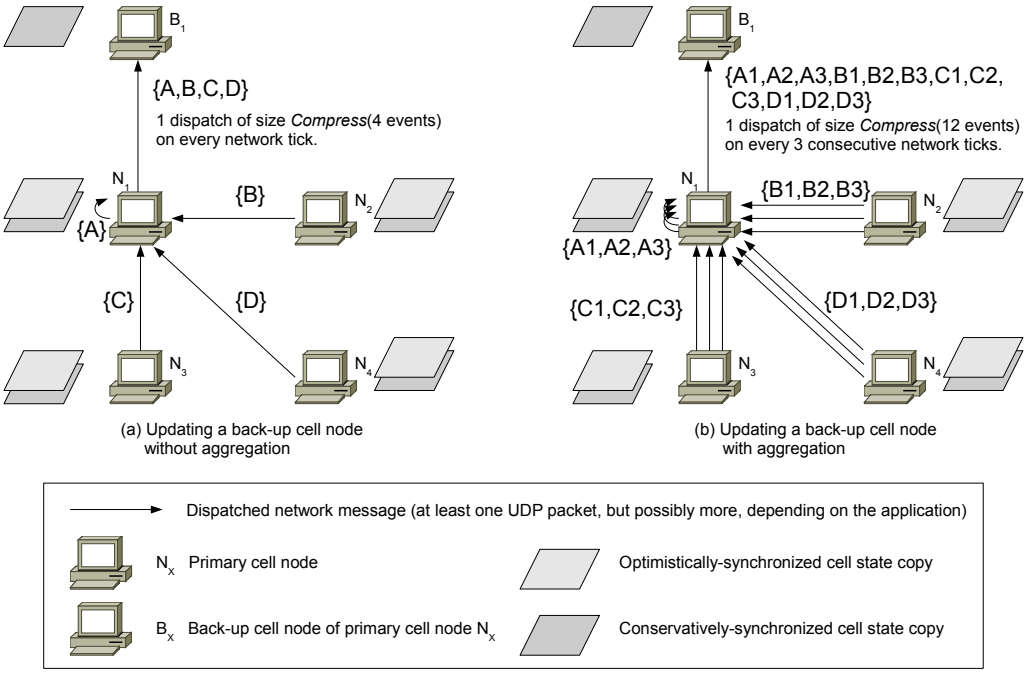


Figure 4.5: Example: using aggregation for back-up cell node updates in a cell of size 4

The back-up cell nodes only have to keep one simulator state, which is the conservatively synchronized one. Back-up cell nodes don't have to update any other node about the state of the cell, nor do they have to display the simulation state to a human player on a screen, so executing events optimistically would be pointless. The primary cell node that 'owns' a respective back-up cell node is responsible for forwarding all of the events it receives from the other primary cell nodes to its back-up node and also forwarding the

[7] Vandalism in a P2P simulator such as FreeMMG 2 can be seen as the same as 'poisoning' attacks against P2P file-sharing networks. In both cases, the goal of the attacker is to disrupt the network itself, not gain any advantage inside the network, such as cheating in a game simulator or downloading more data from a file-sharing service.

events it generates (broadcasts) to the other primary cell nodes.

Updating the back-up cell node takes significant upload bandwidth from the primary node. If there are 10 primary nodes in a cell, this means that, on each network simulation tick of the cell (which we defined as $CTICK$ earlier), a primary node will receive, on average, 9 events from each remote primary cell node, and it will have to forward 10 messages (nine remote plus one from itself) worth of bandwidth to its back-up cell node in a single network tick's time. This might be seen as a serious problem. However, since the back-up cell node has no time constraints to receive the events, since it won't execute them optimistically, the primary cell node can reduce the bandwidth usage significantly by aggregating a few turns worth of events into a single UDP packet (saving on headers and routing) and also compressing the resulting packet before sending it to the back-up node. See Figure 4.5 for an example considering a cell of size 4 (cell with 4 primary and 4 back-up nodes).

However, waiting too long to dispatch events to the back-up node can have a negative impact on the failure recovery procedures of FreeMMG 2. There is some gain to be obtained with aggregation here but a packet certainly won't be held for more than a few seconds. This may give a few tens of turns worth of average-sized events forwarded on a single packet, which might represent significant savings due to aggregation and, more importantly, in data compression. Evaluating those gains is left as a future work.

### 4.2.5 Inter-cell synchronization protocol

One of the great advantages of replicating simulation among cell nodes is that there is no need to disseminate potentially expensive *update* messages between them. Instead, only a few events or commands are exchanged, usually related to player avatars. If cell nodes had to exchange updates for all objects then, for instance, a horde of NPCs or other AI-controlled objects in a cell would generate a huge traffic to allow the cell nodes to synchronize. This would probably be coupled with a scheme where the AI objects are split evenly among cell nodes (a 'partitioning' distribution scheme). But, with our conservative ordering of events coming into the cell, which are the only source of non-determinism in the simulation, we can ensure that the simulation is deterministic and that the state is effectively replicated across all cell nodes. With this, we can guarantee that the AI-controlled objects will take the same decisions on all cell nodes and no messages concerning changes in their state have to be exchanged. This allows a cell nodes to multicast its small BSS 'round' packets directly to all other cell nodes through unicast, and we are able to compensate the lack of IP multicast without consuming unrealistic amounts of upload bandwidth from cell nodes.

However, the drawback of synchronization schemes such as the stop-and-wait protocol in BSS is that they aren't scalable. Waiting for *all* replicas is always necessary. One way to scale a cell beyond ten or so replicas would be to employ ALM. In this scheme, a node doesn't have to send, through unicast, its round packet directly to all other cell nodes. Instead, it sends it to a 'neighborhood', which then becomes responsible for disseminating it further, and so on. The drawback of this is the increased latency which, in TSS/BSS results in an enlarged distance between the conservative and optimistic states, which aggravates the potential problem of CPU usage and the wall-clock time needed to perform rollbacks. Also, latency is never good for games, and achieving cell scalability by introducing multiple hops of communication between cell nodes is simply not a good idea if we ever hope to achieve support for MMOFPS games with future works that don't completely change the architecture. So, the FreeMMG 2 cell is not scalable by design.

That is on purpose.

This leads us to the problem of inter-cell synchronization. So far, two juxtaposed cells simply won't interact at all. There is no way to engage two nearby cells into conservative ordering, for that would simply mean that the cells are merged and the cell scalability (or lack thereof) problem appears. One possible solution would be to run other larger conservative synchronization groups, in parallel, that encompassed several cells. For instance, all cell nodes of two adjacent cells engage in stop-and-wait, but at a reduced rate. This solution creates more problems than it solves. The actual solution we propose for FreeMMG 2 is described below.

The solution is to let adjacent cells exchange update packets. At every $CTICK$ network or simulation tick of a cell $O$, any primary cell node of an originating cell $O$ sends a single update packet to any (chosen by the sender, e.g. a circular sequence) primary cell node of an adjacent destination cell $D$. The receiving node treats the incoming update in the same way it treats incoming player command packets, with the exception that incoming cell updates are *authoritative updates*, that is, they modify the shared state at the receiver unconditionally. The source of these updates is the optimistic state of the sending cell node. That is necessary because the resulting inter-cell synchronization scheme will have a significantly increased latency if compared with intra-cell synchronization, and using the conservative state of $O$ as source of inter-cell updates would aggravate the situation.

The 'cell tick' ($CTICK$) periodic event happens asynchronously at each primary cell node. However, due to clock synchronization between cell nodes, we can assume that $CTICK$ happens at the same time at all primary cell nodes. When it happens, each of the $N$ primary cell nodes at a cell $O$ will send their small round packets to all other $N - 1$ primary cell nodes of $O$ and now, additionally, send large update packets to its neighbor cells. Since there are potentially malicious primary cell nodes in each cell, we recommend that a cell $D$ which is adjacent to $O$ should always receive an update from the next primary cell node of $O$ in a circular sequence. In this way, $D$ is never bound to a continuous stream of bogus updates from any of its adjacent cells and discrepancies between different updates from the same originating cell could be detected at $D$ by the application or other cheat detection heuristics.

Let's assume that the game world is a 2D rectangular grid, and that $N = 8$ primary nodes in each cell. In this case, each cell will have eight (8) neighbor cells. In this scenario, at every $CTICK$, each primary cell node of a cell $O$ will be responsible for sending exactly one update packet to one primary cell node of one of the adjacent cells. That is, at every $CTICK$, we have eight primary cell nodes and eight neighbor cells, thus each primary cell node becomes responsible for updating one adjacent cell. However, if $N \neq 8$, then adjustments are needed. For $N < 8$, either some of the neighbor cells of $O$ won't receive packets on each tick of $O$ or some of the nodes at $O$ will send more than one packet at each tick of $O$. For $N > 8$, some of the nodes at $O$ will not send packets at each tick of $O$.

This specification results in a regular and predictable amount of uploaded UDP packets by each cell node due to inter-cell synchronization. If $N = 8$ in a world that is a rectangular grid of cells, then inter-cell synchronization costs just 1 additional UDP packet to be uploaded per network tick, at each individual primary cell node. The payload size of this UDP packet will vary. It can go from zero bytes if there are no game objects near the border that divides the two adjacent cells being considered, to a very large payload if there are hundreds of dynamic game objects moving along the border that divides

the cells. Since hot-spots are to be expected on cell borders as in any place of the virtual world, there must be a mechanism to deal with this. A combination of packet-size cap, a message prioritization scheme that increases the score of pending messages with time so that they don't starve in the queue, and AoI filtering will have to do the job. The latter should be provided by the application.

The UDP packets exchanged between cells have the same stream-based point-to-point protocol described earlier over them. This means that a primary cell node will have the state for such protocol (such as pending messages to re-send, acknowledgements, etc.) for each primary cell node of each cell. That is, a primary cell node is always actively communicating with $N * M - 1$ nodes, where $N$ is the number of primary cell nodes in each cell and $M$ is the number of neighbor cells that all cells have. However, there is a difference: all logical links between two primary cell nodes of different (adjacent) cells will require very large timeout values. Considering two such cell nodes $P_1$ and $P_2$, the interval between $P_1$ sending two packets to $P_2$ is $CTICK * M * N$. Assuming $M = N = 8$ and $CTICK = 100ms$, that results in 6.4 seconds between packets sent from $P_1$ to $P_2$, and vice-versa. That is assuming that $P_1$ alternates the destination node at $P_2$'s cell fairly (circular selection).

Due to this delay, doing reliable or ordered delivery of messages across cell borders becomes inefficient. Thus, FreeMMG 2 itself only uses message streams that do not guarantee neither the delivery of messages nor their order for inter-cell communication. However, if the application requires it, inter-cell reliable or ordered messaging between pairs of cell nodes is available. The application is not limited to the UDP packet exchange described here, and it can make cell nodes communicate in any way desired. What we describe here is a level of functionality which makes the architecture compatible with current consumer-grade broadband technology. In other words, we want the ADSL users that have only 256 Kbps of upload bandwidth to be able to participate as cell nodes as well.

In this section, we have described how cells can send object updates to their adjacent cells, allowing adjacent cells to 'see' each other's objects in real time. In this way, a player avatar at one cell is able to see objects that are owned by its current cell and also by all the adjacent cells, though with some added delay and thus lowered visual consistency. This scheme leaves two problems open. The first one is how to transfer object ownership between cells. As objects move into other cells, it becomes necessary to hand them over between cells or else objects would only be ever perceived by its original spawning cell and its immediately adjacent cells. The second problem are update conflicts. In the scheme above, we are implicitly allowing objects to be owned by one cell and be located at an adjacent cell temporarily. That is a problem because, a 'foreign' object can be ordered, by its owner cell, to move over a local object. The local cell simulator will then have to resolve the conflict, which isn't trivial because, sometimes, the foreign update will need to be honored and at other times it will not. These issues are addressed in upcoming sections.

### 4.2.6 Player-cell synchronization protocol

The FreeMMG 2 core should provide a set of basic communication primitives that allow several different schemes for synchronizing player nodes and primary cell nodes. Ideally, any player node should be able to send packets to any primary cell node in the system. Then, game code would be responsible for filtering out irrelevant packets. For instance, consider that a player sends a command to a primary cell node that orders its

avatar to move. The cell node receives the network message and quickly discovers that the player has no means to interact with the cell in question. For example, the cell has no avatar object for that player. Then, the cell node can simply discard the packet. Thus, the communication primitives to be provided by a FreeMMG 2 core layer are simple communication primitives that allow for sending messages (packets) to specific primary cell nodes in the system. In other words, it just forwards the API of the low-level point-to-point protocol. Some minor improvements can be made in the process, such as changing the destination address format from an IP address to a node ID in the system. Other than that, the core only provides raw messaging between player and primary cell nodes. There is no need for prior authorization of communication, interest management or the like at this level. In following sections, this primitive messaging mechanism is used by some specific player-cell interaction protocols which should be able to support several different games with different requirements.

### 4.2.7 Client-server protocol

All client nodes must be able to connect to and communicate with server machines while they participate in the system. These client-server connections should remain idle on average, though significant communication can occur between a server and a client on some occasions. For example, if a cell fails in a way such that its state is unrecoverable, the server-side process responsible for managing the cell must rebuild the cell state and transfer it to a full new set of replicas (this is explained in Section 4.3). Another example is if two cells start performing numerous *reliable object transfers* between each other (explained in following sections).

We suggest that the low-level socket protocol used for this be the same used for the other protocols. Like the player-cell communication support at the core, the client-server support provides just plain point-to-point messaging. That is not so important. What is important is how the potentially several server machines allocate processes that communicate with subsets of client machines. We have already defined that cell nodes should communicate with a cell manager process on the server-side. These processes should be either allocated statically to machines or the available server machines should distribute the load of managing cells dynamically. The players, on the other hand, could connect to the cell manager processes of the cell that owns its avatar. Alternatively, players could be assigned to player manager machines which would each manage a maximum amount of players. In any case, the bottleneck would be, probably, the shared data store upon which the separate manager processes would operate.

## 4.3 Fault tolerance at cells

All of the client and server machines on a FreeMMG 2 network may experience a myriad of different software or hardware failures, both due to unintentional failures and intentional ones, such as attacks. Providing full fault tolerance coverage, in detail, to the whole FreeMMG 2 network is not the goal of this thesis.

We do not touch on any kind of server-side fault tolerance. We assume that server-side fault tolerance can be performed as an add-on work. For instance, if a machine running several cell manager processes crashes, those processes should be restarted in another machine. Several practical issues arise, such as reuniting the client-side cell nodes with the new cell manager. It is clear that recovering from server-side failures is not a trivial task, but it is certainly a task that can be detailed as future work or left for implementors to

resolve. For now, we can assume that the server-side infrastructure is sufficiently robust.

As for the client-side, we are concerned with failing nodes. We are not concerned with player node failures, as those can freely crash or drop out of the network with no effect on the cell infrastructure.

So, in the context of this thesis, we are specifically concerned with the client-side cell nodes (primary and back-up) failing. In our model, virtual world cells are served by untrustworthy client nodes which may drop in and out of the network frequently, which causes player serving outages and temporarily reduces the amount of replicas in a cell, potentially weakening it against state cheating. And when cell nodes are not dropping in and out, they might be trying to cheat the game, or they may fail unintentionally, for instance due to programming bugs.

So, in this section, we address our only real fault tolerance concern which is the cell failure. A cell is actually a set of hosts, so we have first to define what makes that set fail, and how and where the failure is detected. So, we first address fault detection, or when does a cell fail. Then we propose a mechanism for fault recovery, which is managed by the cell manager. This mechanism eventually restores the cell to a working state.

Our two main concerns in regards to cell fault tolerance are, in order of importance:

- Minimizing or eliminating the possibility of a successful peer collusion for state-cheating;
- Minimizing or eliminating the possibility of a complete loss of the client-held cell state, forcing the cell manager to restore the state from its 'vital cell state' record. This happens if all cell peers die at the same time (very unlikely) or if the remaining amount of cell peers is so low that it opens the possibility of state-cheating (more likely).

The fault tolerance mechanism's design stems from those two top-priority issues. Having those settled, we try to address the following secondary concern:

- Minimize the average cell serving outage for players due to fault recovery. In other words, recover the cells quickly so that players stay only a few seconds out of gameplay, instead of minutes;

Before detailing our proposed mechanisms, the next section discusses how far we could ever get in regards to attending to the two main points: state-cheating resistance and avoidance of cell state loss.

### 4.3.1 Limits of fault tolerance effectiveness in a FreeMMG 2 network

In this section we set some upper limits on how far can we could ever get on fault tolerance, considering the FreeMMG 2 architecture described so far. By 'effectiveness limit' we do not mean how transparent a cell fault recovery procedure can be, or how much coverage does our fault detection is. Rather, we want to check, assuming the best possible detection and recovery mechanisms, how well we can steer away from state-cheating and state loss while doing fault recovery.

First of all, state cheating, which is an intentional failure, cannot be detected automatically by the FreeMMG 2 model described in this text *if all of the primary cell nodes are colluding* to subvert the state of the cell into something illegal which favors the player accounts of those colluding clients. This cannot be solved unless some other method is

used to detect illegal alterations. For instance, adopt an heuristic that tries to deduce, using knowledge of the game rules and some degree of confidence, if a cell has grown too much of in-game, 'virtual resources' in a period of gameplay that is too short, much like the tools that governments have to detect tax fraud.

Secondly, *the loss of the entire state of a cell is always a possibility*, the question being only the probability of that happening on the network as a whole under normal circumstances and under varied styles and magnitudes of intentional attacks. Consider that it might be rare, but all cell nodes, primary and back-up, can be shut down at the same time. This implies that to offer a 100% guarantee of global consistency, you need a trustworthy and available machine to keep the cell state. In other words, you need one of the cell servers for that. This brings the central consequence that, in order to achieve any degree of decentralization and also guarantee 100% consistency, we need to split the cell state in two parts: the bits that are essential to maintain global consistency and the non-essential bits. The essential bits are synchronized with the servers and the non-essential bits are not. In the event that the entire state of a cell is lost because the peers' states cannot be used (for whatever reason) then the cell state is reconstructed from the essential bits synchronized at the server-side and those bits being the essential part of the cell state that guarantees global consistency, the game world as a whole can have some invariants that are never violated. This also means that all of the gains related to client-server traffic reduction in FreeMMG 2 are related to the proportion of essential to non-essential bits inside the cell states. Thus, the previously mentioned 'vital cell state' feature depends on infrequent, but important (vital), updates being sent from the cell nodes to the cell manager.

Those two are problems that the FreeMMG 2 model, as designed, cannot fully solve, even if its parameters are tweaked to whatever values (e.g. set `N=100` cell nodes per cell or whatever high value).

In FreeMMG 2, the last line of defense against collusion cheating in our model is making it unattractive or unrewarding for groups of coordinated cheaters to even attempt cell collusions in the first place. As will be seen in the experiments chapter, a very large group of colluding nodes (in proportion to the total amount of volunteers) have to serve as cell nodes for a long time on average before they get a shot at subverting a cell. We can argue that the good that this large set of nodes does to the network offsets any cheating that they can do. So, we believe that, in the end, this preventive 'last line' can be effective.

Also, the last line of defense against complete loss of a cell state is synchronizing a low-fidelity version of the cell on the server-side cell manager using the *vital cell state* feature, as suggested in earlier sections. We also believe that this last resort measure can be effective. It complements the main approach, which is to make it highly unlikely that a cell's running state is going to be lost in the first place.

In short, collusion cheating and complete loss of state are the two most catastrophic events that can happen to a cell. We make it highly unlikely for them to happen in the first place. For collusion, we prevent it by making it very unrewarding to be attempted, or forcing the attempters into contributing to the network first, which compensates the damage. The game designers can even lay their own game-dependent checks over those basic mechanisms to further discourage collusion. For complete loss of state, we recover from the highly unlikely cell state losses with a game-dependent recovery procedure based on game-dependent 'vital cell state' that is synchronized at the server-side cell manager in a game-dependent way, thus effectively mitigating this (already rare) damaging situation.

### 4.3.2 Fault detection: cells and cell nodes

All of the client and server machines on a FreeMMG 2 network may experience a myriad of different software or hardware failures, both due to unintentional failures and intentional ones, such as attacks. The FreeMMG model is not designed to treat each and every type of individual node failure differently. Rather, we are only concerned on detecting if a node is behaving in a faulty manner. The individual reasons that may lead a cell node to fail will be discussed in passing through the text but we are not concerned in building an exhaustive list.

A cell 'fails', that is, a cell failure is officially 'detected' when its cell manager decides that it has failed. The cell manager decides (or 'detects') that a fault has occurred in a cell if any of the following conditions are met (the list is intended to be complete):

- It receives a `FAULT` message from any cell node. When a cell node observes any kind of local faulty behavior, or when it decides to try to slow-down or sabotage the network for whatever reason, it can send a `FAULT` message to that cell's server;
- While communicating with a cell node, the server detects a local communication timeout or error on the socket connected to that node;
- While communicating with a cell node, the server receives a network message that violates the protocol (invalid message) from that node.

A cell node may send a `FAULT` message to the cell manager for any number of reasons. As discussed earlier, we are not concerned in enumerating all possible types of individual node failures. The sending of a `FAULT` message to the server may be due to the cell node detecting a fault locally, or because the node is malicious and wants to somehow cause harm to the network. The following is a list of events that a cell node can perceive programmatically and, when perceived, are considered as failures:

- A primary node loses its overlay network (logical) connection to any of the other primary nodes in the cell, its own back-up node, or the cell server. This is detected by a middleware-defined (or game-defined) timeout rather than the sockets library or operating system;
- A back-up node loses its logical connection to the primary node or the cell server;
- A cell node receives a malformed message from another cell node or the cell server which it cannot just discard and ignore;
- A cell simulator hits an exception handling point or software crash where it does not know how to recover from that situation;
- A cell node is unable to keep up with the other simulators on the cell: not enough computing (CPU) resources to run the conservative and optimistic simulator, and the rollbacks (see Chapter 3 for an analysis of this problem);
- A cell node is unable to keep up with the other simulators on the cell or neighbor cells: not enough bandwidth for clearing the backlog of pending messages to send. This increases latency up to a point where this node will harm the network as a whole;

- A cell node detects that the replica (conservative) simulation has gone out of synchrony. Although the conservative simulation requires deterministic inputs, a game programming error can be inadvertently relying on non-deterministic values, pulling the replicas out of synchronization. This can be detected through the exchange of conservative state checksums, and can be denounced by a `FAULT` message.

The `FAULT` message has only one (optional) parameter, which is the ID of a single cell node which is being accused by the sender cell node of having failed [8]. If no such parameter is provided, then the cell manager assumes that the sending node is accusing itself of faulty behavior (same as sending its own ID as parameter).

During fault detection, primary and back-up nodes are treated differently by the cell manager. Below is a list of all possible combinations of sender (accuser) node and reported (blamed) node in an incoming `FAULT` message, and how the cell manager should interpret the incoming message:

- Accuser (primary or back-up) is blaming itself: that node is considered faulty;
- Accuser is primary, blaming its own back-up node: the back-up node is considered faulty;
- Accuser is primary, blaming another primary node: both primary nodes (accuser and blamed) are considered faulty;
- Accuser is back-up, blaming its own primary node: the back-up node is considered faulty;
- Accuser is blaming an invalid node ID (e.g., a back-up node blaming a primary node not its own, or a primary node reporting a non-existent node ID): the accuser is considered faulty;

Back-up nodes cannot accuse their primary nodes. If that were allowed, back-up nodes could take primary nodes out of the cell at will. This concern overrides the opposite concern of the primary node being the faulty one and the back-up honestly trying to point that out. However, if a primary node is faulty, then its back-up node is not essential for this to be detected, since there are several other primary nodes that can detect the faulty behavior of that would-be accused primary node.

When a primary node accuses another primary node, then both have to be considered faulty. This solves the problem of determining if the accuser is being honest or malicious. If the accuser is being malicious, then the optimal solution would be to mark only the accuser as faulty. If the accuser is not being malicious, then at least the blamed node must be considered faulty. However, in some situations, even if the accuser is being honest, the optimal solution could include it as faulty too. For instance, when two honest cell peers lose the peer-to-peer socket connection to each other, it is initially unclear which one is to blame or, more precisely, which one should be removed from the cell and replaced.

[8] Originally, we designed the `FAULT` message to carry a list of faulty node IDs. However, due to the way we ultimately designed fault recovery, it became apparent that a list of IDs would not be of any use. In the event that a primary node detects several faulty primary nodes at once (unlikely), an implementation should instead send a series of `FAULT` messages, each one with a single ID in it. Even if two nodes A and B fail and an observer node O detects the failures at the same time, node O will have to elect either A or B to be the one that has 'failed first'. It doesn't matter which one.

Both would send FAULT messages accusing each other, and the cell manager would have to allow some wall-clock time for all simultaneous accusations to arrive and try to figure out what happened. It is easy to see that trying to be completely fair here would be too complex, would take too long in wall-clock time, and ultimately wouldn't matter because cell nodes are just daemon nodes: no human players are being denied play in a world cell.

One of the main advantages of an immediate attribution of 'faulty' status to cell nodes once a FAULT message arrives is that it shortens the overall wall-clock time for the fault recovery algorithms. This should become clear on the next subsection.

Fault recovery starts as soon as the first cell node is marked as 'faulty', and the fault detection process described here can continuously change the course of recovery. In other words, faulty node detection (this subsection) and cell fault recovery are concurrent. While fault recovery runs, the list of faulty cell nodes can grow, as new FAULT messages arrive at the cell manager. The only exception is when a FAULT message is received from a cell node that is already marked as faulty. In this case, the message is disregarded and no additions to the faulty nodes list are made. The next subsection explains how the cell fault recovery process reacts to this dynamic list of 'faulty' cell nodes to restore the cell to 'working' status.

### 4.3.3 Cell fault recovery

Fault recovery is a disrupting process for the players on an affected cell and possibly on its surrounding cells, so a recovery procedure should try to minimize that disruption by completing instantly or at least quickly, so that players don't notice it or aren't too bothered by it. However, quickness cannot justify recklessness. We must design in some deterrent so that a malicious group of cell nodes won't be able to easily game the fault recovery mechanism to allow illegal state cheating. Our first priority is to avoid those windows of opportunity for cheating.

As it turns out, there is no perfect solution for fault recovery that achieves maximum security and maximum quickness. Thus, we have developed two separate recovery strategies:

- A 'quick recovery' strategy, which completes quickly but also widens the window of opportunity for illegal state cheating on the cell being recovered. The quick strategy is based on replacing individual faulty cell nodes and avoiding any lengthy operations such as downloads and uploads of cell state snapshots between remote hosts;

- A 'full recovery' strategy, which takes a long time to complete, but does not widen the window of opportunity for illegal state cheating on the cell being recovered. The full strategy replaces all faulty cell nodes with a fresh set of volunteers from the volunteer pool, and it may involve lengthy operations such as downloads and uploads of cell state snapshots between remote hosts.

Our over-arching recovery strategy is the following: if the number of nodes marked as 'faulty' (as seen in the previous section) is below a certain threshold, then we employ the quick recovery strategy. If the number of faulty nodes exceeds the threshold, then we switch to the full recovery strategy. Since the number of faulty nodes can only grow during a single recovery process, then the quick strategy can be replaced by the full strategy during the same over-arching recovery process, but the reverse is not possible.

From now on, let $q$ be the number of cell nodes marked as 'faulty' by the cell manager during a single cell failure and recovery process, and $Q$ the maximum number of faulty cell nodes for quick recovery to be considered. So, if $q \leq Q$, then a 'quick recovery' algorithm can be running. If $q > Q$ is detected during the recovery process, then a 'full recovery' algorithm must be running.

Choosing a good value for $Q$ will depend on each game deployment, and it might have to be tweaked during a game's lifetime. However, we can try to define a good default value. For that, we do an analysis below of some candidates:

- $Q = 0$**:** This would be the safest option. However, in this case quick recovery never runs. Upon any sign of failure whatsoever, the cell manager would perform a full recovery. This presents maximum security and can actually be a good solution, when coupled with the 'grief system' which is described in a later subsection;

- $Q = 1$**:** This presents the smallest window for state-cheating while, at the same time, allowing some room for quick recovery to actually run. However, with this threshold, any time a primary node accuses a different primary node, quick recovery will not run. This brings the central consequence that a single malicious node can trigger a full recovery in any cell it is at, at will. Thus, this setting might not be optimal. However, by that same analysis, neither would $Q = 0$ be;

- $Q > 1$**:** Any value from 2 and above avoids the disadvantage listed for $Q = 1$: now at least two coordinated malicious primary nodes are needed on a cell to trigger a full recovery. However, as the value of $Q$ increases, the window for state cheating increases. This, on the other hand, can be countered by increasing the number of cell nodes on each cell. For instance, if a game is being constantly vandalized by malicious fault reporting, both $Q$ and the number of cell nodes in each cell can be increased simultaneously.

Therefore, we establish $Q = 2$ as a default value, since it is the safest value that also prevents lone malicious nodes from triggering full recoveries at will. If maximum security is desired, $Q = 0$ can be employed instead. The $Q = 1$ setting might be useful if it can be coupled with some additional trick, such as temporarily banning 'accuser' nodes from serving in cells for a short time (e.g., two hours).

It should be noted that whenever $q > Q$, the quick recovery algorithm is completely aborted, without any kind of additional 'transition' or 'hand-off' to full recovery. Full recovery is designed to start from whatever is remaining from the smoldering wrecks of the failed cell simulator. As will be seen later, even if all cell nodes have vanished, full recovery can complete successfully.

### 4.3.4 Quick recovery

The quick recovery algorithm is performed for a single cell node that has failed. If multiple cell nodes have failed then one instance of the algorithm should be started independently to replace each failed node. So, while using the 'quick recovery' strategy for the cell, a cell manager can actually be running several instances of quick recovery algorithms.

The quick recovery algorithm is also different, depending on the type of cell node being recovered, either primary or back-up. Replacing a back-up node should be completely transparent and shouldn't affect player nodes in any way. Replacing a primary node is a

bit more complicated, but still the algorithm is optimized for minimum gameplay disruption. Replacing the primary node causes the corresponding back-up node to be replaced also, so the quick recovery algorithm for back-up nodes can be seen as a subset of the algorithm for replacing the primary node.

An important limitation of the quick recovery algorithm is that it only works if the cell node to be replaced still has its pair alive. In other words, a back-up node can only be replaced if its corresponding primary node is still working in the cell. Conversely, to replace a primary node, the back-up must be alive and up-to-date. If this condition does not hold, for any cell node being replaced, then quick recovery must be abandoned completely and the cell manager should enter into 'full recovery strategy' mode, the same as if the $q > Q$ threshold were just crossed.

#### *4.3.4.1 Quick recovery for back-up nodes*

Let B be the failed back-up node and P its primary node. The recovery of a back-up node works as follows:

1. Cell manager selects a new node B' from the volunteer pool, and sends the message `REPLACE_BACKUP` to both P and B'. That message contains the network (socket) addresses of P and B' and the cell ID of P;

2. P, upon receiving `REPLACE_BACKUP`:

   (a) P immediately disconnects completely from B. Any further messages from B will be ignored;

   (b) P Also connects to B' (or accepts a connection from B'). When the connection to B' is successful, P sends `REPLACE_BACKUP_OK` to the cell manager, and starts to synchronize B'.

3. Upon receiving `REPLACE_BACKUP`, B' keeps trying to connect to P until it accepts the connection. When that is successful, B' sends `REPLACE_BACKUP_OK` to the cell manager;

4. If the cell manager times out while waiting for a pair of `REPLACE_BACKUP_OK` messages, it fails the replacement back-up node and re-starts this algorithm;

5. If the cell manager receives `REPLACE_BACKUP_OK` messages from P and B', the cell manager assumes the recovery is complete.

Step 2 ends with P synchronizing B'. Since B' is a blank volunteer node, it has no cell state whatsoever stored in it. P needs to first send a copy of its most current conservative state, followed by a continuous stream of all subsequent events. The bigger is the cell snapshot data, the longer the back-up node will take to download it. Since the download can take a long time to finish, before uploading it P needs to save a copy of the current conservative cell state into a buffer. Though P's cell state is constantly being updated, the download continues until B' finishes downloading the now 'old' snapshot. B' will then have to execute the potentially large backlog of events until it catches up. There are no constraints placed on how long this can take. If B' takes half an hour to catch up, then so be it. P will have to hold all of the relevant event data in process memory until it can be dispatched to the outgoing socket buffer. As always, if P or B' can't perform the task (e.g., not enough memory) it is simply a matter of sending a `FAULT` message to the cell manager, which will take over from there.

#### *4.3.4.2 Quick recovery for primary nodes*

Let P be the failed primary node, B its back-up node, and X any of the primary nodes that are not P. The recovery of a primary node works as follows:

1. Cell manager sends a `REPLACE_PRIMARY` message to all primary nodes (except P) and to B. The message contains the ID of P, which is the node being removed, the ID of B and the network (socket) address of B, and the network address of all primary nodes (except P);

2. B, upon receiving `REPLACE_PRIMARY`, knows that it will replace P, so B disconnects and ignores P. B then attempts to first connect to all primary nodes (or accept their connections);

3. Any X, upon receiving `REPLACE_PRIMARY`:

   (a) X knows that P will be replaced by its back-up node, and that back-up nodes are always running their simulators a little late relative to their primary nodes. This means that some old events that X had already timestamped and dispatched to the cell will have to be re-issued to B. The size of that 'back-off window' depends on many factors, including the amount of aggregation that is performed between primary and back-up nodes. So, to avoid losing any more stored events before X knows which old events B will need, X temporarily suspends its 'fossil collection' (FUJIMOTO, 2001), or elimination of events by age;

   (b) X now know that B will replace P, so X disconnects from P;

   (c) X tries to connect to B (or accept B's connection attempt);

   (d) X immediately disregards any turn flags from P, counting from and including the simulation time CST of X's conservative simulator, regardless if a turn flag from P was already received for the current turn CST or not;

   (e) X may have received an input from P timestamped for turn CST, the current conservative simulator turn at X. However, X cannot just disregard inputs from P for CT because other simulators may be at CT-1 still, and if they do the same, they will discard P's input for CT-1, which was already used by X (this node) to compute its CT-1, since it is now at CT. So, all remaining primary cell nodes have to run a voting round to discover when exactly, in simulation time, P's inputs can be taken off the lock-step simulation. To that end, X broadcasts a `VOTE_PRIMARY_REMOVAL_TIME` message which contains MIN_CT, which is the last local conservative simulation time which already incorporated an input from P at X. It is probably the case that the following will always hold at X: $MIN\_CT = CT - 1$.

4. When a node X receives all `VOTE_PRIMARY_REMOVAL_TIME` from all other primary nodes except nodes being replaced (such as P), it computes the minimum value among all MIN_CT values received, which will be called LAST_VALID_CT, or the last conservative simulation time where the input of P should be counted by all primary nodes. If X verifies that $CT > LAST\_VALID\_CT$, then it means that X has already incorporated an 'illegal' input from X on its conservative state, which X cannot recover from. So, X sends a `FAULT` message to the cell manager, accusing

itself. If X verifies instead that $CT \leq LAST\_VALID\_CT$, then it instead just discards any and all input events received from P so far which have a timestamp ET such that $ET > LAST\_VALID\_CT$, and the lock-step simulator at X will not lock waiting for any input from P after LAST_VALID_CT.

5. If a node X times out while waiting for one `VOTE_PRIMARY_REMOVAL_TIME` message from a remote primary node (excluding the ones being replaced), it issues a `FAULT` message to the cell manager accusing the timed out peer (the other primary node).

6. When a network connection is successfully established between any X and B:

   (a) Both X and B send a `REPLACE_PRIMARY_CONNECTION` message to the cell manager with both node IDs as parameter;

   (b) B sends a `REPLACE_PRIMARY_EVENT_REQUEST` message to X telling the current timestamp T of its latest conservative state copy CST, along with a Hash(CST) so that X will be able to check that B is really synchronized with the cell;

7. Any X, upon receiving a `REPLACE_PRIMARY_EVENT_REQUEST` message from B, will re-send all events it had already sent to the cell, starting from the T timestamp given in the message, up to the current conservative simulation time at X. If X doesn't have events corresponding to T, it sends a `FAULT` message to the cell manager sending its own node ID as parameter. X also checks if the state hash matches what it had already calculated for its local conservative state at simulation time T;

8. If any X times out while waiting for a `REPLACE_PRIMARY_EVENT_REQUEST` message from B, it will send a `FAULT` message to the cell manager with B's node ID as a parameter;

9. If the cell manager times out while waiting for a `NEW_PRIMARY_CONNECTION` message pair from any X and B, both that X and B are marked as failed. This necessarily causes a 'full recovery' to run, because B was the last member of the pair (P,B) of nodes. Now that both are failed, full recovery has to run and this algorithm is aborted [9].

10. When the cell manager receives all `NEW_PRIMARY_CONNECTION` message pairs, it knows that the cell's overlay network is restored, with B in place of P. It also knows that all X nodes and B are already working to synchronize B, and that if there are any further problems, such as B not being able to catch up due to unrecoverable, missing events at one of the primary nodes, that they will complain with `FAULT` messages. At this point, the cell manager continues running the 'quick recovery for back-up nodes' algorithm described in the previous subsection, which will eventually grant a new back-up node to B, which is now the primary node.

[9] Certainly, the quick recovery algorithms could be designed to not require each pair to have at least one member alive. Or, a back-up node could be allocated and synchronized to B before promoting B to primary node, thus allowing B to fail here without falling back to full recovery. However, those introduce additional problems on their own right and also add an amount of complexity that, in our opinion, doesn't pay off.

Several instances of this algorithm can be running concurrently. For instance, if two primary nodes fail at the same time, both will be replaced at the same time by two instances of this algorithm. However, there is no complete isolation. For instance, the set of nodes named 'X' in the above algorithm is common for all concurrent instances. If there are two nodes P1 and P2 being replaced, then set of 'valid' primary nodes (included in 'X') in both algorithms will exclude both P1 and P2, and not only the individual 'P' being replaced by each algorithm separately. If this care is not taken, then the exchange of `VOTE_PRIMARY_REMOVAL_TIME` messages won't work, as all 'failed' primary nodes won't be answering for that messages, regardless that a single exchange of `VOTE_PRIMARY_REMOVAL_TIME` is specific to a single primary node being taken out. Besides this caveat, we have not yet verified if there should be any other interaction between concurrent instances of the algorithm.

As with quick recovery for back-up nodes, the primary node recovery algorithm ends with an optimistic assumption that the replacement node will successfully 'catch up'. And, the primary replacement actually ends with a call to the back-up replacement algorithm, so there will actually be two optimistic 'catch-up' processes running concurrently. Both algorithms are designed to cause minimal disruption to game-play, assuming that they complete successfully.

The central idea here is that, like back-up nodes, a single missing primary node can take all the time it needs to recover, without needing to halt the whole cell for this. For example, if a cell has $N = 8$ primary nodes, and one fails, the remaining seven primary nodes can share the load in serving any 'orphaned' player nodes that were being served by the missing primary node, and they can also temporarily supply neighbor cells with updates, probably at a slightly reduced rate.

During quick recovery, although well-behaved nodes should complete the algorithm successfully, many different things can go wrong for varied reasons. In the case of failures, the cell manager is notified with additional `FAULT` messages from the cell peers. If the $Q$ threshold is crossed during additional fault reporting, then the cell manager finally decides that its optimism is no longer warranted.

The $Q$ threshold allows the cell manager to be highly optimistic and trust the peers on resolving most of the situation in a peer-to-peer fashion, with minimal coordination performed by the cell manager. That is, while $Q$ is not crossed. The threshold allowed us to design two very simple, and diametrically opposed algorithms, and bring them together by using a simple fault detection scheme and a simple criteria for choosing which should be running. In this section we have presented the quick algorithms, and the next section will present the more disruptive, full recovery algorithms.

### 4.3.5 Full recovery

Full recovery is started whenever quick recovery no longer applies during cell fault recovery. At the end of full recovery, all cell nodes will have been replaced by new nodes drawn from the volunteer pool. None of the original cell nodes will remain on the cell.

However, before replacing the cell nodes, the cell manager attempts to download an up-to-date snapshot of the cell state from the cell nodes. Since the cell manager is, most of the time, out of the loop in regards to the current state of the cell, the cell manager is not in the best position to seed the new nodes with the current cell state. Instead, it must first query the old cell nodes that have 'survived' so far to try and extract a copy of the cell state from them. While doing this, the cell manager must make sure that a large enough number among the old cell nodes agrees on a common cell state, to minimize the chance

of illegal state cheating being successfully performed.

We will call the minimum quorum of agreement over a single state $M$. $M$ is the smallest number of cell node pairs that must be 'in agreement' over a common cell state (let's cal it 'S'). Assuming a node pair is formed by two nodes P and B, for that node pair to 'agree' on S, one the following must hold:

- P is alive and agrees with S, and B is dead;
- B is alive and agrees with S, and P is dead;
- P and B are alive and both agree with S;
- P and B are alive, and P agrees with S (B is disregarded);

Agreement means that a cell node knows for a fact that a given state S is valid for its cell. A node tests for validity of S when it receives a hash H from the cell manager, which corresponds to S. The cell node then looks up H on its list of past cell state hashes and verifies if there is a match. If there is a match, then it means that the cell node recognizes S as a valid cell state. It doesn't necessarily mean that the cell node still has the actual cell state data for S, since each node only keeps one 'live' copy of the conservatively-synchronized state, which is constantly updated.

If the cell manager gathers a distributed agreement on a given cell state S, such that an amount of agreeing node pairs A satisfies $A \geq M$, then the cell manager verifies if at least one cell node (any one) has the actual cell state data for S. If that is true, the cell manager attempts to download the data for S from the cell node(s), and then it uses that to initialize the full new set of cell nodes.

If no agreement can be reached by at least $M$ cell node pairs, or if the only agreements are on state hashes and no cell node actually has kept the state data for that hash, then the cell manager completely abandons the old cell nodes and recreates the cell state itself from whatever up-to-date cell data it has stored locally (this is the 'vital cell state' data at the cell manager). This is a last resort measure which is preferred to having a too small agreements decide the state of the cell. Similarly to the $Q$ threshold, the $M$ threshold should probably be tuned so that a good balance between security and full discards of client-side cell state can be achieved.

To clarify the explanation, we will split the full recovery algorithm in two: 'full recovery from client-side state' and 'full recovery from server-side state'. Client-side recovery is always attempted first but, while it runs, the cell manager can decide that it can no longer recover from client state. In this case, the algorithm jumps to the steps described in server-side recovery.

#### *4.3.5.1 Full recovery from client-side state*

The full recovery from client-side state is attempted first. Let X be any cell node. Full recovery works as follows:

1. The cell manager verifies if the potentially reachable set of cell nodes is sufficient to meet the $M$ threshold of agreeing cell node pairs. As examples, if $M = 0$, or if $M > 0$ but all primary cell nodes are unreachable (disconnected from the cell manager), then the full recovery from client state is aborted and the server-side recovery starts instead;

2. The cell manager sends a `FREEZE_STATE_QUERY` message to all cell nodes which are still connected (reachable);

3. Any cell node X, upon receiving `FREEZE_STATE_QUERY`:

   (a) X stops (freezes) all simulation. Incoming and outgoing updates, input, turn flags, etc. are all discarded;

   (b) X sends a `FREEZE_STATE_RESPONSE` message back to the cell manager, which contains a list of tuples in form (CT, Hash(CST), HasData), where CT is a past (or the current) simulation time of the conservative simulator at X, CST is the actual conservatively-synchronized state data computed by X at simulation time CT (which may have already be discarded by X), Hash(CST) is a hash computed over CST (which is kept for a longer time at all nodes), and HasData is a boolean flag which tells the cell manager if X still has the CST data locally ($HasData = TRUE$) or not ($HasData = FALSE$).

4. If a cell node X does not answer to a `FREEZE_STATE_QUERY` message in time, or if the cell manager detects that X is actually unreachable by other means, it removes X from the set of reachable nodes and re-checks the $M$ threshold, potentially aborting full client-state recovery and jumping directly to full server-state recovery;

5. When the cell manager is finished waiting for all `FREEZE_STATE_RESPONSE` messages (either got all responses early or timed out waiting for some of them), it attempts to obtain a state copy S from the clients:

   (a) If there is at least one state S, whose simulation time is T, such that a tuple (T, Hash(S), TRUE) exists among the tuples received, and that, among all tuples, there is an agreement on S (by verifying the submitted hashes) such that the amount of agreeing node pairs A satisfies $A \geq M$, then the cell manager elects S as the 'official' state of the cell. If there are multiple states S that satisfy those constraints, then the one with the greater timestamp T is selected. If no S satisfies the constraints, the cell manager aborts the algorithm and jumps directly to 'full recovery from server-side state';

   (b) The cell manager attempts to download S from one of the cell nodes X that reportedly had it by reporting (T, Hash(S), TRUE). If the download fails, S is requested from another cell node that has it (also, $M$ is re-checked since that X node just became unreachable). If no cell nodes can successfully upload S to the cell manager, that S is discarded and the next best S is selected from the reported tuples. If no S remains, then the cell manager aborts the algorithm and jumps directly to 'full recovery from server-side state';

6. Now that the cell manager has valid cell state data S obtained from a minimum quorum of agreeing cell node pairs in hands, it disbands all cell nodes back to the volunteer pool and draws a full set of new volunteers to the cell;

7. The cell manager sends S directly to each new cell node, including the back-up ones;

8. Once the upload of S is complete, the cell manager sends a message which authorizes the new cell nodes to start the simulation.

If this algorithm completes, it means that the state of the cell was not lost. However, there is a potential problem related to simulation time 'going back'. As is, the algorithm described above may choose a state S that could be even a few seconds behind the most recent state R that the cell had collectively computed. If between S and R there were events such as object transfers between cells, those may have been unintentionally undone. The game (application) code at the cell manager should be able to use the results of the consensus gathered out of the tuples to decide whether it should undo something or not. To do that, the cell manager may have to interact with the managers of neighbor cells and maybe negotiate some specific object removals or re-insertions. This restoration of global game consistency has to be performed by application (game) code. In a following section, we explain how the FreeMMG 2 primitives can be used to implement object ownership transfer between cells in a way which prevents objects from being lost or duplicated in the process.

#### *4.3.5.2 Full recovery from server-side state*

This is the last resort algorithm for recovery. At this point, the cell manager has completely given up on whatever cell state data might be left at the cell nodes that are still alive. The cell manager will instead rebuild the cell state from scratch. This should be done using whatever server-side information the cell manager had about the cell, or maybe use fixed rules such as to use a default 'blank' cell state.

In FreeMMG 2 we will make a 'vital cell state' feature available. It allows the cell nodes to periodically send updates about the cell state to the manager. The frequency of vital cell data updates and their contents is completely left for the application. Likewise, the function that maps vital cell state back to an actual cell state is completely left for the application. The middleware will just call back application code at the cell manager to return a valid cell snapshot.

Besides this vital cell state aspect, the full recovery from server-side state is actually trivial:

1. The cell manager decides on the new state S using whatever server-side data it has available;
2. The cell manager draws a full set of cell nodes from the volunteer pool;
3. The cell manager sends S directly to each new cell node, including the back-up ones;
4. Once the upload of S is complete, the cell manager sends a message which authorizes the new cell nodes to start the simulation.

Like with full client-side state recovery, some object transfers may have to be undone or re-done, or other consistency checks may have to be run between this recovered cell and its neighbors. This is left for the application to do. In a following section, we show how the application can implement global consistency checks and restoration procedures using the primitives that are described in this chapter such as vital cell state storage and inter-cell reliable messaging.

### 4.3.6 Grief system: filtering problematic cell nodes

The *grief system* of FreeMMG 2 is named after the *grief points system* employed in many persistent-state games such as PlanetSide (SONY ONLINE ENTERTAINMENT,

2008). To maintain realism and to encourage well-thought team-play, most real-time combat games allow team-mates to (accidentally) damage each other, an event which is often called 'friendly-fire'. However, there are griefers, which are players that intentionally attack team-mates[10]. Since it is impossible to determine, in the heat of battle, whether friendly-fire was intentional or not, PlanetSide and other similar games simply assigns negative points, or *grief points*, to a player which damages a team-mate. If the amount of grief points assigned to a player reaches a certain high value the player is first warned and, if another higher threshold is reached, the player is temporarily or permanently banned from the game. A grief points system works because players will fall in one of the four categories below, which are all adequately treated:

- Infrequent unintentional attackers: those will almost never trigger the grief threshold and will be able to keep playing normally;
- Infrequent intentional attackers: those seasoned griefers will be able to do some damage, but will have to stop after a very short time or risk triggering the thresholds;
- Frequent unintentional attackers: those honest but extremely clumsy players will be warned first, which will make them know that their skill is probably incompatible with the in-game situations they are putting themselves into;
- Frequent intentional attackers: those will be permanently banned eventually, or at least be on a temporary ban most of the time.

In FreeMMG 2, we employ the exact same concept to detect volunteer nodes that cause cells to fail. Since the cell manager can't really tell which peers have caused the cell to fail, he simply distributes 'grief points' equally to all cell nodes of a failed cell. Once a volunteer node accrues too many grief points, it can be temporarily or permanently barred from serving as a cell node. As in the original player-oriented grief systems, the FreeMMG 2's node grief points 'heal' over time, so a node that fails infrequently and unintentionally over a long period of time will never reach the threshold for banning. The idea is that, as with the original player-oriented 'grief' systems, this will create a tendency to, over time, prevent malicious nodes from triggering cell failures on purpose and go unpunished (or worse, punish the honest nodes instead). The damage is limited, for each rogue node, to what fits below the grief threshold.

However, the grief problem doesn't map automatically from players to cell nodes. For one, players in MMOGs such as PlanetSide are always authenticated and they pay actual real-world money for their accounts. This means that a dedicated attacker cannot create an arbitrarily large amount of fake accounts for griefing without spending real money, which is an effective deterrent. In FreeMMG 2 however, we are envisioning a scenario where we would be drawing unauthenticated volunteers to the network. In this case, we can only track volunteers by their previous track record in serving the network or by their IP addresses, which can be dynamic.

To use volunteer track records, they would have to authenticate every time they connected themselves to the volunteer pool. Even if authentication was required, this would not be a deterrent since volunteer accounts can't have a real-world, concrete cost attached to them, besides something light such as requiring users to fill out a form and provide a

[10]More generally, a *griefer* is a player in a multiplayer game which is playing solely to harass other players (WARNER; RAITER, 2005), causing much *grief* to them and hence the name.

working e-mail address. We certainly don't want volunteers paying real money the game provider for the 'privilege' of serving as volunteers, since that would defeat the whole idea.

To make the grief system work in this scenario, the best solution is to require authentication of volunteers through freely available accounts, and couple that with a mechanism that prefers volunteers with good track record to serve in cells. This is analogous to businesses that don't trust bank checks issued from persons that have just opened an account at the bank in question. Assuming that there is a large amount of honest volunteers, those will tend to 'rise' above the mass of fake, single-purpose accounts created all the time by persistent griefers. Once in a while, blank accounts could be chosen to serve in cells, allowing them to develop a history of good service. For instance, deserted cells (cells that are not serving player nodes currently) could be used to test blank volunteer accounts.

We have not defined specific thresholds for grief points, and we leave that for the middleware implementation or the game implementor. However, this 'grief points' tracking is an essential part of a FreeMMG 2 implementation, and will be its last line of defense against nodes that cause failures on purpose.

### 4.3.7 Fault tolerance: comparing with FreeMMG

In the predecessor FreeMMG (CECIN et al., 2004) model and middleware implementation, there was a concern about communicating different faults to the server-side, and trying to guess whether cell nodes were telling the truth when they accused a peer from failing them or not. The resulting fault detection and recovery procedure was overly complex to program. Every time an exception handler was being written, there was a question of what message it should send to the cell manager, and what parameters should it have, and how that would be handled by the cell manager exactly. Often, the different parameters and messages ended up being handled the same at the cell manager because it could not decide, for instance, if a peer reported as faulty really failed or the reporting peer (or group of peers) was lying.

So, instead of preemptively classifying every possible failure situation and trying to handle each one in an optimal, customized way, the middleware and game implementors can now discover or define particular problematic situations and slap the 'fault' label on them easily, as needed. To do that, it is enough to send the cell manager a `FAULT` message on the exception handling point, and no further analysis is needed. In the end, everything is solved naturally by replacing the cell nodes until a group that works well is obtained.

We believe that the FreeMMG 2 `FAULT` message coupled with the new 'Grief system' is a superior approach to the one employed in the original FreeMMG design and implementation, which involved several different failure reporting messages and specific handling mechanisms. Overall, FreeMMG 2 relies much more on strong tendencies than guarantees for fault tolerance. A strong tendency of filtering out bad nodes by blaming all cell nodes equally replaces the need to accurately point out the actual offending nodes in any particular failure situation.

The only downside of this new approach is the reliance on the assumption that all actual, running FreeMMG 2 networks will be well provided of 'good' cell nodes, that is, large groups of stable nodes which have good peer-to-peer connections to each other. This can only be addressed tangentially. For instance, deploying a game network which only accepts peers with IP addresses that can be mapped to same continent or country. Or, dividing the pool of volunteers into groups that share a common network proximity feature, and draw all volunteers of each cell from a single group. However, for this thesis

we decide to leave this assumption as it is. Working the 'volunteer pool' concept further is left as future work.

### 4.3.8 Remaining issues

The fault recovery algorithms presented earlier do not deal with all issues. There are a lot of small pending problems, but those should not be hard problems. For instance, when full recovery finishes, the network (socket) addresses of the new primary nodes should be informed to all primary nodes of the neighbor cells, since that information is vital so that adjacent cells can send object updates to each other. Similarly, player nodes should be informed of the new cell primary nodes, so that they can resume playing after the several minutes or so of cell outage are over. We designed FreeMMG 2 so that the ties between neighbor cells and the ties between cells and their player nodes are completely flexible. So, there are no hard timing or event ordering constraints associated with those tasks. This re-tying can be implemented without worries of interacting badly with the fault recovery algorithms and vice-versa.

## 4.4 Player-cell synchronization

Thus far, we have defined how the cells work. But, for the game to be playable, the player machines must be able to send *something* to the cells, such as requests to change the shared state or authoritative updates to the shared state, and also receive *something* back from the cells, such as state updates. In this section we provide some alternative ways to make player nodes and primary cell nodes interact.

For player-cell synchronization we have identified several possibilities, and there may be other solutions available still. In the next section we describe three alternative mechanisms suitable for connecting player nodes and the cell overlay. The first one, the 'default protocol', which is primarily a bandwidth-efficient protocol, is more formally defined. The others, which optimize for reduced interaction latency, are ideas whose further development is left as future work.

### 4.4.1 Why focus on bandwidth efficiency

In this section we discuss our view on what are the important properties for the player-cell synchronization mechanism. This explains why we have chosen to focus on bandwidth efficiency instead of latency reduction for the default protocol.

The cell simulator mechanism that was described in previous sections focuses on security. The cell is a way to keep a piece of shared state safely synchronized in real time by a group of 'client' nodes which are each unreliable if considered individually. That focus on security however comes at the price of additional latency and scalability. The additional latency is mainly introduced by the additional hop that events must travel between one primary cell node that is introducing the event to the cell and its peers. We avoid multi-hop event dissemination inside the cell to avoid increasing latency beyond the minimum necessary (it's still a penalty, though). And, which is more important to the point we will make below, by forbidding multi-hop dissemination of events inside the cell we limit the scalability of the cell since a full mesh protocol is $O(N^2)$ messages. Thus, the number of primary cell nodes allowed in a cell will not increase linearly as client bandwidth increases. It is reasonable to expect that the number of primary cell nodes in a cell should linger around ten or so, regardless of any game or middleware parameters such as the expected average and peak avatar populations of cells.

The relatively few primary cell nodes of the cells are the nodes which have to serve the game to the player nodes. Thus, the amount of bandwidth available for serving the game to players is limited. As the amount of player nodes interested in the state of a given cell grows, the only way the cell could reliably increase the available bandwidth is to replace primary cell nodes with 'super peers' which have greater available bandwidth. We do not actually explore that option in the present work; instead, we assume that the available bandwidth for serving players in each cell is limited. To deal with *hot spots*, that is, cells which have to serve a very large number of interested player nodes, we propose that the primary cell nodes just split whatever bandwidth is available among all player nodes. Thus, as the player population in a cell increases, the game quality can decrease to and below acceptable quality.

With these design decisions taken, it results that making a bandwidth-efficient player-cell synchronization mechanism is now imperative. Cutting the cell bandwidth required to serve a player in half means doubling the amount of players supported in a cell, which also minimizes the hot spot problem. As broadband technology, broadband market penetration and the Internet evolve, the hot spot problem will become less and less severe and the supported player population in a cell will increase. To have limited bandwidth is actually a good thing since it encourages rational game protocol design, which will carry over as the Internet improves. After all, UDP-based game traffic has to play fair with all other traffic.

For now, let's consider, for instance, that most cell nodes will be supplied by home users with ADSL connections that feature upload bandwidth caps of 256 Kbps. That's 32 kB/s. We can expect a significant portion of that to be drained by the $O(N^2)$ intra-cell mechanism and the inter-cell mechanism which generates $O(N)$ packets but which are larger in size than intra-cell packets. If the cell node machine happens to be doing any other networking tasks we can expect even less to be available to player serving.

Considering that FreeMMG 2 is more likely to be applied to MMORPGs than to MMOFPSs, at least at first, we have thus decided that bandwidth efficiency should be a priority over latency reduction for the default player-cell protocol. However, it should be noted that since cells function independently of how player nodes connect to them, there is a greater degree of freedom in designing the player-cell interaction support.

One of the main avenues for future work in FreeMMG 2 is experimenting with different player-cell protocols. For now, we just want to define one working solution which contributes towards showing that our core idea of secure cells is feasible when put together with other modules such as player-cell interaction. We actually expect that a FreeMMG 2 implementation will be comprised of a collection of primitives that allow an application to define any kind of player-cell protocol for itself, as well as provide a few ready-made mechanisms that the application can base itself on such as the 'default protocol' presented below.

### 4.4.2 Default protocol

The default player-cell synchronization (interaction) protocol optimizes first for bandwidth efficiency, second for security and third for latency. Its only contribution to maintaining interaction latency acceptable is in connecting player nodes and primary cell nodes directly, that is, by tying them by only one logical hop in the FreeMMG 2 overlay network, as shown in Figure 4.6 below.

Each player will be interested, at any one time, in only one cell. However, that is not a hard requirement. A player node may, at some times, interact with two cells simulta-

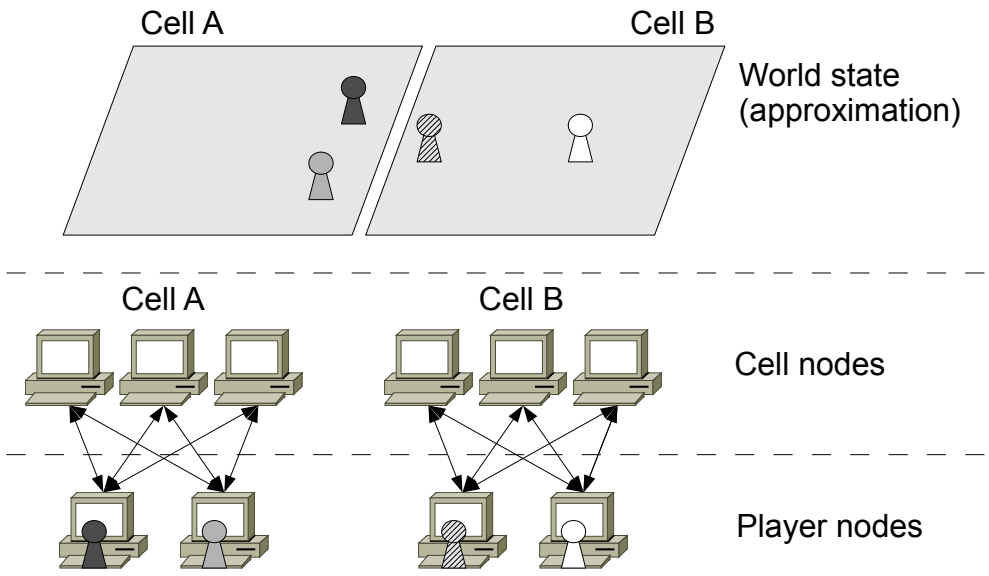


Figure 4.6: Default player-cell protocol. A player node connects to all primary cell nodes of the cell which currently owns its avatar.

neously. That can happen for a short time if the game object that represents the player avatar in the game is being transferred from one cell to another. We will address this issue in a later section. For now, let's assume that, at any one time, a player node will be synchronizing with only a single cell at any given time, which is the cell that currently *owns* (has authority to change the state of) its avatar object. In the default protocol, each player node has to connect to all primary nodes of its cell. By *connect* we mean only that these nodes can *potentially* exchange packets at arbitrary times.

The basic idea is that, whenever a player has to send a command packet, it chooses a random primary cell node to receive it. Upon receipt, the primary cell node submits that command to the rest of the cell as specified by BSS. In the opposite direction, the primary cell nodes pool together their available upload bandwidth to serve any number of players that are currently connected to the cell in the best manner possible. There is a coordination mechanism that guarantees that each player node will receive (most of the time) one update packet from the cell at every BSS cell tick. A player node receives an update packet from a different primary cell node at every BSS tick, according to a deterministic sequence that is indirectly negotiated among the primary cell nodes without the need for additional network messaging. The result is a bandwidth-efficient protocol which, from the point of view of the player node, is similar to a client-server MOG protocol, with the exception of when hot-spots occur. In the following subsections we explain these mechanisms in more detail.

#### *4.4.2.1 Player-to-cell command packets*

Each player node will issue *command* packets which, as far as the FreeMMG 2 middleware is concerned, are abstract requests to change the game state in some way. We assume that command packet issuance is at a fixed rate such as 10Hz or 20Hz at each player. That covers FPS games which is the worst-case scenario. However, in contrast to most client-server protocols, the player node will send each command packet to a different primary cell node. The choice is fully random each time, which results on average in an

uniform distribution of command packets among all cell nodes. Since command packets are small and downstream bandwidth is more abundant in current home user broadband technology, there is, in principle, no need for player nodes to coordinate with other player nodes in choosing which primary nodes will receive commands at any given time. The odds that a primary cell node will be swamped by incoming command packets is very low. Thus, player nodes are free to choose any target for their command packets among all primary cell nodes, without the need for an explicit coordination mechanism.

The goal of this random distribution of command packets among primary cell nodes is to 'increase security', which is the secondary concern of the default protocol. More formally, this contributes towards mitigating the effects of protocol-level cheating that can be performed by individual primary cell nodes. If a player node only sent commands to a single cell node, that cell node could delay or drop all of the player's command packets. By choosing a different primary cell node to receive it every time, we get a transparent mechanism that mitigates the effects of intentionally delayed or dropped command packets, as well as a basic way to uniformly distribute the load of command processing among primary cell nodes. And it is still a single UDP command packet being uploaded at every client network tick, which is as bandwidth-efficient as in a client-server FPS game protocol.

Another security advantage of the random command distribution is that it helps with the problem of primary cell nodes *forging* updates from players. In FreeMMG 2, we do not assume that players cryptographically sign their command packets. For now, we leave that for the application to define. Whenever it is possible for primary cell nodes to forge player commands, the scheme we described helps to detect that forgery is occurring as follows. If a player sends each command packet with an unique sequence number (serial number), and if the player sends each packet to a random primary cell node every time, then a malicious primary cell node has no way to know if it will be chosen to relay any particular command packet in a player's sequence. Thus, if a primary cell node forges a command F with sequence number S for player P, and P sends a command C with sequence number S through other relay, then cell nodes will eventually receive two commands, C and F, with the same S and P parameters. That is enough to detect that either some primary cell node is forging updates or the player is doing so. In other words, that 'group' (the cell nodes and the player in question) have collectively failed. That can be the starting point of a detection mechanism based on the *grief system* idea described earlier that, in the long run, can identify malicious cell nodes as well as players that are too frequently involved in groups that fail. However, since the cell is more important than a single player, the thresholds should first assume that the player is the one being malicious. Then, if primary cell nodes too frequently show up with conflicting player commands for several players, then they should be replaced and punished.

#### *4.4.2.2 Cell-to-player update packets*

In the reverse direction, the primary cell nodes have to use whatever upload bandwidth they have left to collectively sustain a stream of large game state update packets to each player node to which the cell owns the corresponding avatar object. Since cell nodes are heterogeneous, each one will have a different amount of upload bandwidth available for serving players. We assume that this bandwidth amount can be computed and updated in real time by each cell node individually. That estimate should be conservative enough so that the node will never overflow its UDP socket buffers with outgoing data. The mechanism described below operates inside the upload bandwidth share that is reported

as available to it. Obviously, reporting a wrong value that exceeds the actual bandwidth available to the node will cause it to malfunction. Also, the bandwidth consumed by a single stream of updates (for one player) is a global constant defined by the application. That should be conservatively estimated as well by the application programmer.

When a primary node first joins a cell, all other cell nodes assume that it has no upload bandwidth available for serving player nodes. The amount of upload bandwidth that each primary node has available for serving player nodes is actually part of the cell state, and is conservatively synchronized using the same mechanisms used to synchronize the execution of any other kind of input that changes the cell state. The list of fully active primary cell nodes is also part of the conservatively-synchronized state, though intermediary states (such as when a node is downloading a state snapshot) have to be modeled outside of the conservative protocol, since they are steps into establishing the node as a member of the conservative protocol itself. Finally, the official list of player nodes that are being served by the cell is also conservatively synchronized among all cell nodes. The latter can be simply inferred from the list of avatar objects currently represented in the conservative state. That may result in players receiving updates from two different cells for a short period during avatar object transfers between cells, but that is of minor concern – the player can simple choose one cell as its current source of updates at any given time. We will discuss object transfers in a later section.

Among the first events submitted to the cell by a primary cell node is an event that conservatively changes its reported upload bandwidth that is available for serving players. We suggest that this be communicated as a 16-bit unsigned integer number which represents an amount in kilobytes per second, rounded down from the actual value. Needless to say, the node can send a new event whenever it detects that its available upload bandwidth has changed. For one, this allows for an implementation of an adaptive protocol which 'discovers' safe values for cell node upload bandwidth by trial and error.

Now, the primary nodes have a way to know each other's upload bandwidth available for serving update packets to players. They also have a synchronized view of the player nodes that need to be served. With that information in hand, each node locally, and without any additional coordination, can determine which player nodes it should update whenever an unconditional, periodic network 'tick' occurs at the cell[11]. The function should ensure that, most of the time, the primary cell nodes will distribute updates uniformly among all players. As an example, consider that the collective upload bandwidth available at a cell is of 512 Kbps (for example, each of eight nodes reports 64 Kbps available for serving players), and that the cost of a stream of updates is globally and statically defined as 32 Kbps (4 kB/s). This allows the cell to send 16 update packets to players at every BSS tick. Thus, up to 16 players can be served by the cell with maximum quality. If there were 32 players instead, then half of these should be updated at even ticks and the other half at the odd ticks, thus lowering game quality by half. By knowing the ID of all primary cell nodes and all player nodes, as well as all bandwidth parameters involved and having a synchronized sequence of numbered ticks, coming up with a deterministic function that distributes the load fairly should not be hard, thus we do not specify it any further in this thesis.

Another key characteristic of the update distribution function is that it should not

[11]One should remember that in BSS, each cell node 'ticks' at a fixed rate. The tick event triggers the sending of BSS synchronization packets from each primary cell node to all other cell nodes, as well as inter-cell packets and primary-to-backup packets. Now, in addition, it will also trigger the cell-to-player update packet sending function.

bind players to the primary cell nodes. Even if the set of players and cell nodes and the reported bandwidth remains constant, at each tick a player should be served by a different cell node. The distribution should be uniform in such a way that a player node will get a given update from any of the primary cell nodes with approximately equal probability. The distribution can not be fully random however, since it is derived from a known, deterministic function. As with player commands, by having players receive updates from all cell nodes we mitigate the effects of protocol-level cheats. If a single primary cell node drops, delays or forges updates to a player, it will happen only for a fraction of the incoming updates.

That is as far as we can get in protocol-level cheat mitigation without increasing the bandwidth required to update a player beyond a single stream of update packets. Without insight into game rules, having primary cell nodes sign their outgoing updates or having several cell nodes send several updates simultaneously to the same players doesn't help us because the source of the updates is the *optimistic* state of each primary cell node. Since it is legal for optimistic states to diverge for any given simulation turn value, the middleware has no way to detect forged updates using a transparent mechanism such as doing bitwise comparison of updates issued by different cell nodes. As we have stated before, dealing with most protocol-level cheating is left for the application or for future work. Our goal here was simply to mitigate its effects by leveraging the fact that we already have several sources of cell state due to our replication approach.

#### *4.4.2.3 Additional remarks*

As an additional measure for dealing with hot-spots, player nodes can detect when a cell is overloaded by checking the frequency in which they are receiving updates. In these cases, players can pro-actively scale back their command sending frequency using a hard-coded function to be supplied by the application. This should be just a last-resort measure to avoid overflowing primary cell nodes with incoming command packets when very bad hot-spots occur in a cell. At this point the game should be already unplayable, so a trivial function such as reducing command rates in direct proportion to the drop in update rates should be sufficient.

Finally, it should be noted that there is no harm whatsoever when player commands or cell updates go to 'wrong' recipients. We model that as being equivalent to packet loss, which should be tolerated by the application anyway. We assume that player commands model implicit *requests* such as reporting the current state of player input devices, and thus the cell doesn't need to keep a 'complete history' of all commands of all players. Likewise, updates incoming into player nodes supersede each other, and the application is responsible for making the best use out of whatever updates it can get using the techniques discussed in Section 2.4.

Thus, the application should guarantee correctness and never assume anything about these packets arriving or not. Assuming that the mechanics defined above tend to converge after the parameters settle and the conservative state evolves, it becomes largely unnecessary to model what happens, for instance, during a change of reported upload bandwidth by a primary cell node, or during players joins and leaves. We consider that the above description of the default protocol covers the player-cell synchronization feature in a satisfactory way for now, considering the security and decentralization focus of FreeMMG 2.

### 4.4.3 Fixed cell node protocol (draft)

An alternative for decreasing interaction latency and for increasing consistency lies in optimizing the mapping between player nodes and primary cell nodes. Instead of the current security-oriented random mapping, a player node could instead measure the network distance between itself and all primary cell nodes and choose the closest one for sending commands and receiving updates. Or, alternatively, players in the same cell sharing an AoI could be assigned to a common primary cell node, reducing latency and thus increasing visual consistency for interactions between them. These two techniques could be combined in the same mapping algorithm for optimal results. Figure 4.7 illustrates this proposal. In the figure, the single links between a player node and the cell represent the relative stability of the choice of primary cell node by each player node. Of course, these can change, which would possible trigger a socket 'connection' procedure and change the overlay in a 'lazy' fashion.

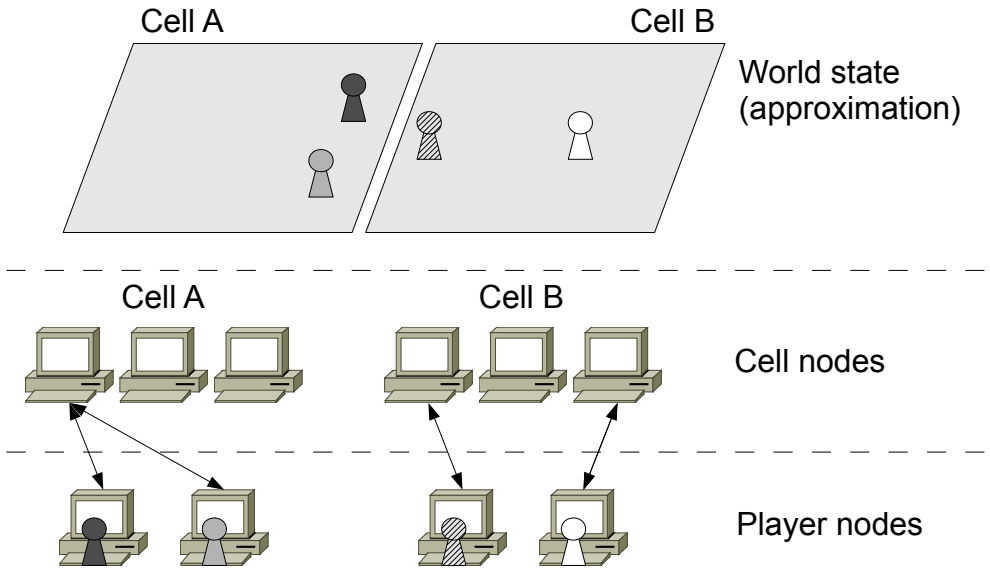


Figure 4.7: Fixed cell node protocol. A player node connects to the cell which currently owns its avatar, and to the primary node to which it has the best network connection.

This change, however, affects the security profile of the model. Forcing player nodes to depend on a single primary cell node aggravates the issue of a single primary cell node misbehaving by, for example, sending fake updates to its players or dropping their commands. These can be mitigated if players communicate both with the best primary node and also with another node at each turn as discussed in the previous section. But there is also the issue of the limited and varied amount of upload bandwidth available at each primary cell node. A player node may find out that its preferred primary cell node cannot handle any more clients, and thus need to fall back to the next best primary node in a list, and so on. It may be necessary to weight in the RTT reported by each player to every primary cell node in order to achieve an optimal allocation of player nodes to primary cell nodes, for whatever definition of 'optimal'. We have not developed this idea further and we leave it for future work.

### 4.4.4 Player mesh protocol (draft)

Players could exchange packets directly between each other. That could be leveraged somehow to decrease the interaction latency and increase consistency among player views. In practice, what we mean is that this could be leveraged to allow players to inform their current positions to each other directly, since avatar position is the only factor that is guaranteed to generate a constant stream of temporal inconsistencies across the whole game session and across any MOG or MMOG imaginable[12].

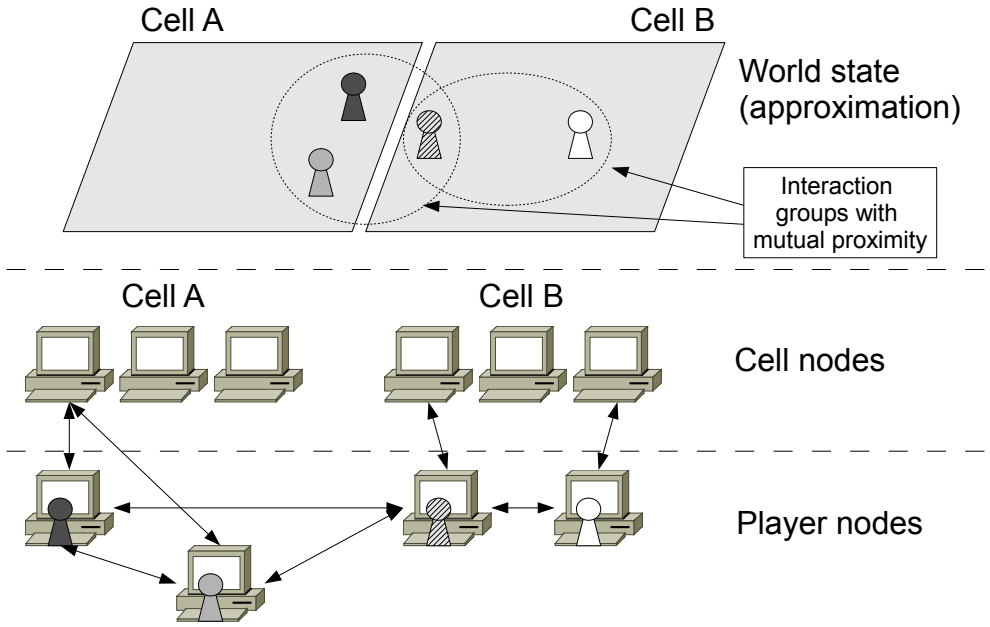


Figure 4.8: Player mesh protocol. A player node connects to the cell which currently owns its avatar, and to the primary node to which it has the best network connection and, additionally, to all player nodes in range.

One path to such a solution would be one where each player dictates its own position to its cell and also to other players in its AoI. Figure 4.8 provides an example of what the resulting logical overlay would look like, compared to the previous two protocols. Notice that the player also connects to the 'nearest' cell node as in the previous section's protocol. To avoid *speed cheating* (or *speedhack*), the *hack-proof synchronization protocol* (FUNG; LUI, 2009) could be employed between players and between player and its cell, or maybe a detection approach could be employed. However, the current FreeMMG 2 model does not descend into such details. By having players send out multiple packets, FreeMMG 2 would be probably assuming more about the game protocol, since it would be difficult to have players exchange *movement requests* (such as input device state) between each other in this kind of setting. Player mesh (or *mutual notification*) protocols generally imply that the players are authoritative over their avatar state. That is actually optimal from the point of view of overall reduction of interaction latency, but this introduces some cheating opportunities.

Besides the issue of cheating, there is also the issue that player nodes will also use more bandwidth. That has the potential side effect of reducing the idle bandwidth of

[12] We are not aware of any work that discusses latency or consistency in MOGs and that does not include avatar position in its evaluation.

player nodes and reduce even further the possibility that player's machines act simultaneously both as cell nodes and player nodes. However, using this kind of overlay may be necessary to achieve consistent support for MMOFPS games. We leave this as an option to be explored in future works.

## 4.5 Server-cell reliable messaging service

In some situations, an individual cell node will communicate individually with servers. For instance, whenever a cell node needs to find out the IP address of other nodes. However, sometimes a *cell* has to send a message to the server, and vice-versa. This occurs when the server cannot trust the information from a single cell node and instead has to be informed by a given *quorum* of cell nodes about a message before it considers the message to be valid. Let $N$ be the number of primary cell nodes in a cell. Let $Q_{CTS}$ be the required quorum of primary cell nodes that agree on a cell-to-server message before the server will actually receive the message, that is, pass the message to server-side application code for parsing. As with most other quorum values, the value of $Q_{CTS}$, as well as the value of $N$, are left for the application to determine. We recommend $Q_{CTS}$ to be at most $N - 1$ to avoid single malicious cell nodes from being able to break the quorum and thus prevent all cell-to-server communication before that situation is detected and the cell has a chance to 'fail' and recover[13].

Once the server receives one cell-to-server message from a single cell node, it will create a data structure which tracks the completion of the quorum. That data structure is indexed by a message hash and the ID of the originating cell (e.g. its coordinates). The data structure also receives a (wall-clock) time value upon creation, which is the time when it should be deleted. The structure is *only* discarded when that time limit is reached, regardless of whether and when the quorum is reached. That time limit is determined by the application. If the tracking structure receives enough message copies from enough distinct primary cell nodes, the message is 'received' and sent to the application. That fact is registered in a flag in the tracking structure, which lingers until it times out. This is to prevent late messages from other nodes, after the quorum is reached, from being considered as a new message. When the tracking structure times out, the flag is checked to see whether a silent discard is done (message was delivered) or if a failure of some sort has occurred. If the latter, we recommend that the server (the manager for that cell) should just run a full recovery procedure for the cell (see Section 4.3.5).

The opposite communication direction is simpler. When the server wants to send a message to a *cell*, it just sends packets to each primary cell node individually and directly. Since the server is a trusted party, there is no need to gather a quorum. However, applying the message into the shared cell state may require synchronization. The cell nodes can use the regular BSS synchronization mechanism used to synchronize player input to indirectly agree with a round number (simulation time) in which the incoming server message should be executed by all replicas. As usual, the event is only executed in the conservative state when all primary cell nodes conservatively agree on a time for that event. This implicitly prevents a potential security problem where a primary cell node would forward fake server messages to the other cell nodes.

Server-to-cell messages is the means by which the primary cell nodes of a cell will obtain a collective and conservatively-synchronized list of all current primary cell nodes

[13]This will become clearer in a following section which deals with the fault tolerance mechanism that ensures that each cell behaves as expected in the long run.

of all neighbor cells. Of each one, the server will tell the IP address and UDP port of each, and also their IDs and security certificates, if any. This mechanism is also used to obtain a conservatively-synchronized view of all current primary cell nodes of the cell in question (the one receiving the message), though this is not the only means by which primary cell nodes get that information. Other such lists, dissociated from the conservative state, are necessary to implement the early connection steps by which the cell nodes connect to each other. For inter-cell relationships, no such lists are necessary and any information is to be retrieved from the conservative state only. This use of the conservative state and server-to-cell messaging to have conservatively-synchronized lists of nodes for the current cell and all adjacent cells is essential for the implementation of inter-cell reliable messaging, described in Section 4.6.

As a final remark, all point-to-point communication being considered here between primary cell nodes and servers is using *reliable delivery* streams. Also, it should be noted that the source of cell-to-server messages is, probably, the execution of a simulation step by the conservative simulator of each primary cell node. In the stop-and-wait protocol, a conservative step is very likely to take place at approximately the same time at all replicas. That step is also deterministic, thus, it follows that the conservative state update logic is the best place to take a simultaneous decision such as deciding to send an identical message to the server. That sort of guarantees that the 'timeout' value for the message quorum tracking structure at the server can be proportional to network round-trip times and network clock synchronization errors only. Five or ten seconds should be more than enough for the timeout value and shouldn't cause any significant side-effects elsewhere.

## 4.6 Inter-cell reliable messaging service

In this section we describe how to achieve what we call *inter-cell reliable messaging*. Currently, we have it so that each cell will arbitrate over the state of its own set of objects. Each object in the game is the property of one cell, though its state is communicated by the cell that owns it to the neighbor cells in some cases, as discussed in Section 4.2.5. Inter-cell reliable messaging is a basic mechanism or 'primitive' that the FreeMMG 2 middleware provides that allows the implementation of other higher-level functionalities. One such functionality is transferring objects between cells in a fault-tolerant, cheat-resistant and decentralized (peer-to-peer) way, by allowing cells to communicate directly in a peer-to-peer fashion as if the cells were single nodes and not a set of nodes each.

To be able to transfer object ownership between cells reliably, there must be a way for two adjacent cells to perform some sort of negotiation so that objects can be removed from one cell and be added to another cell. One way to accomplish this would be to have the originating cell send the object to a trusted server, and have the trusted server send the object to the destination cell. That could be implemented using the simple *Server-cell reliable messaging* primitives described in Section 4.5. That is precisely what we did in FreeMMG (CECIN; BARBOSA; GEYER, 2003). That, however, increases the involvement of server-side resources in the game. What's worse, that involvement becomes dependent on game design: the game designer has to worry about the amount of objects it creates and their speed to avoid flooding the servers with traffic. In FreeMMG 2, we decentralized most of the object transfer task. To do that, first we have designed a basic mechanism that allows one *cell* to send a message with guaranteed delivery to the other *cell*. It is achieved by having a sufficiently high number of nodes in the originating cell send *the same message* through reliable point-to-point IP-based communication to a

sufficiently high number of nodes in the destination cell.

### 4.6.1 Service overview and preliminary definitions

Figure 4.9 will help to illustrate what this service is about. The service allows any cell S to send a message to any other cell D. What we have here is a point-to-point protocol that allows two groups of nodes, S and D, to exchange a message in a virtual, unidirectional channel that delivers messages reliably and in an ordered fashion. In a FreeMMG 2 network, a cell will have two such channels, one outgoing and one incoming, for each neighbor cell[14]. The API is very simple:

- On S, a *Send* function allows the cell to send a message. The function returns an integer which is the sequence number assigned to the outgoing message;
- On D, an *Incoming* callback is called so that the cell can receive the message. The ID is the same assigned to the message at the time of sending;
- On S, an *Ack* callback is called when S knows that D has acknowledged the receipt of a given message ID.

The Send function can be called only from the logic that updates the state of the conservative state of the cell. That is, during a conservative 'tick' or conservative simulation time (step) advancement. This guarantees that, generally, all primary nodes of the cell will agree that a message has to be sent to the destination cell. Likewise, the Incoming and Ack callbacks are invoked during middleware code execution that is triggered due to the conservative state of the cell being updated due to conservative simulation time advancement.

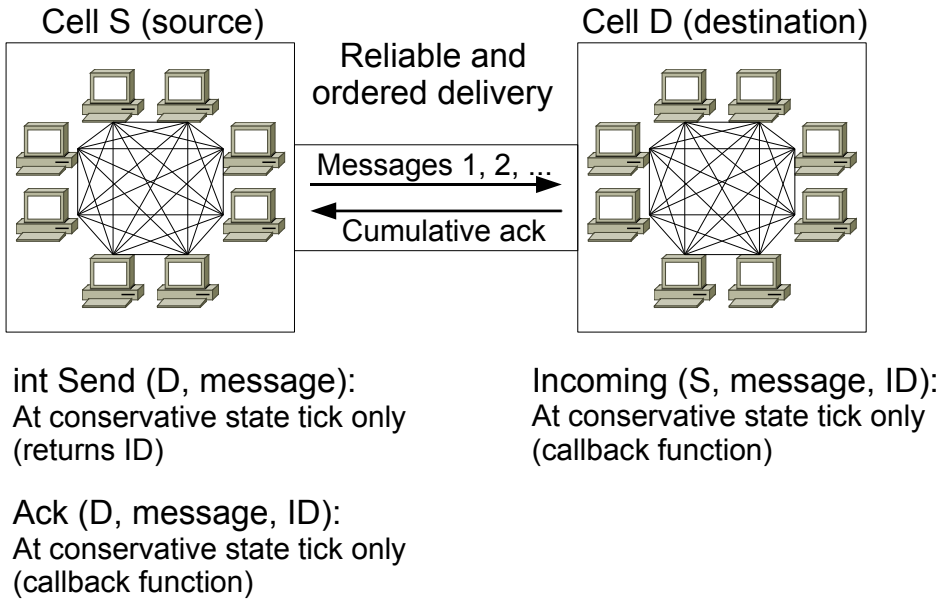


Figure 4.9: Inter-cell reliable messaging service (overview)

The Send function is implemented partially by adding the outgoing message to the cell's conservative state. That is, the queue of non-acked, pending, outgoing messages is

[14] In the event that non-neighbor cells have to interact, they should do so through the servers.

stored at the cell's conservative state. The application does not have to actually read it. This only means that the middleware uses the same mechanism to implement this internal functionality (as well as others). An implementor may choose to create another separate state and call it, e.g., 'conservative middleware state'. That's an implementation detail that doesn't concern us, for now. Other key feature required to implement this service and that is stored on conservative state memory is the list of current (valid) primary cell nodes of both S and D, at both S and D. There are others which we will describe as we unroll the service description.

Since the outgoing message queue is cell state, and cell state can be lost, this means that messages once sent through Send calls may be lost if S fails. Also, after the message is delivered and processed through the Incoming callback at D, the state of D can also be lost, resulting that the message is delivered and received, but its results are lost. Thus, this messaging mechanism does not offer actual long-term guarantees (e.g., durability). This example is analogous to any distributed system where two nodes S and D communicate through a TCP/IP stream. Even if D receives the message and acknowledges to S, this doesn't guarantee that D won't later fail and lose the side-effects of processing the message. What the protocol should guarantee (it is our intent, at least) is ordered and guaranteed message delivery while both cells do not lose any of their conservatively-synchronized state.

To achieve guaranteed persistence in applying the side-effects of inter-cell messages at D, D has to save portions of its state at the server, and any persistence is only obtained for those portions of cell state that are saved and when they are saved. The idea is that, instead of performing any and all inter-cell messaging through the server, the application programmer will instead determine when the servers have a too-outdated view of the D's state and have D send digests of cell state (*vital cell state* updates) to the server. Upon a failure that requires the *full recovery from server-side data* algorithm to run, the amount of such losses are then minimized. In short, the *reliable* in *inter-cell reliable messaging service* means just that packet loss, regular peer churn and a few malicious nodes alone won't prevent the message from being delivered and acknowledged.

All actual communication carried out by the service implementation is over the basic inter-cell synchronization protocol described in Section 4.2.5. That is, at any given cell tick event at S, at most a single UDP packet is outgoing from one of S's primary cell nodes to one of D's primary cell nodes. Thus, this service competes with any outgoing object state updates (unreliable delivery blocks) from the application for space in these outgoing packets. We assume that the time between any pair of primary cell nodes of both S and D exchanging inter-cell synchronization UDP packets is bounded to a few tens of $CTICK$s (proportional to a cell's $N$ or number of primary cell nodes). Assuming $CTICK$ is at most 100ms, this means that any two given nodes should exchange these UDP packets once one or two seconds. From that, we can establish that the service would take at least a few seconds to deliver and acknowledge each message. That is assuming enough bandwidth to transmit the messages. Needless to say, inter-cell reliable messages should be as small as possible to avoid overflowing the sender.

Our proposed implementation of the service does not assume that the basic point-to-point socket protocol supports (adds) reliable messaging over UDP. Even if the socket protocol *does* support it, we believe it is better to not assume this. Thus, our algorithm, described in the next section, assumes only UDP for the communication between cell nodes of adjacent cells. One main advantage of this is in avoiding any doubts related to configuring connection timeouts, measuring RTTs to decide message re-send times,

overhead of managing a large mesh of connections to all adjacent cells and their primary nodes, etc. All this is aggravated by the discussion of the previous paragraph, which was about the large time between packets in each one of the numerous cross-cell socket connections of all primary cell nodes. Having said that, the algorithm certainly can be simplified if a reliable protocol over UDP is to be used. The main difference then would be in the elimination of the variables with 'P2P' in their names and associated logic (see next section). We leave that as exercise to the reader.

### 4.6.2 Service description

We now explain how the service works. First, follows a definition of variables and constants that are used. These definitions make reference to each other and thus some won't be understood by sequential reading. The reader will hopefully acquire an understanding of how the service works by iterating between these definitions and the algorithm description. In the next section, we also provide an example execution of the service, which shows the algorithm execution step-by-step.

- The function *Send*, and the callbacks *Incoming* and *Ack*, are the interface to the service, as shown in the previous section;
- N is the number of primary cell nodes on each cell;
- S is the source cell (the sending end of the channel);
- D is the destination cell (the receiving end of the channel).
- $S_i$ and $D_i$ denote a node i on either S or D, where i is in the range [0,N-1];
- QDID and QDIS are constant primary cell node quorum thresholds that must hold at some node $D_i$, during its execution of a conservative simulation tick, in order for a pending incoming cell-to-cell message to be officially received by the cell (that is, to be handed to the application through an *Incoming* callback). A node $D_i$ considers a pending message to be received when at least QDID nodes at D have informed to the cell that at least QDIS nodes at S have sent the message directly to them;
- QSP is a constant quorum threshold. It is used to detect when a node $S_i$ receives at least QSP direct notifications from different D nodes that report D's current OUTC2CACK above S's current value for INC2CACK. When that occurs, the $S_i$ node in question decides that it is likely that the acknowledgement value is valid and commits an acknowledgement vote to S in favor of raising the value of the cell's INC2CACK;
- QSC is a constant quorum threshold. It is used to detect, during S's conservative tick, whether the value of INC2CACK should be raised or not. Once at least QSC nodes from S vote in favor (see the QSP constant above) of raising INC2CACK, it is raised. Whenever the INC2CACK cell variable is raised from a value X to a greater value X', this has the effect that all pending outgoing messages at S with message IDs in the interval [X+1,X'] are acknowledged at S (that is, S knows that D has received these cell-to-cell messages and thus calls the *Ack* callback);
- Each $S_i$ keeps two transient data structures which are private to each node and which need not to be synchronized with that of other nodes: INP2PACKS and INC2CACKS;

  - INP2PACKS means ‘incoming peer-to-peer cumulative cell message acknowledgements’. It is an array of integers which stores the largest P2P message acknowledgement received from each $D_i$. It is just an optimization which allows each $S_i$ to avoid re-sending messages that a specific $D_i$ has already received through the inter-cell UDP packets that flow from S to D as described in Section 4.2.5;
  - INPEERC2CACKS means ‘incoming cell-to-cell cumulative cell message acknowledgements from peers’. It is an array of integers which stores the largest cell-to-cell message acknowledgement received from each $D_i$. This is necessary since, when a single $D_i$ informs a single $S_i$ that D is acknowledging a cell-to-cell message ID, that is not necessarily a consensus in D ($D_i$ may be malicious). Thus, this array is stored at each $S_i$ to track the stance of all $D_i$ nodes about the current cumulative cell-to-cell message ID acknowledgement marker;

- Each $D_i$ keeps a transient data structure which is private to the node and which needs no synchronization: OUTP2PACKS, which means ‘outgoing peer-to-peer cumulative cell message acknowledgements’. It is an array of integers which stores the highest cell-to-cell message ID received directly from each $S_i$ so far. At a given $D_i$, its value of OUTP2PACKS for a given $S_i$ determines the value it sends back to that $S_i$ which is then stored at its INP2PACKS structure at the $D_i$ index (see above the description of INP2PACKS);

- Each $D_i$ keeps, as part of its conservative state, the following variables: INMSGPENDING and OUTC2CACK;

  - INMSGPENDING is the list of *possible* incoming messages from S. For instance, when a single $S_i$ node sends a cell-to-cell message through an UDP packet to some $D_i$, that message will end up being stored at the cell’s collective (replicated) INMSGPENDING state variable. But since that $S_i$ may be a malicious node that has fabricated that message, D will gather more such individual node messages from S before it actually receives the message and forwards it to the application (through the *Incoming* callback). These partial messages that trickle in from S are conservatively stored at the collective (replicated) INMSGPENDING structure. Once a message is successfully received, or if its gathering operation times out, it is removed from INMSGPENDING;
  - OUTC2CACK is an integer variable that stores the ID of the largest cell-to-cell message from S that was received by D thus far. Whenever a $D_i$ node sends one (or more) UDP packets to an $S_i$ node due to a network tick occurring at D, the value of that variable is sent. Thus, this is an ‘outgoing cell-to-cell acknowledgement’. Once a majority of nodes at D manages to send this value to a majority of nodes at S, S is able to confirm the amount of its cell-to-cell messages that D has received so far and thus clean up its OUTMSGPENDING structure;

- Each $S_i$ keeps, as part of its conservative state, the following variables: INC2CACK, OUTMSGPENDING, OUTC2CID and INC2CACKS;

- INC2CACK is an integer variable that stores the greatest outgoing message ID acknowledged by D. For instance, if INC2CACK is 10, then all outgoing cell-to-cell messages from S to D with an ID less than or equal to 10 are acknowledged and no longer have to be retransmitted to D;
- OUTMSGPENDING stores the cell-to-cell messages that are buffered for sending to D and that haven't been acknowledged by D yet. A message with a given ID is removed from OUTMSGPENDING when the local value of INC2CACK is greater or equal than the message ID, meaning that the message is acknowledged by D;
- OUTC2CID is an integer variable that simply stores the ID to be used by the next outgoing message from S to D. Whenever a new message is sent, OUTC2CID is incremented by one;
- INC2CACKS is an one-dimensional array that has one entry for each $S_i$ node. Each entry is an integer value which represents the current committed value (committed to the cell through conservative synchronization), by each $S_i$, of its perception of incoming OUTC2CACK values that are sent by $D_i$ nodes and that are greater than the current INC2CACK value. In other words, when an $S_i$ node receives several OUTC2CACK values from a majority of D nodes, and that value is greater than the current INC2CACK value, that $S_i$ node is convinced that D is acknowledging more messages and commits a vote to the cell to raise the INC2CACK value. That vote is stored in the INC2CACKS array. When a majority of positions in INC2CACKS agrees with a higher value for INC2CACK, the latter is incremented;

Figure 4.10 shows a fragment of the service implementation in C++ pseudo-code. The service is to be built on top of the BSS/TSS simulator and many other pieces of infrastructure which won't add anything by being included here. Figure 4.10 shows only part of the essence of the functionality that is added by the inter-cell reliable messaging service.

Figure 4.10 shows the simple implementation of the Send function. It can only be called from inside code that executes a conservative state 'tick' logic at S. Thus, this guarantees that the execution of the Send function has identical side-effects at all nodes at S. That is, assuming that there are no malicious nodes at S or some other kind of failure that prevents the execution at each $S_i$ from being identical. In that case, there are the other mechanisms that detect whenever the replicas at S are not all agreeing with a common evolution of the conservative state. The same mechanism that is used to ensure that the game state is properly replicated is used to ensure that these replicated variables used by inter-cell messaging service are properly replicated. In the event that they go out of synchrony, this is detected by one or more nodes by way of exchanging conservative state hashes and the fault recovery mechanisms described earlier are executed to restore consistency to the cell.

Successive calls to the Send function will place pending messages on the replicated outgoing message queue (OUTMSGPENDING). The variables depicted in Figure 4.10, including OUTMSGPENDING, are specific to a pair of S and D cells. If a cell has X neighbors, it will have X copies of these variables, one for each sending channel.

As discussed in Section 4.2.5, at every synchronized BSS 'tick' event of the cell, one UDP packets are sent from each cell to its neighbor cells. The function *SendData* in Figure 4.10 is what is called by BSS when it decides that the $S_i$ node (that is running the

```
// Send Logic at primary cell nodes of S for a given destination D (implicit)

// Private state (just the part required for sending)
map<NodeID, MessageID> INP2PACKS;

// Conservative state (just the part required for sending)
MessageID OUTC2CID = 0;
map<MessageID, Message> OUTMSGPENDING;

// Send a message (destination cell D is implicit).
// Necessarily invoked from the conservative simulator tick code.
int Send(Message m) {
  OUTMSGPENDING[ ++OUTC2CID ] = m;
  return OUTC2CID;
}

// Called when an Si has to send an inter-cell packet to a Di.
// Code shown is just part of the logic required to send an inter-cell packet!
void SendData(Node Di) {

  Packet p = new Packet;

  // ... more send logic ...

  int mbudget = number of message retransmissions that fit in the outgoing packet;
  int mcount = minimum (mbudget, elements in OUTMSGPENDING);

  p.put( mcount ); // How many messages we're sending to Di

  // Avoid re-sending messages that the target Di peer has already acknowledged.
  MessageID firstid = minimum (lowest MessageID in OUTMSGPENDING, INP2PACKS[ Di ] + 1);

  // Put all messages being retransmitted in the outgoing packet.
  // We attempt to retransmit all non-acknowledged messages if there is room.
  for (MessageID mid = firstid; mid < firstid + mcount; mid++) {
    Message mdata = OUTMSGPENDING[ mid ];
    p.put ( mid );
    p.put ( mdata );
  }

  // ... more send logic ...

  // Send the UDP packet p from Si to Di
  send_packet (Di, p);
}
```

Figure 4.10: C++ pseudo-code: logic required at the sender (S) to support the Send function.

code) must send one UDP packet to a given $D_i$ primary cell node at cell D. For clarity, the provided listing of SendData abstracts many aspects, such as determining the maximum size of the packet so that flow control is achieved, and splitting a single large UDP packet into many smaller packets to avoid fragmentation at the IP layer.

In essence, the SendData function implementation ensures that any messages still in OUTMSGPENDING at the time the function is called are re-sent, in order from lowest ID to highest ID, to the chosen $D_i$ node every time an outgoing inter-cell synchronization packet is being sent from S to D. Assuming a working BSS algorithm at both S and D, this logic tends to a state where all $S_i$ nodes will have communicated at least the first pending message in the cell's OUTMSGPENDING queue directly to all $D_i$ nodes. In other words, each $D_i$ node will receive the exact same message directly, through a secure point-to-point channel, from every $S_i$. If that happens, an additional consensus step at D would be sufficient to 'receive' the message, which is precisely what we do. The pseudo-code for that, which runs at each D node, is shown in Figure 4.11 and Figure 4.12.

```
// Incoming (receive) Logic at primary cell nodes of D for a given source S (implicit)

// Private state (complete).
map<NodeID, MessageID> OUTP2PACKS;

// Conservative state (complete).
class MsgVote {
  map<NodeID, map<NodeID, Hash> > votes; // Inner are S node IDs, outer are D node IDs.
  map<Hash, Message> versions; // Stores the full message for each known hash.
}
map<MessageID, MsgVote> INMSGPENDING;
MessageID OUTC2CACK = 0;

// Callback function implemented by the application. Source cell (S) is implicit.
// Must be invoked from the conservative tick function.
int Incoming(MessageID id, Message m);

// Function called whenever a Di receives an UDP packet from a Si. Di is 'this' node.
// Code shown is just part of the logic required to receive an inter-cell packet!
void ReceiveData(Node Si, Packet p) {

  // ... more receive logic ...

  int mcount = p.get(); // How many messages to be read.
  for (int i=0; i<mcount; i++) {
    MessageID mid = p.get();
    Message mdata = p.get();

    // Skip late vote events for messages that have already been received.
    if ( mid <= OUTC2CACK )
      continue;

    // Check whether this Di has already forwarded a vote for mid for this Si.
    if ( if exists a value mapped to INMSGPENDING[ mid ].votes[ Di ][ Si ] )
      continue; // Ignore and go to the next message.

    // Submit an 'input' to the cell in the same way that a player input would.
    // These events have no effect when applied to the optimistic state.
    Event e = new Event();
    e.put( "C2C Incoming vote" );
    e.put( Di ); // This is checked so that nodes at D cannot impersonate each other.
    e.put( Si ); // The peer that sent the cell-to-cell message to this Di.
    e.put( mid ); // The cell-to-cell message ID that was received by Di from Si.
    e.out( mdata ); // The data of that message.
    PushCellInput( e ); // Execute this event in the future, conservatively.

    // Update the outgoing ack value sent to Si (reduces retransmissions by Si).
    OUTP2PACKS[ Si ] = maximum ( mid, OUTP2PACKS[ Si ] );
  }

  // ... more receive logic ...
}
```

Figure 4.11: C++ pseudo-code: logic required at the receiver (D) to support the Incoming callback (1 of 2).

The code in Figure 4.11 begins by defining all the main state variables needed by the nodes at D to receive the cell-to-cell 'reliable' messages. The most complex structure is INMSGPENDING. It maps incoming message ID values to a struct that contains two other structures: a bi-dimensional *votes* array which maps pairs of (D,S) nodes to a message hash value, and a *versions* array which maps a message hash value to the corresponding message content.

After the variables, the ReceiveData function is shown in Figure 4.11. It simply unpacks the message retransmissions from Si and commits them to the cell as votes, making sure that it is not committing a vote that has already been integrated in INMSGPENDING,

or a vote for a message that has already been received and acknowledged, which would just waste bandwidth.

```
// Conservative simulation tick function.
// events: list of events conservatively synchronized for execution in this tick.
// Code shown is just part of the logic!
void ConservativeTick(vector<Event> events) {

  // Process events for execution in this conservative tick
  for every event 'e' in the 'events' list {
    String eventType = e.get();
    if (eventType == "C2C Incoming vote") {
      NodeID Di = e.get();
      NodeID Si = e.get();
      MessageID mid = e.get();
      Message mdata = e.get();
      Hash msg_hash = calculate_hash ( mdata );

      // Skip late vote events for messages that have already been received.
      if ( mid <= OUTC2CACK )
        continue;

      // Update the INMSGPENDING structure towards a consensus for message 'mid'
      INMSGPENDING[ mid ].votes[ Di ][ Si ] = msg_hash;
      INMSGPENDING[ mid ].versions[ msg_hash ] = mdata;
    }
    else // ... handle all other types of events ...
  }

  // Check for messages that can be received (quorum reached).
  // Since we guarantee that messages are received in the same order they are sent,
  //  we have to scan the INMSGPENDING map from lowest to highest message ID.
  for every 'mid' MessageID in the 'INMSGPENDING' map, from lowest to highest value {
    bool message_received = false;
    map<Hash, int> d_quorum;
    for every 'd' NodeID in the 'INMSGPENDING[ mid ].votes' map {
      map<Hash, int> s_quorum;
      for every 's' NodeID in the 'INMSGPENDING[ mid ].votes[ d ]' map {
        Hash hvote = INMSGPENDING[ mid ].votes[ d ][ s ];
        s_quorum[ hvote ] ++;
      }
      for every 'hs' Hash in the 's_quorum' map {
        if (s_quorum[ hs ] > QDIS)
          d_quorum[ hs ]++;
      }
    }
    for every 'hd' Hash in the 'd_quorum' map {
      if (d_quorum[ hd ] > QDID) {

        // Message received.
        OUTC2CACK = mid; // D will start acknowledging that up to 'mid' was received.
        INMSGPENDING.remove ( mid ); // Clean up all vote data for message 'mid'.
        Incoming( mid, mdata ); // Pass incoming cell-to-cell message to app.
        message_received = true;
        break;
      }
    }

    // To guarantee receipt of messages in order, must stop checking now
    //  if the current message IDs in the ascending sequence cannot be received.
    if (message_received == false)
      break;
  }

  // ... other conservative tick logic ...
}
```

Figure 4.12: C++ pseudo-code: logic required at the receiver (D) to support the Incoming callback (2 of 2).

INMSGPENDING keeps track of all message data reported by any nodes of D on behalf of any nodes of S. A Di node is constantly receiving cell-to-cell messages from Si nodes, stating that a message of ID 'mid' has a given content 'mdata'. If a malicious S node is reporting a corrupted 'mdata' version for an 'mid' that is being currently negotiated, that corrupted message version will be included in the INMSGPENDING structure, but it is unlikely that it will be accepted as the official version of message 'mid' since a minimum quorum must be reached for a message version is accepted. That filter is shown in Figure 4.12. Once a quorum is reached for a given message ID, it is forwarded to the application through the Incoming callback, and removed from the INMSGPENDING structure since it is no longer pending to be received. Finally, the OUTC2CACK counter is increased to keep track of the largest message ID successfully received by D thus far.

```
// Called when a Di has to send an inter-cell packet to a Si.
// Code shown is just part of the logic required to send an inter-cell packet!
void SendData(Node Si) {

  Packet p = new Packet;

  // ... more send logic ...

  // This should probably be at the beginning of all inter-cell packets ...
  p.put( OUTC2CACK ); // Si is constantly updated on Di's value for OUTC2CACK.
  p.put( OUTP2PACKS [ Si ] ); // Let Si know the messages that Di has already received.

  // ... more send logic ...

  // Send the UDP packet p from Di to Si
  send_packet (Si, p);
}
```

Figure 4.13: C++ pseudo-code: logic required at the receiver (D) to support the sending of an acknowledgement to S for its cell-to-cell messages that have been received by D.

Figure 4.13 shows that Di acknowledges inter-cell message retransmissions to an Si by sending the current OUTP2PACKS[ Si ] on every outgoing packet. It also shows that D acknowledges inter-cell messages to S by having every Di send their copy of OUTC2CACK to the Si's at S. As with the SendData functionality at S which sends messages, the SendData at D will tend to have every Di send the current OUTC2CACK value to every Si. Eventually, S will be able to reach a consensus on the increased value of the incoming OUTC2CACK value, which is recorded at S in the INC2CACK replicated variable.

Figure 4.14 shows what happens when a Si node receives an inter-cell packet from a Di node. Each Si node has an idea about what is the greatest cell-to-cell Message ID ever received by D, that is, what is the latest message ID acknowledgement value from D. That is stored at INC2CACKS[ Si ], which is a replicated (cell) variable. That is the 'official', synchronized position of Si on the matter. Si also has a private data structure which it uses to track a more up-to-date, but private, version of that information, which is the INPEERC2CACKS map. Notice that, in Figure 4.14, Si scans its INPEERC2CACKS maps to see if a majority (a quorum of at least 'QSP' nodes) of D nodes has reported an ack value that is greater than its current official vote. If that is the case, Si submits a new vote to the cell to raise its INC2CACKS[ Si ].

Figure 4.15 shows the final stage of the algorithm. When the conservative state execution advances at S's primary cell nodes, these will execute the vote acknowledgement events, which updates the shared data structure which keeps track of every S node's current vote about what is the current value of acknowledgement message ID marker.

```
// Ack receive logic at primary cell nodes of S for a given destination D (implicit)

// Private state (just the part required for receiving acknowledgements)
map<NodeID, MessageID> INPEERC2CACKS;

// Conservative state (just the part required for receiving acknowledgements)
MessageID INC2CACK = 0;
map<NodeID, MessageID> INC2CACKS; // Indexed by S nodes.

// Callback function implemented by the application.
// Must be invoked from the conservative tick function.
int Ack(MessageID id, Message m);

// Function called whenever a Si receives an UDP packet from a Di. Si is 'this' node.
// Code shown is just part of the logic required to receive an inter-cell packet!
void ReceiveData(Node Di, Packet p) {

  // ... more receive logic ...

  MessageID p_pack = p.get();
  MessageID p_cack = p.get();

  INP2PACKS[ Di ] = maximum ( INP2PACKS [ Di ], p_pack );
  INPEERC2CACKS[ Di ] = maximum ( INPEERC2CACKS[ Di ], p_cack );

  // Check whether this Si is convinced that a majority of nodes at D is
  //  raising the cumulative acknowledgement ID for cell-to-cell messages
  //  relative to Si's last perception (INC2CACKS[ Si ] -- not the cell's
  //  perception of the acknowledgement).
  int quorum = 0;
  MessageID minraise = INFINITY; // Value that is never the smaller among two.
  for every 'mid' MessageID in the 'INPEERC2CACKS' map {
    if ( mid > INC2CACKS[ Si ] ) {
      quorum++;
      minraise = minimum ( minraise, mid );
    }
  }
  if (quorum > QSP) {

    // Commit a vote to the cell to raise Si's own perception of the acknowledgement.
    // Submit an 'input' to the cell in the same way that a player input would.
    // These events have no effect when applied to the optimistic state.
    Event e = new Event();
    e.put( "C2C Acknowledgement vote" );
    e.put( Si ); // This is checked so that nodes at S cannot impersonate each other.
    e.put( minraise ); // Heightened D message acknowledgement value as perceived by S.
    PushCellInput( e ); // Execute this event in the future, conservatively.
  }

  // ... more receive logic ...
}
```

Figure 4.14: C++ pseudo-code: logic required at the cell-to-cell message sender (S) to receive acknowledgements from the message receiver (D) (1 of 2).

Whenever a majority (a quorum of 'QSC' nodes) at S decides that the current value of INC2CACK is to be increased, it is increased as far as a majority of nodes agrees to. That results in the Ack callback being called for application code at S and messages being removed from the OUTMSGPENDING message queue at S.

For simplicity, it is not shown in all the pseudo-code listings the logic that cleans up these data structures whenever nodes are removed from S or D. For instance, if a node Di fails, all entries related to Di have to be removed, to avoid Di's votes from helping to receive a message, for instance. These cleanups are to be performed at the ConservativeTick function, at least for the conservatively-synchronized, replicated data structures such as INMSGPENDING.

```
// Conservative simulation tick function.
// events: list of events conservatively synchronized for execution in this tick.
// Code shown is just part of the logic!
void ConservativeTick(vector<Event> events) {

  // Process events for execution in this conservative tick
  for every event 'e' in the 'events' list {
    String eventType = e.get();
    if (eventType == "C2C Acknowledgement vote") {
      NodeID Si = e.get();
      MessageID mid_raise = e.get();

      INC2CACKS[ Si ] = mid_raise;
    }
    else // ... handle all other types of events ...
  }

  // Check for a raise on outgoing MessageID acknowledgement (quorum reached).
  int quorum = 0;
  MessageID minraise = INFINITY;
  for every 's' NodeID in the 'INC2CACKS' map {
    if (INC2CACKS[ s ] > INC2CACK) {
      quorum++;
      minraise = minimum ( minraise, INC2CACKS[ s ] );
    }
  }
  if (quorum > QSC) {

    // Acknowledge all messages between old and new value of INC2CACK.
    for (MessageID mi = INC2CACK + 1; mi <= minraise; mi++) {
      Ack( OUTMSGPENDING[ mi ] , mi ); // Notify application
      OUTMSGPENDING.remove ( mi ); // Remove message from the outgoing queue
    }

    // Advance the acknowledgement marker at S
    INC2CACK = minraise;
  }

  // ... other conservative tick logic ...
}
```

Figure 4.15: C++ pseudo-code: logic required at the cell-to-cell message sender (S) to receive acknowledgements from the message receiver (D) (2 of 2).

### 4.6.3 Considerations on selecting values for the quorum thresholds

We have not researched into adequate values for the quorum thresholds QDID, QDIS, QSP and QSC, as we were first trying to come up with any algorithm that seemed correct and workable (an initial solution). In the initial version of this algorithm we simply demanded that every individual node notified every other node correctly, which would be equivalent to setting $QDID = QDIS = QSP = QSC = N$, where $N$ would be the constant, global number of primary cell nodes allocated to each cell. That was problematic for several reasons. The main problem was that a single malicious primary node in either the source or destination cell could start preventing any message from getting across simply by ignoring the messages. Even with an outer mechanism to detect and correct such situations, the power for a single primary node to individually interrupt the flow of messages between any cell and all its neighbors is too damaging in a vandalism (griefing) sense. Therefore, some faulty pairwise negotiations between individual primary cell nodes would have to be tolerated, and thus the quorums were introduced as to allow future works to experiment with them.

On the other hand, the $QDID = QDIS = QSP = QSC = N$ thresholds are the safest against state cheating. As we will see in the next section, the reliable inter-cell

negotiation mechanism is to be used for implementing object ownership transfer between cells – that is, to support one object moving from one cell to another. The lower the quorum thresholds above are, the easier it is for a given quantity of malicious nodes to transfer nonexistent objects to adjacent cells. If these objects can be made to be arbitrarily valuable (e.g. treasure chests in an RPG with arbitrary amount of treasure), this could break the economy of the game.

So, we do not know exactly how far we *can* go into reducing the quorums without opening it up for state cheating, and we do not know how far we *must* go into reducing the quorums to avoid a small minority of malicious nodes from stopping transfers altogether. We do not also know if the quorum variables could be combined into fewer global configuration variables for the same or similar effect. For instance, maybe they can all safely be set to a byzantine quorum (LAMPORT; SHOSTAK; PEASE, 1982), or maybe not. Due to our primary focus on cheat prevention, we consider that evaluating this will more important, initially, than evaluating the performance of the protocol, which is also required and which we have also not done. We leave these for future work.

### 4.6.4 Summary and additional remarks

In the previous sections, the basic communication overlay and its basic protocols were described. One of such basic protocols was the regular inter-cell synchronization packet stream (Section 4.2.5), which created a series of UDP packet streams between pairs of primary cell nodes of distinct cells. In this section, we defined a 'reliable messaging service' (a higher-level protocol) which, building on top of these inter-cell packet streams, provides the means for adjacent *cells* to signal each other in a fairly reliable and cheat-resistant way.

The algorithm's worst case scenario, in a performance (messaging cost) perspective, is when $QDID = QDIS = QSP = QSC = N$, as per the previous section. In this case, the cost of the algorithm is $O(N^2)$ messages, which is not efficient. However, we believe the algorithm still can be made to be feasible for several reasons. First, $N$ (number of primary nodes in a cell) is already small enough. Second, inter-cell messages should be no longer than a few tens of bytes (including the service overhead). Third, it doesn't have to guarantee the timeliness of message delivery. Fourth, the cost can be amortized by reducing the quorum thresholds. And fifth, individual messages do not increase the amount of UDP packets sent; instead, all messages are aggregated to packets that are already outgoing, maximizing gains from packet compression and minimizing packet overhead.

However, any logic that depends on this service will certainly have to be frugal in the amount of messages generated and their size. We can, for now, ignore the timeliness of delivery, but we can't ignore the fact that, on average, less bytes must be sent than what the individual sender and receiver nodes can cope with (flow control). Again, we have not prototyped this service nor evaluated how much bandwidth would be available for inter-cell reliable messaging. In this way, we cannot currently shed light on how many bytes per second of inter-cell traffic, in practice, a game designer would be able to generate. We only know that, with the current design, it could turn out to be as low as 1% of the upload bandwidth available to the average primary cell node, or maybe even less. We're guessing that a cell could generate something in the order of a few hundreds of bytes per second of outgoing inter-cell messages for each adjacent cell. Certainly, there are games that could already be made to work with this kind of constraint.

The algorithm, as it is defined, does not guarantee liveness in the presence of cell

recoveries (from cell failures) that cause the conservative simulation time to decrease. For example, consider a destination cell D which has, in its conservative state, OUTC2CACK = 500 for a sender adjacent cell S. Now, consider that D has failed and the server recovers an old conservative snapshot where OUTC2CACK = 400. Now, it is very likely that the unidirectional communication channel between D and S is broken, since D will be waiting for message 401 to be received before it will forward newer messages to the application (due to the ordered delivery constraint). Conceptually, this scenario is no different than a socket disconnection. However, we have not defined the service to cope with the channel failing, so either these virtual channels have to be checked and fixed when a failure occurs, or a proper 'inter-cell socket' abstraction and API must be created, with accompanying code that handles these 'cell socket' connections and disconnections, etc. For now, the simplest solution is to have the cell managers query the potentially broken cell nodes and fix the inter-cell reliable messaging channels as needed. As this is 'leaf' functionality that doesn't affect anything else, and that there are several ways to address this (including a redesign of the service), and that there are enough open ends already, we have left it for future work.

We see two immediate ways to optimize the service. The first would be to simply have some $S_i$ nodes issue hashes of messages instead of the contents of such messages, either at random or in a coordinated way. For large messages, this could save a significant amount of bandwidth, since when any message copy is received by the destination cell, no more full message copies are required, only hashes which are in practice votes that confirm that the message is legit. The caveat is that the hash function would have to be a secure one, and secure hash functions produce large hashes. As a reference, SHA-1 produces 160 bit (20 byte) hashes. So, if the message is smaller than the hash, the message would always be sent instead. The overhead introduced by this optimization would be a single bit to tell if a node is sending a hash or a message.

Another easy optimization would lie in implementing a public-key cryptosystem, having each $S_i$ sign its outgoing messages with its private key, and having each Di validate the signatures using the sender's public key. In this way, the messaging inside D can be greatly reduced, as now a single $D_i$ can forward a message from an $S_i$ to its cell peers as definitive proof that $S_i$ approves the message. Once a node at D receives the signed vote from $S_i$, it will no longer forward the same message to D should it receive it from $S_i$ directly, and it will also update its own OUTP2PACKS structure as if it had received the message directly from $S_i$. If all that is implemented, the QDIS quorum check can be eliminated entirely. The trade-off would be the added requirement that a public-key cryptosystem be present, which is not a requirement imposed by any other module in the FreeMMG 2 architecture so far.

Describing an inter-cell reliable messaging protocol in this section was important because one of the key motivations of FreeMMG 2 lies in allowing cells to reliably transfer objects between each other, and to allow some high-consistency interactions to be resolved, though not timely, directly between cells. In the next section we show how these mechanisms could be implemented, assuming that a working and validated inter-cell reliable messaging protocol is present to support them.

## 4.7 Supporting complex inter-cell object interactions

So far, it is not clear how objects can be made to move freely around the virtual world. As it currently stands, our description of FreeMMG 2 suggests that objects would belong

to their originating cell forever. An object can move into the terrain of a cell that is adjacent to its original cell, but ownership would be retained at the original cell.

It is also unclear so far how more complex interactions could be supported. For example, if a 'missile' object from a cell S collides with an 'avatar' object from a cell D, how to resolve that interaction? What if the collision is detected only at S, and what if it is detected only at D? What if it is detected at both S and D but at each cell's different simulation times which bear no correlation to each other?

These questions point to the so-called complex object interactions across multiple cells. These are places in simulation 'tick' logic where a cell has to *write to foreign objects* as well as to its own objects. In other words, objects owned by different cells have to modify each other's state. Simple interactions are the ones that do not require this. For instance, if a cell S detects that its own object OS is going to collide with a foreign object OD owned by an adjacent cell D, it can simply move OS out of the way (assuming that the collision has no side-effects such as projectile detonations and such). That may not be the optimal solution in terms of visual quality in some cases, but it is simpler. This example could be turned into a complex interaction if S decided to move OD out of the way instead.

We have defined three main protocols that can be used to support these complex cell interactions: the inter-cell reliable messaging service (Section 4.6), the inter-cell synchronization protocol (Section 4.2.5) and the server-cell reliable messaging service (Section 4.5). These protocols were primarily built for security and not low latency. Thus, any mechanism built over these tools will certainly perform worse in terms of visual consistency and interaction response times than regular interaction between objects owned by the same cell. That is especially significant because the better-performing support of intra-cell interactions was also not fully optimized for latency. Future works that attempt to decrease latency and increase visual consistency in FreeMMG 2's complex inter-cell interaction support should therefore start improving on the three fundamental protocols described above.

Thus, in this section, we will describe how a few in-game situations can be made to work across cell borders using the previously described protocols. Our main goal with this section is to show that, *assuming that the relevant support protocols could be shown to work*, it would be possible to have adjacent cells coordinating to achieve *correct* and *cheat-resistant* interaction across their borders with a reduced involvement of server-side resources.

In the remainder of this section it will be assumed that the application will model the state of a cell as a collection of objects. The FreeMMG 2 core functionality does not require this. Core functions only handle pieces of game state as opaque blocks of bytes. However, most games benefit from having the state modeled as an object-oriented system. That is also easier to visualize, which is why we will assume that for this and the following examples. In an implementation of FreeMMG 2, we would recommend an intermediate 'object module' between the core protocols and the application. Such module would be responsible for layering the object metaphor on top of the basic functions for the application to use. That module could then implement object transfer services that would work for several types of game objects to be supported.

### 4.7.1 Maintaining global consistency in a peer-to-peer cell-based MMOG

In the original FreeMMG model, we have identified what we called a 'global consistency problem' (CECIN et al., 2004). In FreeMMG, whenever a segment (same as a cell)

failed, its state was restored to the last snapshot (segment state copy) that the peers running the segment had sent to the server. However, if a segment S had a given object X when its last snapshot was taken, that means that a restore of S from that snapshot would restore the presence of X to S, disregarding the case where the object X had already moved out of S. That global inconsistency scenario is illustrated by Figure 4.16. By 'global' we mean that, although consistency is maintained inside individual cells (segments), it is broken if the collection of cells is considered. In the figure, a duplicated object is shown, which is inconsistent since, obviously, objects should not be erased or duplicated gratuitously by the middleware.

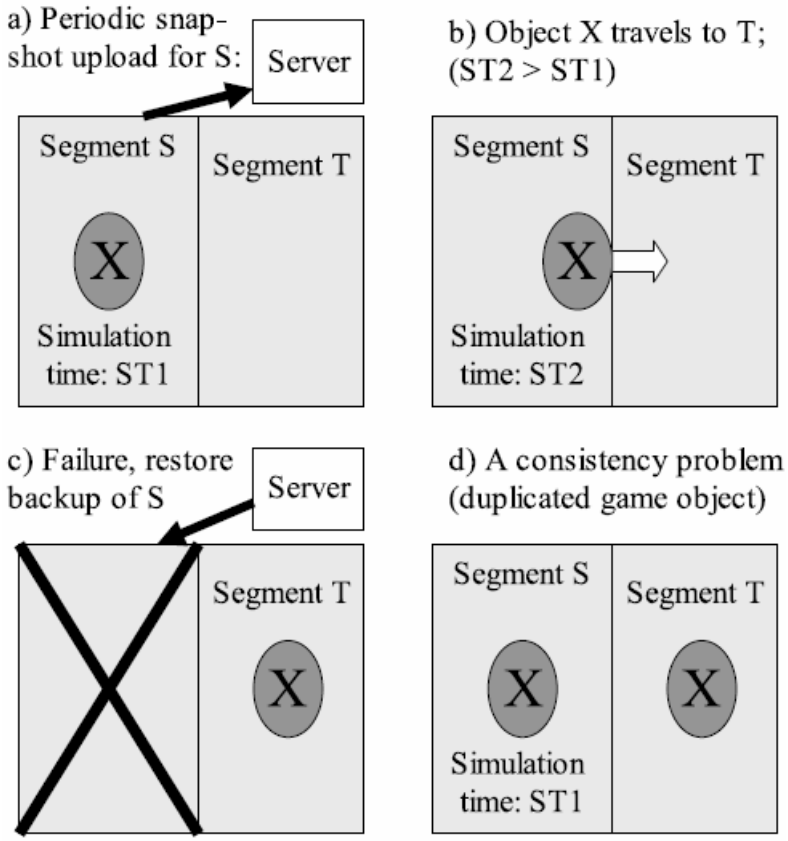


Figure 4.16: Global consistency problem in the original FreeMMG model.

In FreeMMG 2, there is the same problem. The main difference is that instead of blindly backing up the full state of all cells at the server, FreeMMG 2 lets the application decide which objects are 'important' and thus should be regularly backed up and protected from loss or duplication, and which objects aren't important and thus should be left to be duplicated or lost upon failures. The important objects can be then backed up at the server-side both periodically and when they are about to be transferred from one cell to another. The collection of important objects and variables of a cell constitute what we call its 'vital state'. The vital state has the same purpose as the back-up snapshots from FreeMMG: to restore the state of cells from server-side data whenever needed.

The core realization here is that we do not have a mechanism that truly guarantees the consistency of the game state stored at the cell replicas at all times. It will be always possible, though generally unlikely for adequate replication and quorum parameters, that a cell is composed exclusively of malicious colluding nodes that set its state to arbitrary

contents. It would be trivial for the members of a compromised cell to not only fabricate arbitrary amounts of virtual wealth, but also to break the game's global consistency. For instance, a compromised cell may create a copy of an unique item, avatar or NPC that is already located elsewhere on the virtual world, or delete such object if it is currently located at the compromised cell.

In essence, *the only way to completely guarantee global consistency is to store cell data at the server-side*. Thus, when a cell manager receives a new update for a cell's vital state record, it should check whether that update breaks global consistency or not. For example, if a compromised cell commits a snapshot to its server-side cell manager that contains a duplicated unique object, the cell manager has the means to verify if that unique object is fake. One way that could be done would be by checking in with every other cell manager or with a central database of unique objects to see whether that unique object reported by the compromised cell is already registered as being in another cell. If the unique object is found elsewhere, the state reported by the compromised cell is rejected.

Thus, the server-side infrastructure has the means to always guarantee invariants over the global game state. However, using server-side data to restore cells is supposed to be only a last-resort measure. Ideally, the cell nodes themselves will be able to coordinate and prevent cell state from being corrupted by unintentional or intentional failures. The goal of keeping the essential parts of the game state consistent at servers is only to have a fall-back mechanism. Whenever there is any doubt whether the state of the cells is consistent or not, the server-side data can be checked to dissolve any inconsistencies found at the client-held cell state storage network.

In the next sections, we will describe how to support some complex interaction between objects owned by different cells. Each scenario will be explained both for when the affected objects are not important (no server-side involvement) and for when the objects are important and thus cannot be lost or duplicated (with updates to server-side vital cell state). In the latter, we explain how the server-side cell managers can use the 'vital cell state' to protect the important objects from a permanent situation of global inconsistency.

### 4.7.2 Example 1: transferring an object between two cells (no server)

A complex interaction that is probably required in any game is moving 'game objects' between cells. That is, transferring the ownership of a game object between two cells. Even more specifically, deleting an object from a source cell's replicas and inserting a copy if it at the destination cell's replicas.

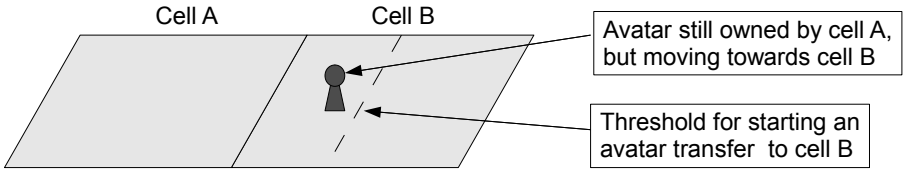


Figure 4.17: In-game situation that precedes the following object transfer examples.

Figure 4.17 shows the starting state of the in-game situation that we want to support. A single avatar object, owned by Cell A, has already moved inside the virtual terrain that comprises Cell B. One should remember that objects are allowed to be owned by a cell and be positioned at an adjacent cell. In the figure, the avatar is moving further into the territory of Cell B. After it crosses a certain threshold (to be set by the application), the

avatar will be transferred to B. Instead of handing off objects as soon as they cross the borders, a threshold is used to reduce the amount of object transfer operations between cells. Ahmed et al. (AHMED; SHIRMOHAMMADI; OLIVEIRA, 2007) features a conceptually identical mechanism, and explains it in greater detail.

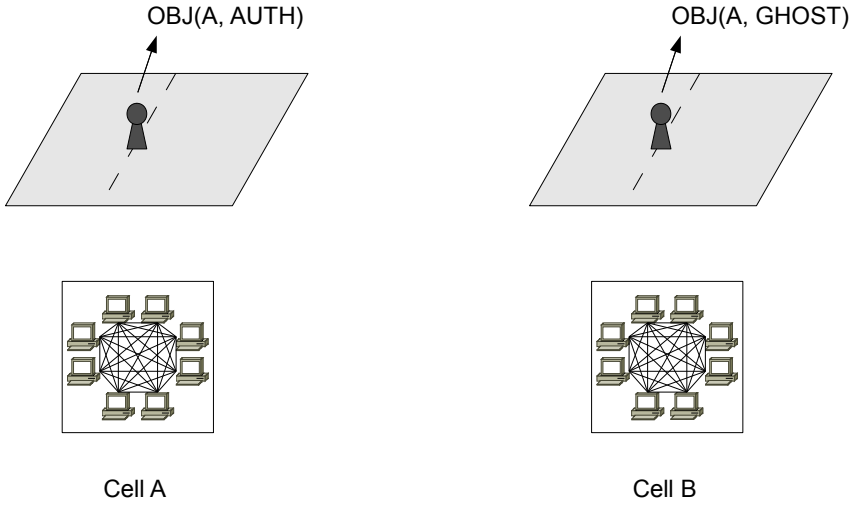


Figure 4.18: State of cells A and B prior to the object transfer being triggered.

Figure 4.18 shows the avatar that is switching cells as a game object OBJ. The terminology used in Figure 4.18 will be used in the following figures as well. In parentheses after the object name, we describe where (in which cell) that copy resides (either A or B) and its role. The possible roles are: AUTH, meaning the authoritative copy of the object (meaning the specified cell owns it); GHOST, meaning a copy of an object owned by an adjacent cell; and TEMP_AUTH, meaning that the specified cell has started to delegate ownership of the object to an adjacent cell but hasn't completed the operation yet (this means the specified cell has partial ownership over the object).

Figure 4.18 shows the state of cells A and B before the transfer is triggered. For simplicity, the current and following figures will show only a single, simplified state for the cell, abstracting away replication and the movement of the object. In Figure 4.18 the game state is stable. Both the optimistic and conservative states of all cell nodes are the same. All cell nodes at A contain the authoritative copy of OBJ, while all cell nodes at B contain the ghost copy of OBJ.

In Figure 4.19, OBJ crosses the threshold at the conservative state of Cell A and the transfer process is initiated. Due to this, two actions are executed at all primary nodes of Cell A:

- First, OBJ is marked as a 'temporary authority'. To this end, it is marked by the application (or object management layer) with the state TEMP_AUTH. The only reason for a TEMP_AUTH object state is to avoid objects blinking in and out of existence during the object transfer procedures which, being based on the inter-cell reliable messaging service, can take several seconds to complete. The application must ensure that any side-effects to game state caused by a short-lived TEMP_AUTH object cannot cause the game's global consistency to break;
- Second, Cell A sends a message to Cell B informing that the game object 'OBJ' (whose relevant state is provided in the message) is to be now owned by Cell B.

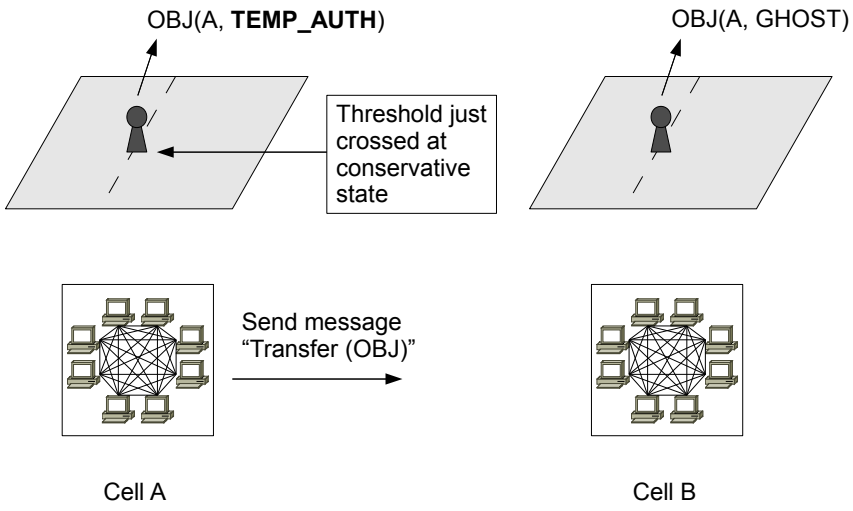


Figure 4.19: Object transfer triggered at Cell A during the conservative simulation step.

> Note that all such inter-cell messages in the examples use the reliable inter-cell messaging service.

In other words, OBJ is now temporarily 'locked'. The application will be responsible for ensuring that the behavior of OBJ is now minimal. For instance, an NPC may keep being moved by the AI, but it may be prevented from attacking or taking damage, for instance. The extent to which restrictions apply to TEMP_AUTH objects is to be determined by the application. Alternatively, important modifications to the TEMP_AUTH object can be forwarded to Cell B through inter-cell messaging. Then, after the object arrives, the pending modifications that were committed after the transfer operation could then be applied. For now, we assume that TEMP_AUTH objects just keep moving and animating, and that arbitrary movement and animation commands are not meaninful to the long-term consistency of the game state.

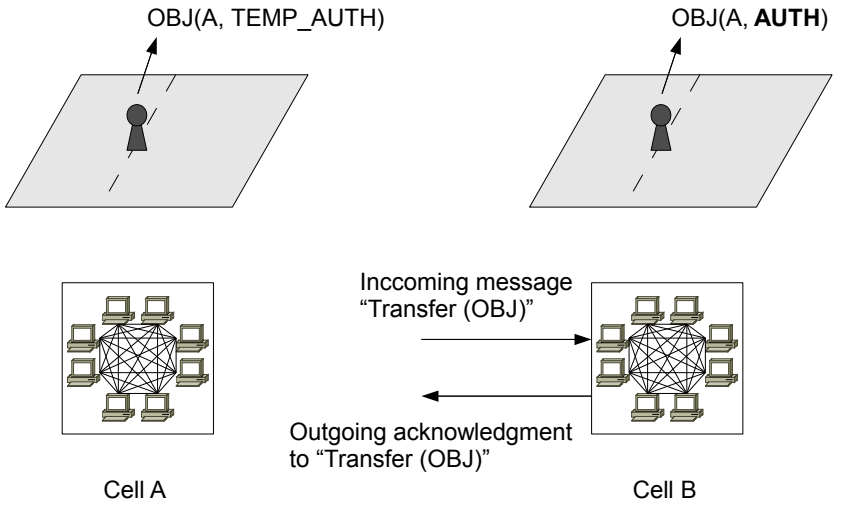


Figure 4.20: Object transfer message received by Cell B.

Figure 4.20 shows Cell B receiving the inter-cell reliable message. Upon receiving

the message, Cell B replaces its ghost copy of OBJ with the authoritative copy. The application can either simply use the object state informed in the message or, which seems more reasonable, merge the state informed in the message for OBJ with the latest updates from the TEMP_AUTH object at A. For instance, while the transfer was being negotiated, the object may have already moved a lot. Using the state from the ghost copy would provide a better visual result.

Now, the cell nodes that can observe the state of both cells A and B will receive updates from two masters: the TEMP_AUTH object at A, as well as the AUTH object at B, are both sending object updates for neighbor cells and player nodes. Observers that can perceive both updates (such as a common adjacent cell) should drop the updates from TEMP_AUTH and use the ones from AUTH only. Others, such as player nodes, will use the one that is available. Naturally, Cell B ignores updates from the TEMP_AUTH version of the object now that it knows it holds the authoritative copy. Cell A can now behave as a hybrid of master and ghost copy, obeying (consuming) the updates from Cell B to correct its object (as a ghost copy would), but still propagating OBJ updates (as a master copy would).

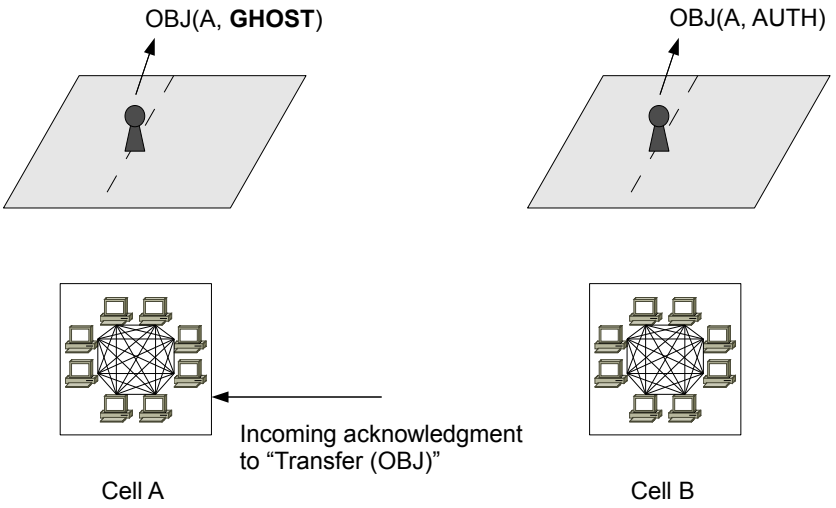


Figure 4.21: Cell A receives the message acknowledgement from Cell B, completing the transfer.

Finally, in Figure 4.21, Cell A receives the confirmation that Cell B has received the object transfer message. Since the message is unconditional, all cell nodes at A can now remove OBJ. Actually, instead of deleting, the optimal action is to convert OBJ into a GHOST at Cell A, since it should already be receiving updates from the AUTH version of OBJ at Cell B.

This mechanism works consistently because the sending, receiving and acknowledging of these inter-cell messages are reliable, ordered, and happen always inside conservative simulation step code. However, the mechanism described can lose or duplicate OBJ if there is a cell failure, as discussed earlier. In the following section, this problem is addressed with additional steps that can be taken whenever OBJ is an important object that should not be lost or duplicated.

### 4.7.3 Example 2: transferring an object between two cells (with server)

To transfer important objects that cannot be lost or duplicated, it becomes necessary to contact the servers to inform whenever an object moves from one cell to another. To achieve this, the mechanism described in the previous section is preceded with some additional steps which we present below.

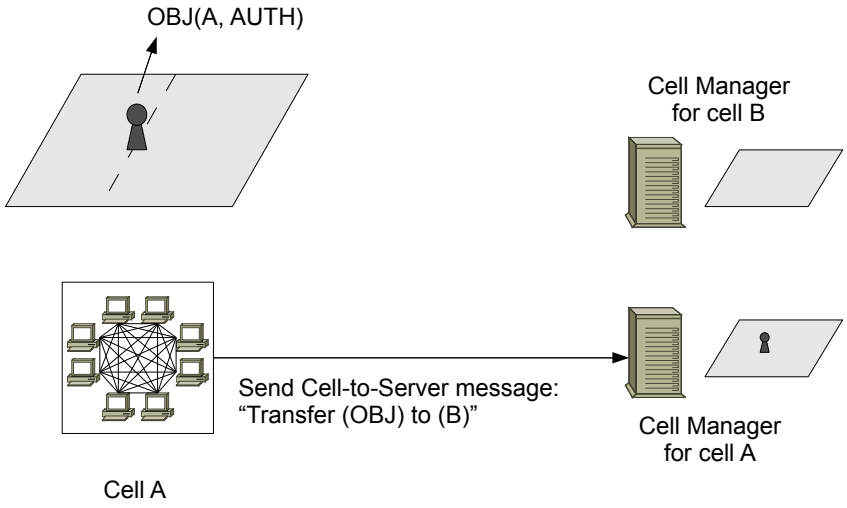


Figure 4.22: Object transfer registered at the server-side first (1 of 3).

Figure 4.22 shows the first step, which is having the cell send a reliable message to its cell manager telling that OBJ is moving to Cell B.

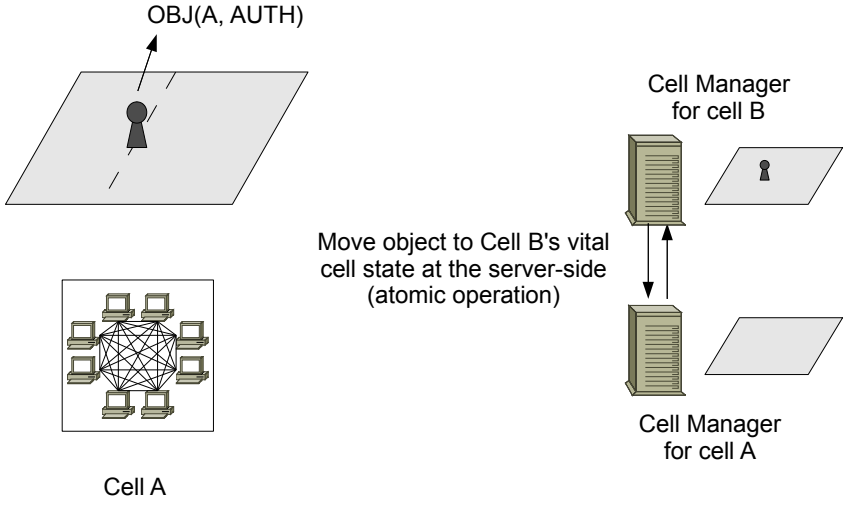


Figure 4.23: Object transfer registered at the server-side first (2 of 3).

Figure 4.23 shows the second step, which is an internal synchronization between the cell managers of cells A and B. Basically, the vital cell state of both cells must be updated atomically, moving OBJ from one cell manager to the other. That simple negotiation will guarantee that, should the state of cells A and B be lost, OBJ will be restored to exactly one of them (Cell B).

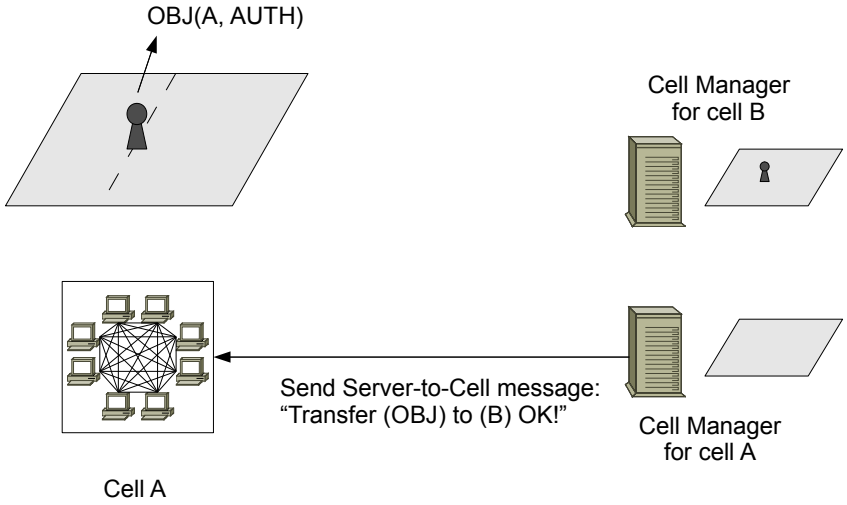


Figure 4.24: Object transfer registered at the server-side first (3 of 3).

Finally, in Figure 4.24, the primary nodes of Cell A receive, inside conservatively-synchronized logic, the notification that OBJ has already been moved to Cell B at the server-side. Now, the peer-to-peer transfer specified in the previous section may be initiated.

An alternative to this would be transferring important objects through the servers only, bypassing the need for the more complex peer-to-peer object transfer protocol presented in the previous section. However, that would double the bandwidth cost at the server per transfer. First, Cell A would notify the server of the departure of OBJ to B, changing OBJ at Cell A to the state TEMP_AUTH. Second, the server would notify Cell B of the insertion of OBJ. Third, Cell B confirms to the server that OBJ is inserted. And fourth, the server notifies Cell A to change the object's status from TEMP_AUTH to a GHOST. The solution presented in the figures above requires only two messaging steps with the server, instead of four. However, the server transfer method may be simpler to implement and may resolve faster.

### 4.7.4 Example 3: picking up a foreign object

Besides transferring objects due to movement, objects may have to be transferred for other reasons. In this section we consider a scenario where an avatar tries to pick-up an object that is owned by a different cell.

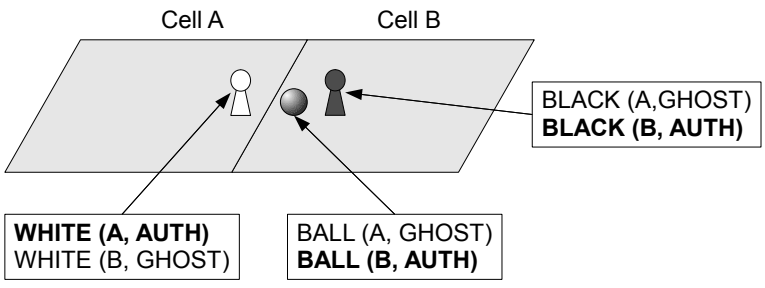


Figure 4.25: Starting scenario for the concurrent ball pick-up example.

Figure 4.25 shows the scenario. The WHITE object (an avatar) is owned by Cell A, while its ghost version is perceived at Cell B. The BLACK object is owned by Cell B, while its ghost version is perceived at Cell A. Finally, a BALL object that is to be disputed by the two avatars, is located at Cell B together with BLACK.

Now, the example consists of considering what happens if WHITE tries to retrieve the ball. How that is modeled depends on the application. Let's consider that the object is deleted upon retrieval and that some attribute of the avatar that picks it up is modified such as an integer counter of 'balls retrieved'. Using that modeling, one way to allow WHITE to retrieve the ball would be the following:

- First, an invisible object BALL' is created at Cell A, at both the conservative and optimistic states. It links both BALL and WHITE. Whenever an update is generated from WHITE, BALL' causes the update to send an increased 'balls retrieved' attribute (assuming that it can be viewed by third parties). BALL' also works as a negator of the BALL object: both Cell A and Cell B, upon having a copy of it (authoritative or ghost), suppress the sending of updates for the BALL object to observers such as players and neighbor cells. Effectively, the creation of BALL' works as a speculative (and easily reversible) mechanism for trying to retrieve BALL;

- After that, Cell A sends a reliable inter-cell message to Cell B that requests that the ownership of BALL be passed to Cell A;

- Upon receiving the request, Cell B may provide two different answers:

  - If some other avatar such as BLACK has already retrieved BALL by the time the request is received, the request is denied and a message is sent from B to A notifying that BALL is not available. In that case, when Cell A receives the negative reply, the BALL' object is simply deleted, effectively undoing the tentative (optimistic) retrieval operation performed by Cell A;
  - If BALL is available, then B transfers it to A. Upon receiving ownership of BALL, both BALL and BALL' are deleted, and the effect of ball retrieval is executed upon WHITE (e.g., increase by one the counter attribute of balls retrieved at the WHITE object).

Note that this applies both for important and non-important objects. In the case of important objects, the transfer of BALL to Cell A is done with updates sent to the cell managers as explained previously.

The solution presented above may cause some other complications. For one, WHITE cannot be transferred to another cell while BALL' exists. If that was possible, BALL could be transferred to Cell A but the avatar would have already moved to other cell. That could be addressed by creating a BALL" at wherever WHITE moves in and so on, but the benefit of this is questionable. Holding the avatar at the cell while the transfer is pending would be simpler. However, that presents yet another complication. If Cell B suffers a fatal failure while Cell A has BALL' and is waiting for the answer from B, then Cell A may end up stuck without an answer and a left-over BALL' object that disallows WHITE from moving out of Cell A forever. A solution to that would be having the BALL' object time out or re-send the request for transferring BALL to Cell A. Actually, a method or behavior of the BALL' object could be left responsible for drawing the remote object in, in the true spirit of object orientation. These issues are to be resolved by whatever

layer creates the concept of objects in the simulation, using the communication primitives we have described earlier. Studying the issues that arise from object-oriented game state modeling is left for the application or for future work in a reusable object simulation layer.

### 4.7.5 Example 4: projectile collision and damage (a Quake 'rocket')

Finally, let's consider another typical scenario in an avatar-based game: some sort of projectile object is hurled towards an avatar in an attempt to destroy the latter. Figure 4.26 shows the scenario setup. A MISSILE object owned by Cell A is about to collide with an AVATAR object owned by Cell B. The main difference in this example is that no object transfers are required. The objects will only be deleted or modified in some way.

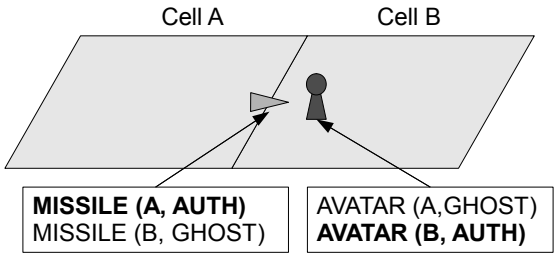


Figure 4.26: Inter-cell complex interaction between a rocket and its victim.

We will now assume, for simplicity, that the avatar is not moving and that there is no way for the rocket to miss its target at any simulator in the network. In that case, we would recommend the following solution:

- When a collision is detected at any cell node of Cell B, nothing happens;
- When a collision is detected at any cell node of Cell A:
  - First, the MISSILE object is transformed into an explosion at the point of impact. That object should stay alive long enough to allow Cell A to perceive the transformation through the inter-cell unreliable object updates mechanism. This allows for Cell B to perceive the explosion as quickly as possible, without the overhead of the inter-cell reliable messaging;
  - Second, Cell A calculates the damage that is to be done on all affected avatars. For local avatars, Cell A applies the damage instantly. For foreign avatars, Cell A sends reliable messages to the relevant neighbor cells informing which of their objects were damaged and by how many damage points. In our example, a single inter-cell reliable message from Cell A to Cell B is sent, informing that AVATAR has suffered damage proportional to a direct MISSILE hit;
- When (and if) the transformation of the MISSILE object into an animating explosion is perceived at Cell B, it generates damage special effects and animation as quickly as possible for AVATAR. That is, other players may see AVATAR being tossed around the room and hear a sound clip of pain, and the player controlling AVATAR may additionally see its screen blink or tilt, etc.;
- Finally, Cell B receives the reliable damage message and applies it to the avatar.

This solution presents two main advantages. For one, individual cell nodes cannot cheat and fabricate damage messages. Second, timely events such as visual rocket detonations and avatar reactions resolve relatively quickly. The main disadvantage is that avatars may take some time from being hit to dying. There may be ways to make this visually acceptable. Also, there are certainly many other possible solutions that trade off between security, bandwidth usage, latency and consistency. If the FreeMMG 2 primitives are actually implemented, future works could certainly experiment with several different solutions for supporting the different kinds of interactions.

## 4.8 Closing remarks

In this chapter we have provided a description of the FreeMMG 2 middleware model. We started by describing the basic protocols. Then, we considered how cells could detect and recover from failures. And finally, we explored how reliable and ordered group communication could be achieved while still keeping the middleware secure against state cheating and also bandwidth-efficient.

We cannot say at this time whether our model, as described, is of practical use or not. The only definitive test would be to finish and deploy a game that uses a working version of a middleware whose implementation is based on the model. Even then, we believe some aspects of the model would be changed due to practical insights that would be inevitably gained during implementation. Thus, FreeMMG 2 is still a work in progress. The model presented in this chapter is just the result of a lot of reasoning and a bit of experimentation. No more, no less. In the next chapter we present a partial validation of FreeMMG 2, mainly through a network simulation.

# 5 VALIDATION

In this chapter, we argue for the feasibility of the FreeMMG 2 model. Ideally, we would be able to implement all of FreeMMG 2, tie it to a very representative piece of game logic, and then test the resulting software in order to validate several aspects of the model such as bandwidth usage, interaction latency, latencies between visual inconsistency occurrence, detection and recovery, the correctness and feasibility of the fault recovery procedure, resistance to state cheating, etc. However, our validation by experimentation of the model is partial: we have not used experiments to cover for all aspects that would be interesting to inspect and validate.

At first, it seems it would be natural to prioritize the security-related aspects of the model for validation since that was our primary focus during its design. However, that was not our priority for the experimentation phase because there are already other results in the literature that point towards the feasibility of replication as a fault-tolerance and anti-cheating measure. With that out of the way, the second most important aspect for evaluation would be the bandwidth efficiency of the model, which we obviously cannot draw from existing results. Thus, that has become the focus of our experiments. More specifically, we have studied the bandwidth usage at client nodes (primary and backup cell nodes, and player nodes) through simulation.

We have used the Simmcast framework (BARCELLOS; FACCHINI; MUHAMMAD, 2006) to implement the basic parts of the FreeMMG 2 model as a network simulation[1]. The simulator logic includes not only middleware logic but also the game layer logic. For the game logic, we have modeled a very simple 2D world were player avatars move using a Random Walk algorithm. With that simulator we then evaluate whether the amount of bandwidth consumed at the client nodes of the architecture (cell nodes and player nodes) is realistic or not. To that end, we perform simulations on a series of scenarios with different amounts of cell nodes per cell and different amounts of total player nodes in the virtual world.

## 5.1 Simmcast functionality used

We have built our FreeMMG 2 simulation on top of the Simmcast network simulation framework (BARCELLOS; FACCHINI; MUHAMMAD, 2006). Simmcast is a Java-based, multi-threaded, discrete event simulator. The Simmcast simulator is not distributed, that is, a single experiment run cannot be cooperatively executed by multiple machines, but a single Simmcast process is able to simulate a distributed system com-

[1]The source code of the FreeMMG 2 simulator is available at http://sourceforge.net/projects/freemmg/ (module 'freemmg2' on public CVS). The Simmcast website is at http://www.simmcast.org/.

posed of thousands of hosts.

At the heart of any network simulation lies the representation of the physical network. Simmcast supports any complex network topology consisting of *host nodes*, which are the actual participants in the distributed system, and *router nodes*, which together represent the autonomous routing infrastructure that allows the host nodes to exchange data packets with each other. Unidirectional network links between nodes can have a given bandwidth, a chance of packet loss, and an emulated latency which can follow one of the probability distributions available for use. Simmcast supports multicast, but for our simulation we only used its unicast support.

Each host node in the system must have its behavior defined in a class, and running the behavior of a host node requires at least one thread. Simmcast supports the sending and receiving of network packets between the simulated host nodes (threads). When sending a packet, a host informs the destination and the payload, which is a Java object, which is then retrieved by the receiver if the routing is successful. With those basic pieces from Simmcast, we have built our FreeMMG 2 simulator.

## 5.2 The FreeMMG 2 network simulator

In our simulator, the physical network is abstracted into a star topology. The center of the star is a single *router node*, named INTERNET in the code, which represents the Internet routing infrastructure. All other nodes in the simulation are *host nodes* which connect bidirectionally to the INTERNET router node. Since we are not particularly interested in what happens when congestion or flow problems appear, modeling Internet routing details did not seem to be essential.

In practice, a flow control problem in FreeMMG 2 would mean that a host is misconfigured such that the bandwidth it reports to the middleware as available is incorrect. Regular Internet congestion is to be tolerated through sensible time-out parameter tuning, and severe congestion is not to be tolerated at all. A node with severe congestion simply isn't interesting to the network and should fail. We partly model connectivity problems through the emulation of packet loss and the fluctuating latency of the links.

The simulator does not model the entering and leaving of nodes. The physical network configuration is static from start to finish of a simulation run. Node failures and attacks from malicious nodes were also not simulated. From a bandwidth assessment perspective, these are not essential. The model, as discussed in the previous chapter, indicates that cells keep running normally when a client node fails. The failure of individual cell nodes decreases the game *quality* but otherwise does not increase bandwidth usage beyond whatever the cell nodes themselves have advertised as available.

Our simulator builds a network of WORLD_WIDTH $\times$ WORLD_HEIGHT cells, where WORLD_WIDTH and WORLD_HEIGHT are integer simulation parameters which define the dimensions of the world, which is a rectangular grid of cells. Each cell is modeled as a collection of simulated primary cell nodes (PCNs) and backup cell nodes (BCNs). Each cell contains 2 $\times$ NCELL cell nodes: NCELL PCNs and NCELL BCNs. Thus, NCELL is the simulation parameter which specifies the amount of (primary, backup) node pairs that every cell requires to run. The cell overlay network and other aspects of the FreeMMG 2 logical connections are entirely our additions. We did not use any of the Simmcast group communication facilities. Finally, the TOTAL_PLAYERS input parameter specifies the number of player nodes (controllers of individual avatars) in the simulation. The player population remains constant and no players join or leave the network

during the simulation. We have not modeled server nodes in the simulation.

The connection overlay guarantees that packets between cell nodes of the same cell (primary-to-primary and primary-to-backup) arrive despite packet loss. This emulates the reliable messaging streams required for the BSS messages. Packets in these logical links are re-sent until an acknowledgement is received by the original sender. Retransmissions of reliable packets are separated by an amount of time equal to 1.25 times the round-trip latency between sender and receiver (as per simulation parameters). The receiver sends out one acknowledgement packet immediately when it receives one reliable packet retransmission, regardless whether the receiver logic has already forwarded the packet to the overlay application or not. So, even if both the reliable packets and the acknowledgements packets are sometimes lost, a reliable packet is eventually both delivered at the intended recipient and later acknowledged at the original sender.

On top of the modeled FreeMMG 2 connection overlay, there is a simplified version of the intra-cell synchronization (BSS), back-up node updating and inter-cell synchronization protocols. Communication is implemented as a reliable packet exchange inside the cell (BSS) and between PCNs and their BCNs, only. Reliability is implemented as a simple re-sending of packets that are not acknowledged in a fixed time window (we have added the acknowledging and re-sending logic). Player nodes have their avatars owned by one cell at a time. Player nodes send one stream of command packets to randomly-chosen PCNs of their current cells, and receive one stream of update packets from its cell's PCNs in a round-robin fashion. Each PCN keeps two copies of the game state, conservative and optimistic, and an event table to resolve the pending conservative steps. We did not implement the rollback and re-execution procedure as the game we modeled on top of the simplified BSS did not need it.

And finally, on top of the BSS, primary-backup, inter-cell and player-cell protocols, we have placed a very simple 2D game logic. In our game protocol, each player is given the ability to dictate its own position. All avatar-related messages (commands or updates) are 32 bytes long, which we particularly think is too big but, then again, it could be even larger such as the 200 byte messages found on the EverQuest protocol (NORDEN; GUO, 2007). So, we think that choice of size is a good trade-off. Players send 32 byte update messages and receive back a list of 32 byte update messages inside each packet, comprised of updates to its own avatar and all avatars that the PCN that sent the update knows to be in the player's AoI, which is a cirular area around the avatar of radius SOI_RADIUS.

The primary cell nodes of one cell also notify the primary cell nodes of all neighbour cells about avatars of interest, as specified by the inter-cell synchronization protocol. There is also an Area of Interest (AoI) filtering in this case, as PCNs of the source cell only notify neighbour PCNs about local avatars that are of potential interest to players owned by the neighbour cells. That is also calculated from SOI_RADIUS, but is instead a rectangular area calculated inside the cell that has the interesting objects. That area is the part nearest the cell that is interested in the objects. In other words, it is as if the interested cell looked up a distance of SOI_RADIUS deep into the cell that it wants to know about, calculated from all points on the border. Thus, the result is approximated as a rectangular area with SOI_RADIUS as its smaller dimension (either horizontal or vertical, or both) at all times.

To maintain simplicity, the player nodes themselves decide when they change cells by locally evaluating their own position. Player nodes move around the world using a Random Walk algorithm at a specified constant velocity (a simulation parameter). When changing cells, players simply stop sending updates to the old cell and start sending up-

dates to the new cell. The conservative state of the cell detects when a player stops sending updates to it for several consecutive cell ticks and, in that case, the avatar times out and is removed. Both the optimistic and the conservative state of a cell keep separate the list of avatars owned by the cell (that is, avatars being updated directly by player nodes) and ghost avatars, which are avatars hosted by neighbour cells. Finally, since players are responsible for determining their current cell, we have not modeled the inter-cell reliable messaging service nor the reliable object transfer procedure. These are unaccounted for in the bandwidth results.

Even with the simplifications in BSS, the absence of server nodes, and the lack of inter-cell reliable messaging in the simulation, we consider that have modeled the bulk of the packet load specified in the previous chapters in several separate protocols. We assert that the bandwidth figures we can obtain from the flows we did model are sufficient to draw some a conclusion about the general feasibility of the FreeMMG 2 model in regards to peer bandwidth usage.

## 5.3 Simulation scenarios

The following are invariants in all the simulation scenarios:

- Mean link latency. All scenarios use a 25ms mean latency with a standard deviation of 5ms. Note that a 25ms average latency in our star topology means an average one-hop latency in the overlay of 50ms, and an average 100ms latency for the round-trip time between any two overlay nodes;
- WORLD_WIDTH = 5 and WORLD_HEIGHT = 5, resulting in a 25-cell game world. There are no borders in the virtual world: as an avatar reaches a corner of the grid, its coordinates wrap around. That is, the virtual world is a torus;
- CELL_SIZE = 4096 units. Each cell is an $4096 \times 4096$ square;
- SOI_RADIUS = $\frac{CELL\_SIZE}{4}$, or 1024 units. Each avatar is only interested in other objects that are inside a circle of a 2048 unit radius;
- Avatars move at $\frac{CELL\_SIZE}{32}$ units per second. In all our scenarios, that means 128 units/s. Thus, an avatar would take at least 32 seconds to traverse a cell moving either horizontally or vertically;
- Messages sent through the physical links take a random amount of time to reach the destination. The random packet latency generator uses a normal distribution. During the simulation, all links share the same mean (varies between scenarios) and the same standard deviation which is always 20% of the mean value;
- Links are configured for emulating 2% packet loss;
- Links have 1 Gbps of bandwidth available for downloading and uploading packets. This is the maximum traffic that a node can generate. We have set it to a high value since we wanted to measure how much bandwidth the model demands. If we had limited it, the simulator could end up queuing packets, which would mask the real bandwidth demand. And since we are not studying the behavior of the system upon paths becoming congested or links overflowing, there is no reason to set this parameter to a realistic value;

- Links can enqueue up to 1000 packets. For instance, the INTERNET node can hold up to 1000 outgoing packets on each of its links to each host node;
- All nodes tick, that is, run their main logic and send out batches of packets, at 100ms intervals. That includes players sending commands and cell nodes sending BSS synchronization messages, updates to players and updates to neighbour PCNs;
- Packets sent to players from primary cell nodes are limited to around 1500 bytes (avatar updates are added to outgoing packets until the 1500 byte limit is exceeded during packet construction). Since players receive up to 10 packets per second, players may receive an inbound traffic of up to, approximately, 15 kilobytes per second. That should be sufficient for many types of games. In the event that players have more avatars in their SoI than they can be updated about, the servers (cell nodes) should decide which updates are more important and send those;
- Likewise, inter-cell synchronization packets are limited to one per cell tick and to a maximum size of about 1500 bytes. Even if the area of interest of one cell, in regard to a neighbor, may be much larger than that of a single avatar, what we really wanted to discover was the *minimum* bandwidth required for a node to participate in a FreeMMG 2 network. In that context, the limit of 1500 bytes allows, approximately, 45 avatar updates of 32 bytes each to be exchanged between two neighbor cells every at cell tick (100ms intervals), which is already a significant update rate;
- The measured part of the simulation lasts 5 minutes (300 seconds) of simulation time, with an additional ten seconds at the beginning and two seconds at the end discarded from the results, for a total of 312 seconds duration in simulation time;
- All results we present are the average of 10 runs with the same parameters.

Based on the invariants above, our simulation scenarios are the following:

- Scenario 1. The NCELL parameter, or number of cell node *pairs* on each cell (primary and backup), is tested for the values 6, 8, 10 and 12. The TOTAL_PLAYERS parameter is set to 500, or a density of 20 players per cell (since there are 25 cells);
- Scenario 2. The NCELL parameter, or number of cell node *pairs* on each cell (primary and backup), is tested for the values 6, 8, 10 and 12. The TOTAL_PLAYERS parameter is set to 2000, or a density of 80 players per cell (since there are 25 cells).

For both scenarios, we measure the average and maximum upload and download bandwidth consumed by each primary cell node. We measured bandwidth usage by observing each second of simulation separately. The ‘average’ value in the charts is the average among all simulation seconds of all primary cell nodes and of all simulation runs. The ‘maximum’ value in the charts is obtained as follows. For each primary cell node, during all of the simulation, the simulation second which yielded the maximum bandwidth usage is considered that primary cell nodes’s maximum for that run. Then, for a given simulation run, the average of all those maxima is then taken as the maximum of the simulation. Finally the maximum value in the charts is the average of such values among all runs. Thus, the maximum is actually the average of all maxima among all primary cell nodes, which we believe is more representative than considering an arbitrarily large spike from a specific simulation second of a specific primary cell node.

## 5.4 Simulation results

Figure 5.1 shows the average bandwidth used by a cell node when there are, on average, 20 players to be served by each cell (500 players on a 25-cell world). With NCELL at 6, approximately 33,000 bytes/s of upload bandwidth, on average, is required of a primary cell node. With an NCELL of 12, approximately 43,000 bytes/s of average upload bandwidth is required. Though each additional cell node brings more upload bandwidth to the pool to serve players and adjacent cells with updates (inter-cell synchronization), that is not enough to offset the quadratic cost of the intra-cell synchronization protocol, thus resulting in a quadratic curve that is amortized. As NCELL increases beyond the low values shown, the bandwidth curve should approximate $N^2$. We did not test with higher values of NCELL because the time to complete simulations was already too large for a NCELL of 12. Also, because the number of cell nodes on each cell is a global static parameter, the smaller NCELL is, the less volunteer nodes are needed in the pool to enable a virtual world of a given size in cells to be served.

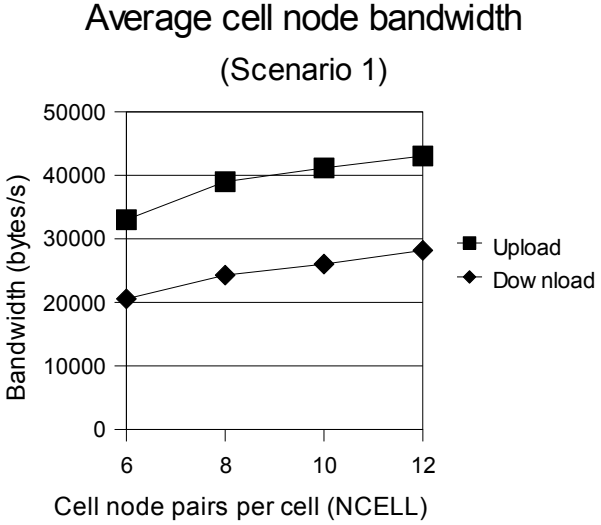


Figure 5.1: Average primary cell node bandwidth usage (Scenario 1)

Figure 5.3 shows the maximum bandwidth used by a cell node when there are, on average, 20 players to be served by each cell (500 players on a 25-cell world). For a NCELL of 6, the result is approximately 41,600 bytes/s for upload, and for a NCELL of 12, the result is approximately 56,400 bytes/s for upload. The latter is equivalent to, approximately, 450 Kbps of user traffic. Thus, it seems that, for Scenario 1, cell nodes seem to be compatible with ADSL broadband technology, though perhaps not exactly a low-end, capped connection (e.g., a 256 Kbps ADSL connection). However, it should be noted that there are several parameters that can be tuned to, perhaps, allow low-end broadband connections to be supported. For example, there can be gains with packet compression and by using smaller update messages (32 bytes can be considered too much for an update message). On the other hand, some games may require more in-game objects (e.g. NPCs, projectiles, etc.) or more frequent updates, so these results could be seen as a fair estimate. They could certainly be made either much smaller or much larger by an application programmer.

Scenario 2 does a better job of stressing the model. In Scenario 2 we set the number

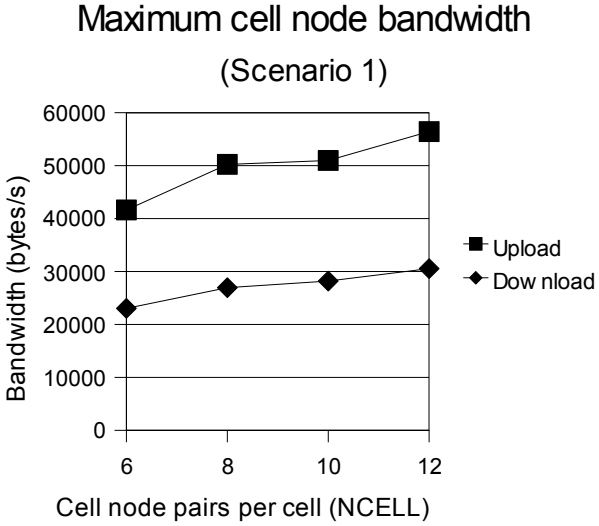


Figure 5.2: Maximum primary cell node bandwidth usage (Scenario 1)

of player nodes to 2000, or a density of 80 players per cell since there are 25 cells in the virtual world. Figure 5.3 shows the average bandwidth consumed at cell nodes in this scenario. For a NCELL of 6, the average upload bandwidth is approximately 58,100 bytes/s (about 465 Kbps), and for a NCELL of 12, it is approximately 66,200 bytes/s (about 530 Kbps). Again, the explanation for the bandwidth curves looking linear rather than quadratic is the compensating effect of sharing the player update and adjacent cell update traffic among all primary cell nodes. Again, testing with greater NCELL values should yield curves that approximate the quadratic shape.

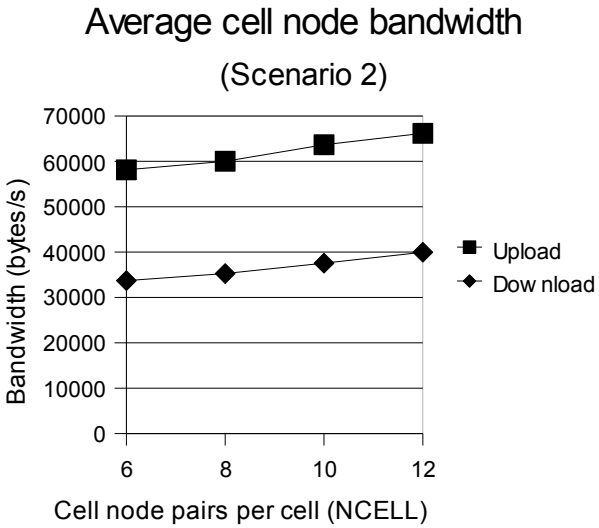


Figure 5.3: Average primary cell node bandwidth usage (Scenario 2)

Figure 5.4 shows the maximum bandwidth usage for primary cell nodes and for Scenario 2. An NCELL of 6 yields 111,800 bytes/s for maximum upload bandwidth (about 895 Kbps), and an NCELL of 12 yields 139,200 bytes/s for maximum upload bandwidth

(about 1.11 Mbps). The maximum values from Figure 5.4 indicate that, for a large player density, it may become infeasible to allocate the primary cell node functionality in a FreeMMG 2 network to nodes that have low-end broadband connections, which is not a positive result. On the other hand there is still room for improvement such as by employing compression, by sending smaller updates, or by reducing the frequency of player commands and cell updates as the player density increases in a cell.

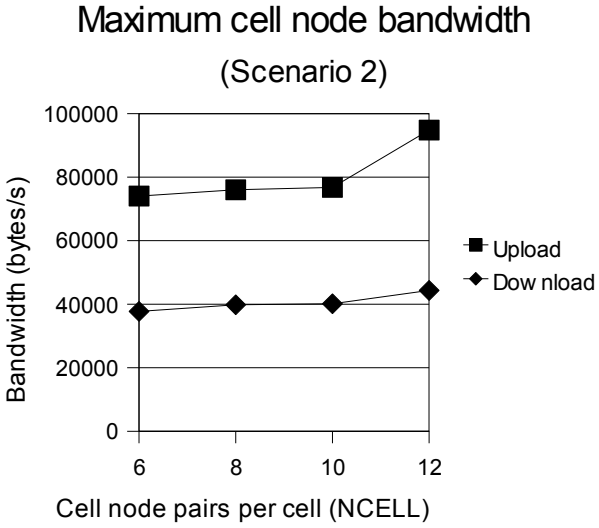


Figure 5.4: Maximum primary cell node bandwidth usage (Scenario 2)

Considering that consumer broadband continues to evolve, we can conclude that the cell nodes of FreeMMG 2 use a realistic amount of bandwidth, at least for NCELL values of 12 or less, and for an amount of players on each cell in the order of tens of players. Upcoming broadband technologies such as VDSL and VDSL2 are capable of sustaining several megabits per second of upload traffic (ERIKSSON; ODENHAMMAR, 2006). Those existing technologies would already support the load generated by our experiments, especially the upload traffic portion which is significant due to FreeMMG 2 not relying on IP multicast.

Table 5.1: Average and maximum avatar cell change events

| Amount of players | Player density | Average cell changes per second | Maximum cell changes per second |
|---|---|---|---|
| 250 | 10/cell | 10.1 | 30.3 |
| 500 | 20/cell | 19.9 | 44 |
| 1000 | 40/cell | 40.8 | 74.8 |
| 2000 | 80/cell | 76.1 | 121.2 |

As stated in the simulator description, we did not model inter-cell object transfers. In our simulation, players just advertise their avatar positions to what they decide to be their current cell. Table 5.1 shows our final experiment, where we measured the average number of times per second that player avatars change from one cell's area to another one. On these experiments, we varied the amount of players while maintaining the world size at 25 cells (so, player density was varied). The results show that avatar changes per

second increases linearly with player density. Thus, even in cell-based multiple arbiter MMOG models, hot-spots should not result in inter-cell object transfer mechanisms that do not scale, even if the bandwidth or time cost per transfer is high. This is important because FreeMMG 2's inter-cell reliable messaging, required for object transfers, is expensive, both when it is done through the server (increasing the game operator's cost) and when it is done in a peer-to-peer fashion (due to our many-to-many messaging scheme). It shoud be remembered, however, that the results shown in Table 5.1 are without the delayed zone crossing optimization (AHMED; SHIRMOHAMMADI; OLIVEIRA, 2007) (see Section 4.7.2) and that player movement was simplfied to a Random Walk algorithm. As always, the best answer to these and other uncertainities would be the evaluation of a concrete, deployed virtual world based on FreeMMG 2.

## 5.5 Resistance to state cheating

One important aspect that we have not validated through experiments is whether the replication approach, coupled with random selection of cell nodes, is enough to offer strong resistance against state cheating. However, existing works show that, as the number of replicas increases, the probability of drawing a colluding majority to a group decreases exponentially (CORMAN; SCHACHTE; TEAGUE, 2007; ENDO; KAWAHARA; TAKAHASHI, 2005). We have also performed some initial experiments which pointed out that, even in a virtual world with thousands of cells, it is reasonable to expect that, on average, less than one cell gets compromised in a period of months or years. The exact odds of a cell being corrupted depend on a host of factors such as the percentage of colluding attackers on the volunteer pool, the number of replicas, the minimum quorum for accepting peer-held cell state after a fatal failure, the threat model (attacker capabilities), and others. Future works could assess these odds considering the FreeMMG 2 architecture. In any case, with the multiple arbiter approach, it is clear that the odds are naturally in favor of state cheating resistance.

One interesting preliminary result that we obtained is that, considering that there is a large group of colluding attackers in the pool, it is very advantageous to remove cell nodes on purpose after they have served in a cell for a while. We found out for our specific simulation scenarios that four hours was the optimal maximum serving time for cell nodes at a given cell, minimizing the chance of a successful collusion at a cell. Removing cell nodes now after several hours of serving would certainly not impact the game play too much, as cell nodes can be replaced transparently by their back-up nodes. In that way, the resistance to state cheating is greatly enhanced almost for free.

## 5.6 Closing remarks

In this chapter we have presented results obtained from a network simulation of FreeMMG 2. These results allow us to have an idea about the amount of bandwidth that would be required of the peer nodes of a FreeMMG 2 network running an avatar-based game. The results show that at least the base protocols of FreeMMG 2 demand a realistic amount of bandwidth of the peer nodes, considering current broadband technology such as ADSL or Cable.

Though the validation performed so far is not ideal, we consider that it at least warrants us a comparison with related work. That is done in the next chapter.

# 6 FREEMMG 2 COMPARED WITH RELATED WORK

This chapter compares FreeMMG 2 with other seven recent peer-to-peer MMOG architectures that stand up well to our focus on decentralization with security. It also compares FreeMMG 2 with the original FreeMMG model (CECIN et al., 2004) for completeness. Through the comparison we aim to show that FreeMMG 2 is an original and relevant contribution.

## 6.1 Introduction

In Section 2.7, we have reviewed several peer-to-peer MMOG architectures. Most of these architectures were then classified into one of four categories: player mesh, fat server-side, single arbiter and multiple arbiter. We briefly review these categories below:

- *Player mesh* models achieve significant decentralization, but they become essentially peer-to-peer overlays where nodes are players and players with proximity in the virtual world become neighbors in the overlay. They usually do not address the issue of state cheating and thus are not suitable for MMOGs where the virtual economy is encoded as part of the peer-kept virtual environment state;

- *Fat server-side* models are ones where the general peer-to-peer paradigm was applied in a way to the MMOG system which did not result in sufficient offloading of computing or communication tasks to volunteer (client) machines. In other words, by still requiring significant server-side infrastructure, they do not achieve sufficient decentralization;

- *Single arbiter* models achieve significant decentralization and replicate the contents of virtual world cells among several client machines, but they become vulnerable to state cheating if the 'master' replica of a cell is compromised and thus state cheating becomes possible;

- *Multiple arbiter* models achieve significant decentralization and they also achieve resistance to state cheating and (sometimes) network-level attacks by replicating the contents of virtual world cells and not assigning a special role to any one of the replicas of a cell.

Since FreeMMG 2 is a multiple arbiter model that focuses on both decentralization and resistance to state cheating and network-level attacks, most works chosen for the comparison with FreeMMG 2 are of the *multiple arbiter* kind: Kabus and Buchmann (KABUS; BUCHMANN, 2007), Endo et al. (ENDO; KAWAHARA; TAKAHASHI,

2005), Hampel et al (HAMPEL; BOPP; HINN, 2006) and Izaiku et al. (IZAIKU et al., 2006). These are papers that, among all other reviewed works, more closely satisfy our core requirements.

However, we have also have included in the comparison two *fat server-side* models, ACORN (NORDEN; GUO, 2007) and DaCAP (LIU; LO, 2008) because they present many interesting features and they do achieve significant decentralization if compared solely to other server-intensive designs. Also, the Peer Cluster (CHEN; MUNTZ, 2006) architecture from the *single arbiter* category was chosen because it presents some interesting features in spite of its excessive reliance on a single master region controller.

Table 6.1: Meaning of the symbols used in the architecture comparison tables

| Symbol | Meaning |
|---|---|
| ✓ | Feature provided or issue addressed. |
| ✓- | Feature provided with significant limitations or issue partially addressed. |
| ✓- - | Feature provided with severe limitations or issue tangentially addressed. |
| ✓($) | Feature provided by server-side infrastructure. |
| ✓($/2) | Feature provided partially by server-side infrastructure. |
| ✓- ($/2) | Feature provided partially by server-side infrastructure, and with limitations. |
| | Blank: feature not provided or issue not addressed. |
| $N_C$ | Number of controllers (replicas) in one cell. |
| $N_P$ | Number of players in one cell. |
| $W$ | Number of *witness nodes* (GAUTHIERDICKEY et al., 2004a) in one cell. |
| $N_N$ | Number of neighbour cells that one cell has. |

The comparison is made in three separate sections. Section 6.2 performs feature comparison, Section 6.3 presents an assessment of how far each model decentralizes tasks, and Section 6.4 assesses the coverage of relevant security threats. Each section comes with a comparison table which sums up the comparison. The meaning of the symbols shown in the comparison tables is listed in Table 6.1.

## 6.2 Feature comparison

As of April 2008, about 98% of all commercial MMOGs were RPGs (WOODCOCK, 2008b). MMORPGs have two key characteristics that are important for the architecture designer. The first one is, in our view, present in any conceivable MMOG: the presence of long-term challenges. In other words, any other genre of MMOG will have this 'RPG element' simply because a massive game in a persistent-state virtual world *requires* either a virtual economy, some form of character enhancement, or some other form of long-term challenge. One of the few MMOFPS games, PlanetSide (SONY ONLINE ENTERTAINMENT, 2008), had long-term character enhancement though it didn't feature a virtual economy. RPG games have always been about long-term challenges since their inception as tabletop games. This explains in part the success of RPGs as online virtual worlds.

The second key characteristic of MMORPGs is a high tolerance to interaction latency (CHEN; HUANG; LEI, 2006; FRITSCH; RITTER; SCHILLER, 2005), especially if compared to FPS games (CLAYPOOL; CLAYPOOL, 2006). Since interaction latency is one of the many weaknesses or trade-offs of most peer-to-peer MMOG architectures, and that the MMORPG is currently the most popular genre of MMOG, most peer-to-peer MMOG architectures end up being sold as MMORPG support solutions. That includes

FreeMMG 2 and all other models chosen for comparison, as shown in the first line of Table 6.2. The only exception in the comparison is FreeMMG which aimed at MMORTS support and would probably be less adequate as a MMORPG platform due to its high interaction latency – it orders all events conservatively – and hot-spot problem. We know of no P2P MMOG model which can claim support to MMOFPS games, and this includes all models in the comparison. Hampel et al. (HAMPEL; BOPP; HINN, 2006) and FreeMMG 2 both indicates that MMOFPS support is to be investigated by future work.

Table 6.2: Feature comparison

| Feature | ACORN | DaCAP | Hampel et al. | Endo et al. | Kabus and Buchmann | Izaiku et al. | Peer Cluster | Free-MMG | Free-MMG 2 |
|---|---|---|---|---|---|---|---|---|---|
| MMORPG support | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓- - | ✓ |
| MMORTS support | | | | | | | | ✓ | |
| Inter-cell object interaction | | | | | | | | | ✓- ($/2) |
| Inter-cell transfer of objects | ✓ | ✓($) | | ✓($) | ✓ | | | ✓($) | ✓($/2) |
| Hot-spot handling | ✓ | ✓($) | | | | | ✓($) | | |
| State persistence | ✓($) | ✓($) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Authentication | ✓($) | ✓($) | | ✓($) | ✓($) | ✓($) | ✓($) | ✓($) | ✓($) |

Besides game genre support, Table 6.2 also compares support for common features. The more centralized approaches, ACORN and DaCAP, both address the problem of hot-spots, while the decentralized models do not. ACORN and DaCAP, like all the other models framed as *fat server-side* in Chapter 2, are able to handle hot-spots with much more ease than replication-based models.

- DaCAP solves hot-spots by running hot-spot cells in *C/S mode* (client-server), that is, it moves cells with high player populations to servers. DaCAP also attempts to partition cells dynamically, but it is unclear how that plays with the *C/S mode* migration mechanism;

- The Peer Cluster paper suggests that only a few tens of players are to be supported in most cells. Peer Cluster also states that temporary hot-spots (such as an army of avatars gathering at an otherwise deserted region) and fixed hot-spots (game hub locations such as virtual cities) are to be handled by the powerful commercial servers of the game service provider. This is the same that DaCAP does;

- We are not completely sure how ACORN's single (and moving) coordinator of one world region is able to is able to handle 1,000 players (NORDEN; GUO, 2007). However, serving 1,000 players in one region is effective hot-spot support by any measure. As a reference, consider that most massive encounters supported by commercial MMOGs peak at a few hundred avatars. In any case, the mechanisms for handling hot-spots in ACORN are moving the coordinator functionality to more powerful nodes and ACORN's underlying DHT which can trade-off increased latency to leverage ALM bandwidth-saving techniques such as packet aggregation and increased hops in the DHT;

- The other, more decentralized models, do not address hot-spots in any definitive way, at most tolerating them to some degree. Kabus and Buchmann states that their

model is to support *instanced games*. These don't have hot-spot problems by definition since the amount of players in each instance is limited by game rules. Endo et al.'s architecture includes a server-side infrastructure that handles *sites* (cells) by default. Their idea is that sites are to be off-loaded to peers when the server becomes overloaded. However, they do not mention whether the server-side is to be dimensioned to handle any or all hot-spots, which is why we didn't classify it as a 'fat server' model, nor as a decentralized one that handles hot-spots;

The other major feature yet to be delivered consistently by P2P MMOGs (especially multiple arbiter and secure models) is transparent interaction across cell borders. It is especially difficult to synchronize distinct groups of nodes, each group replicating one cell, in a many-to-many fashion without IP multicast and considering the limited upload bandwidth of home users. Some sort of synchronization is required if the borders between neighbouring cells are to disappear to players. None of the chosen models achieves this ideal. However, the *player mesh* models reviewed in Chapter 2 are able to achieve this with ease due to their focus of connecting players in AoI reach directly without arbitrary cell borders getting in the way. However, that is only one of the few advantages of these models. Below we sum up the support for inter-cell interaction offered among the chosen works:

- FreeMMG 2 is the only one among the chosen works to explicitly address direct interaction between *neighbour objects* (objects owned by different cells), which includes neighbour avatars. However, the latency of inter-cell interaction between objects is greater than that of intra-cell interaction, resulting that probably the difference is to be perceptible by players, so the transparency ideal is not fully met. FreeMMG 2 is also able to transfer ownership of objects between cells in a lazy fashion which also avoids objects blinking out of existence during transfers, which was a problem in the original FreeMMG model. Also, the original FreeMMG model transferred objects through the server, while in FreeMMG 2 the server participation in object transfer is only required whenever *global consistency* is an issue. For example, when the transferred object cannot be lost or duplicated during or after the transfer (CECIN et al., 2004);

- ACORN and Kabus and Buchmann both achieve decentralized object transfers between cells, but they do not explicitly address the issue of neighbour objects interacting. ACORN lets players carry their own character state between regions as in SimMud (KNUTSSON et al., 2004), which in the case of ACORN we assume is a procedure protected against state cheating due to all ACORN region simulation being replicated at trusted servers. So, if players in ACORN lie about their character state during region migration they are risking being detected as cheaters and banned;

- DaCAP and Endo et al. transfer objects between cells with the help of servers, like in FreeMMG. DaCAP's cheat prevention relies heavily on servers checking the gains of player avatars whenever a region border is crossed. If the cell is running in *P2P mode*, all current peer members report the results (gains) for the migrating avatar and the server checks for a consensus. Both DaCAP and Endo et al. do not explicitly address the issue of neighbour objects interacting;

- Hampel et al., Izaiku et al. and Peer Clustering do not mention inter-cell synchronization at all.

All compared works address the issue of providing long-term state persistence. In contrast, the MOPAR (YU; VUONG, 2005) architecture is one that doesn't persist cell state. ACORN simulates a full copy of the MMOG simulation at the server, and that server-simulated state is used as the ultimate source of state whenever the state maintained on the P2P overlay fails. That is one of the main benefits of ACORN and any proposal where servers receive all player events and thus are able to keep an up-to-date version of the virtual world being served by the P2P overlay. DaCAP achieves server-side persistence by storing all relevant world data in a server-side database. The other more decentralized models achieve persistence mainly by mass replication, among client machines, of the MMOG state.

Finally, most compared works authenticate players using a centralized source of trust, which is what most hybrid MMOG architectures do. The only exception in Table 6.2 is Hampel et al. which neither addresses the issue of player authentication nor mentions any kind of server infrastructure. However, their work could be trivially extended to include an authentication server so this is not an issue.

So, as far as the core features are of concern, ACORN, DaCAP and FreeMMG 2 arguably share the lead. DaCAP may however be too reliant in server-side infrastructure. ACORN seems to handle hot-spots effectively and in a decentralized way. FreeMMG 2 explicitly addresses direct object interaction across cell borders, though with increased interaction latency. FreeMMG 2 also helps the application in maintaining a globally-consistent game state even when individual cells fail and roll back to old server-side data. The latter comes at the cost of involving server-side resources in the specific inter-cell interactions and object transfers that can potentially break the global consistency when they fail.

## 6.3 Decentralization and communication cost comparison

One of the main motivations behind peer-to-peer MMOG architectures is reducing or eliminating the costly server-side infrastructure. In this section we evaluate the chosen works according to the amount of CPU and communication load that they leave at the server-side and also the load that the proposed architectures impose upon 'peer' nodes (client-side nodes) that are to replace the bulk of MMOG servers. Table 6.3 summarizes that comparison.

The first criteria for comparison is whether the client-side hosts that replace servers can be provided solely by active player nodes or whether extra nodes like dedicated volunteers or super-peers are required. The ACORN paper mentions that either is a possibility and their 1,000-player region simulation indicate that some super-peers might have to be moved to act as coordinators in hot-spot areas. Endo et al. handle some regions at the server but that is not a hard requirement – it seems to be a just way to use up all available server resources first and thus to help minimize the site (cell) simulation load left for client hosts. Izaiku et al.'s region controller, whose function is constantly passed around all monitors (replicas) of a region, needs to be able to dispatch $N_P$ updates in a single network step, where $N_P$ is the number of players in the region. It is reasonable to assume that, currently, at least some home users' nodes may not be able to cope with this upload bandwidth requirement for some values of $N_P$. FreeMMG requires additional *witness nodes* on each region which might increase the amount of peer resources needed to serve in the overlay beyond the resources that can be obtained from active players. Finally, FreeMMG 2 is the one that needs the most amount of peer resources. In FreeMMG

Table 6.3: Decentralization and communication cost comparison

| Criteria | ACORN | DaCAP | Hampel et al. | Endo et al. | Kabus and Buchmann | Izaiku et al. | Peer Cluster | Free-MMG | Free-MMG 2 |
|---|---|---|---|---|---|---|---|---|---|
| Playing peers only | ✓- | ✓ | ✓ | ✓- | ✓ | ✓- | ✓ | ✓- | |
| RCs per cell | 1 | $[1, N_P]$ | $N_C$ | $N_C$ | $N_C$ | $N_C$ | $N_C$ | $N_P + W$ | $2N_C$ |
| Updates sent per RC | $N_P$ | $N_P$ | $\frac{N_P}{N_C}$ | $\approx \frac{N_P}{2}$ | $N_P$ | $\frac{N_P}{N_C}$ | $[0, N_P]$ | 0 | $\frac{N_P+N_N}{N_C}$ |
| Updates received per player | 1 | $[1, N_P]$ | 1 | $\frac{N_C+1}{2}$ | $N_C$ | 1 | 1 | 0 | 1 |
| Commands sent per player | $[2, N_P + 1]$ | $[1, N_P]$ | $N_C$ | $[1, N_C]$ | $N_C$ | $N_C$ | 1 | $N_P + W$ | 1 |
| Low server CPU | | ✓- - | ✓ | ✓ | ✓ | ✓ | ✓- | ✓ | ✓ |
| Low server upstream | ✓ | ✓- - | ✓ | ✓- | ✓ | ✓ | ✓- | ✓- | ✓ |
| Low server downstream | ✓- | ✓- | ✓ | ✓- | ✓ | ✓ | ✓- | ✓- | ✓ |

2 it is explicitly assumed that non-playing volunteer nodes will be available to help in composing the peer-to-peer cell simulation overlay for active players to play into.

The above comparison criteria is complemented by the number of region controllers (RCs) required to compose a cell. Note that the number of RCs does not correspond directly to the network or CPU load of cell nodes. Rather, it only indicates the amount of distinct simulation processes (LPs) needed to compose a cell. Most works require $N_C$ RCs, which just means a small amount of RCs which can cope with all-to-all (full mesh) synchronization schemes. ACORN requires 1 coordinator (RC) per region. That coordinator responsibility is constantly passed around peers to minimize the window of time where a coordinator may cheat before being detected. Additionally, trusted servers (called 'CAP' nodes) are able to roll back any cheats since they execute a replica of the simulation performed by the coordinator of each region. A DaCAP cell is either running in *C/S mode* with 1 server-side coordinator or in *P2P mode* where each player in the cell is also an arbiter. FreeMMG runs all cells in peer-to-peer mode similar to DaCAP with an additional amount of randomly chosen *witness nodes* present in each cell. Finally, FreeMMG 2 requires $2N_C$ cell nodes: $N_C$ primary cell nodes and one backup cell node for each primary cell node, resulting in a relative $2N_C$ cell node count when compared to the other multiple arbiter models.

From the third to the fifth line of Table 6.3 we assess the bandwidth that is required of both RC and playing peers.

The third line show that ACORN, DaCAP and Kabus and Buchmann require RCs to send out $N_P$ updates every network step, while Endo et al. amortizes this to about a half. In the case of DaCAP, this is because any increased $N_P$ amount is to be handled by well-provisioned servers. In the case of ACORN, it is either a well-provisioned super-peer or the game protocol imposes a reduced packet rate between players and the region coordinator for some reason. However, Kabus and Buchmann and Endo et al. just do not exploit all the bandwidth reduction that can be obtained by cell state replication. Hampel et al., Izaiku et al. and Peer Clustering have each cell send out only one update per player resulting in an average of $\frac{N_P}{N_C}$ update packets being sent per each RC on each network step. In the original FreeMMG, no update (large) packets are exchanged between peers, which allows for RTS game support where each player controls large amounts of avatars. Finally, FreeMMG 2 adds the overhead of inter-cell synchronization where each neighbour cell

is counted as an additional player for simplicity. The actual cost of updating a neighbour cell may be either larger, when too many avatars are near the cell border, or smaller, if two neighbour cells are not currently interacting.

The fourth line shows the amount of updates received per player on each network tick. Kabus and Buchmann and Endo et al. require each player node to receive $N_C$ and $\frac{N_C+1}{2}$ updates per network tick respectively which is rather wasteful if compared to the single update that client-server architectures and most others send to its players. DaCAP's updates that players send to each other in *P2P mode* are probably small single avatar position updates and similar messages, and $N_P$ is limited to a small amount so this network load is not a problem. Finally, the original FreeMMG had no constant exchanges of state updates anywhere, and FreeMMG 2 requires players to receive only one update at any given simulation tick time, which is as bandwidth-efficient for a player node as it can get.

The fifth line shows the amount of command messages (in practice, UDP unicast packets) that a player has to send at every client-side network tick[1]. From all the bandwidth-related criteria, this is the one where a high count will have the least impact. That is, requiring each player to send a few tens of unicast command packets per network tick is feasible since they are always small packets. However, it would be desirable to leave clients sending a single stream of unicast UDP packets if possible since all overhead imposed by the base architecture will add up with other overhead from other libraries and the application. Below we sum up the design of the chosen works for outgoing player commands:

- ACORN has players sending at least two updates, one to the region coordinator and one to the CAP (trusted server). In addition, player nodes in an ACORN region that share an AoI exchange messages directly like in 'player mesh' models, causing an additional $[0, N_P - 1]$ command streams to be uploaded by each player. We are not sure what are the semantics of these messages but we assume they are authoritative player updates that get corrected (or not) by the coordinator;

- DaCAP and Endo et al. has player nodes sending either one stream of command to a server-side RC or has player nodes sending multiple command streams when the region is in *P2P mode*. In the latter case, DaCAP has players sending messages to each other ($N_P$ streams) and Endo et al. has players sending messages to all *site servers* ($N_C$ streams);

- Endo et al., Kabus and Buchmann and Izaiku et al. have player nodes always sending out commands to all RCs ($N_C$ streams);

- FreeMMG has players and witness nodes sending commands to each other constantly ($N_P + W$ streams);

- A FreeMMG 2 cell will generate, at most, a single stream of update packets for each player node that is interested in its state. In FreeMMG 2 we assume and recommend that applications bind a player node to only one cell at any given time, which is always the cell that currently owns the player's avatar object.

[1] As discussed in Chapter 2, the rate of outgoing command packets from players is usually greater than the rate of incoming updates. But for comparison purposes we can assume that they both lie somewhere around 10 or 20 packets per second which is enough for most networked games.

Finally, the bottom of Table 6.3 compares the server-side load in terms of CPU and bandwidth of each architecture:

- ACORN's servers are CPU-intensive since they have to simulate the whole virtual world. Also, they receive command packets from all players. However, the player packets can probably be aggregated and compressed to reduce significantly the bandwidth usage of both player nodes and the CAP (server-side simulator) nodes, so the server-side bandwidth usage issue of ACORN seems manageable;

- DaCAP handles hot-spots, object transfers and cheat-detection server-side, but the proliferation of hot-spots can be prevented with game design and with the cell subdivision scheme that is mentioned in the paper. Thus, its server-side CPU usage will be lower than ACORN's. The server uses bandwidth to serve hot-spots and to perform inter-cell object transfers. Even if the proliferation of hot-spots is controlled, the server bandwidth has to be dimensioned to serve peak loads. The DaCAP server has to send player update packets, which are large, probably resulting in significant server upload bandwidth usage as the number of total players in the system increases to massive scale;

- Peer Clustering, like DaCAP, handles hot-spots server-side, if any. These include both temporary hot-spots and 'hub' locations like virtual cities. Peer Clustering however does not run cheat detection server-side and it does not specify any inter-cell interaction, thus resulting in lower server-side resource impact than DaCAP for now;

- Endo et al. and FreeMMG use server-side bandwidth to manage object transfers between regions in addition to other light-weight tasks that are performed by other architectures. The actual communication load or cost of this function depends on many factors, including game design;

- Hampel et al., Kabus and Buchmann, Izaiku et al. and FreeMMG 2 do not force any of the server-side involvement mentioned above and thus achieve the greatest level of decentralization among the chosen works. The main trade-off seems to be their poor handling of hot-spots (see Table 6.2).

In this cost comparison there is no clear winner. In developing FreeMMG 2, our particular criteria was to prioritize both vital security attributes (which is compared in the next section) and decentralization. To this end, we have come up with a solution that requires the larges amount of LPs allocated to each cell, and we have left the hot-spot problem unsolved. Still, some works perform better than FreeMMG 2 in reducing the amount of messages: Hampel et al., Izaiku et al. and Peer Clustering. However, considering the earlier feature comparison and the following security comparison, we believe the extra cost paid by FreeMMG 2 is worth the extra functionality if compared to these works.

## 6.4 Security comparison

Finally, the most important comparison in our view is the security-related one. Our top priority is to protect the state of a MMOG being illegally altered at any cost. We have chosen the prevention route since, in our view, automated detection and correction of state

cheats and other non-preventive solutions to state cheating are inconvenient in many ways. There are several alternatives to achieve this, and ACORN's approach of a 'silent' server-side execution that is queried every once in a while to correct peer-held state is one such approach – we also tried that approach with the VFreeMMG model (CECIN; BARBOSA; GEYER, 2005). The other known approach is the *multiple arbiter* one, where peers form replication groups and a high quorum of colluding malicious nodes in each state partition is required to pass illegal state modifications 'under the radar' of the trusted servers that manage the peer network.

Table 6.4 shows the security comparison. ACORN is the most safe model across all chosen works, and among the practical works we encountered during our literature review it is the safest among all. ACORN is better than FreeMMG 2 because, since it simulates the whole virtual world server-side, no state loss is possible in practice. In FreeMMG 2, state loss is always a possibility in the long run if attackers are granted maximum attack power. The fact that back-up cell nodes have hidden IP addresses however is a significant deterrent to network-level attacks being used to cause loss of cell state. Thus, we assert that except for the most dedicated groups of attackers, losing peer-held cell state is possible but unlikely in FreeMMG 2. But the application must also set adequate replication parameters to FreeMMG 2 so that cell state is not discarded when only a few nodes are marked as 'failed' during cell fault recovery.

In any case, in the event that any cell's state is lost, a FreeMMG 2 game application can use the 'vital cell state' facilities of FreeMMG 2 to partially restore it. This feature can and should be used by the application to guarantee a globally-consistency game state. In other words, this allows the game to always maintain invariants over the global game object collection (the global set of simulation state variables), even in the unlikely scenarios of cells losing their state. We believe that, in any hybrid model, servers should be used to selectively persist the most valuable in-game data, such as the amount of virtual currency of each player, since these are hardly synchronization tasks that can deplete server resources.

The Peer Clustering model employs the same state 'vital state' storage at the server-side of FreeMMG 2. Kabus and Buchmann take periodic (and probably complete) snapshots of cell state to servers, which partially prevents state loss due to network-level attacks or otherwise, but that is less efficient than prioritizing state items. Additionally, they also do not explicitly address the issue of maintaining global consistency. FreeMMG also prevented state loss by taking full cell snapshots, but it did also not address the problem of global consistency nor it helped the application to do so. In FreeMMG we only identified the problem.

Table 6.4 shows that most chosen works address state cheating if network-level attacks (NLAs) are excluded from the threat model. The only exception is the Peer Clustering model which, though it maintains multiple replicas of a cell, one of them is authoritative over the others and it concentrates the receiving of player command packets. Thus, it features a single point of arbitration which is prone to cheating as discussed earlier. Also, without network-level attacks, regular peer churn or single peers failing on purpose (griefers) does not cause state loss nor it facilitates state cheats to occur, considering reasonable peer churn rates.

If network-level attacks are included in the thread model, only ACORN and FreeMMG 2 are able to prevent state cheating from happening. Though it is possible in theory to perform state cheating in FreeMMG 2, it is mathematically unlikely to happen for any slightly realistic scenario if the replication parameters are chosen adequately by the ap-

Table 6.4: Security threat coverage comparison

| Threat | ACORN | DaCAP | Hampel et al. | Endo et al. | Kabus and Buchmann | Izaiku et al. | Peer Cluster | Free-MMG | Free-MMG 2 |
|---|---|---|---|---|---|---|---|---|---|
| State loss (NLAs considered) | ✓ | | | | ✓- - | | ✓- | ✓- - | ✓- |
| State cheats (NLAs considered) | ✓ | | | | | | | ✓- - | ✓ |
| State cheats (NLAs excluded) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| Information exposure cheats | ✓- | ✓- - | | | ✓- - | | ✓- - | | |
| Griefers (single nodes misbehaving intentionally) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| Tolerance to regular peer churn | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

plication. The only possibility is if a large enough amount of malicious colluding nodes are drawn by chance from the volunteer pool and placed on the same cell. Again, this is unrealistic as discussed in Chapter 5. Our trade-off is in, perhaps, the large amount of replicas needed in each cell, which may be pushed further up due to a possibly high quorum of replicas required to stand up as 'non-failed' after a cell failure is detected.

Finally, as far as protection against other forms of cheating, no chosen work addresses time cheats or most protocol-level cheats, and most of them are vulnerable to information exposure cheats due to the use of peer-held cell state and cell replication strategies. ACORN is surprisingly resistant to information exposure cheats due to its single and, most importantly, moving coordinator, and they also assume secure DHT routing which we believe includes (optionally) cyphered dissemination of updates over the (potentially) multi-hop DHT path between the coordinator and the update recipients. DaCAP and Peer Clustering are immune to information exposure only on hot-spots whose data is kept by trusted servers. Kabus and Buchmann partially address information exposure by cyphering data at peers that are supposed to help in disseminating or serving it but are not supposed to either alter that data or peek at its contents.

Thus, based on the assumptions that protecting MMOG state against illegal modification is a non-negotiable requirement and that attackers may be sufficiently motivated to orchestrate network-level attacks against peer-to-peer MMOGs, the only peer-to-peer MMOG models we know that both achieve some degree of decentralization and that are also sufficiently secure are ACORN and FreeMMG 2. Both also address the issue of state loss, with ACORN's solution guaranteeing that no state loss is actually possible, while in FreeMMG 2 that absolute guarantee only covers *vital state*, whose scope is to be determined by the application.

## 6.5 Closing remarks

The following assumptions, which have been laid out throughout this thesis, dismissed the need to compare FreeMMG 2 with the *player mesh*, *fat server-side* and *single arbiter* peer-to-peer MMOG models:

- Significantly reducing the cost of service providers for hosting a MMOG service is desirable;

- Supporting virtual economies or at least some form of long-term challenge is important for any game design that claims to be a MMOG design. A player count is not sufficient, and there are other basic requirements such as the game being real-time and presenting a graphical interface;
- Protecting MMOG state against illegal modification is an essential requirement for any peer-to-peer MMOG architecture;
- Network-level attacks are a threat that cannot be ignored, since they can cause the architecture to lose or drop game state, or they can be combined into more sophisticated attacks to perform state cheating. Attackers may be motivated to do so due to the real value of MMOG state;
- Information exposure cheats, time cheats, automation cheats and any remaining protocol-level cheats can be either ignored or addressed by the application or add-on detection modules.

This chapter performed a detailed comparison of a few works which stand up well to those assumptions, especially those which we identified as belonging to the *multiple arbiter* approach. From this comparison, we have shown that ACORN and FreeMMG 2 are the only models that meet our security requirements. Between these, ACORN is significantly more server-intensive, which also clashes somewhat with our basic motivations.

Hopefully, we have also shown that FreeMMG 2 provides an interesting feature set. We consider inter-cell interaction support to have priority over proper hot-spot support, and among the relatively secure works FreeMMG 2 is the first to explicitly address this issue, as far as we can tell. This may stem from our idea of what the first deployed peer-to-peer MMOG would look like. The idea of handling hot cells with servers, which we have found in DaCAP and Peer Clustering models, is a very interesting and elegant solution. FreeMMG 2 could be extended to include such support. For now, FreeMMG 2 cells just handle hot cells by sending fewer updates to players, until most that players see on-screen is either frozen objects or seconds-long dead reckoning sequences. This may not be an issue if the hot area is a city where player's cannot shoot at each other (CRIPPA; CECIN; GEYER, 2007), but it is certainly an issue if massive player battles are to be supported.

Lastly, we should note that our comparison has disregarded visual consistency and player command responsivity (interaction latency) issues. In FreeMMG 2 we tried to address this indirectly through trying to minimize the amount of hops between players. However, peer-to-peer cell-based models in general, and multiple arbiter models in particular, cannot really compete with server-centric designs in terms of latency. Due to the difficulties imposed by our focus on security and decentralization, our approach to latency and game genre support was left to be simply a matter of considering FreeMMG 2 capable of serving MMORPGs because RPGs tolerate latency and because other architectures with similar multi-hop designs also claim to offer that support.

# 7 CONCLUSION

Massively multiplayer online games such as World of Warcraft, PlanetSide, EverQuest, Lineage, and many others are Internet-enabled, massively-populated virtual worlds where players can interact with a virtual environment and with each other's avatars through a graphical and real-time interface. To human players, the state of a virtual world is perceived as being like the state of the real world: persistent and evolving over an arbitrary span of time depending on some transformation rules. A MMOG virtual world necessarily supports competition and thus protecting the system against various attacks from cheaters, griefers and vandals is vital.

MMOGs are an example of *virtual worlds*, which is a broader concept. Other non-MMOG examples of virtual worlds include the text-based MUDs or the current turn-based, web-based MMOGs (not graphical or not real-time) and virtual worlds created for collaboration or training (non-competitive). However, some virtual worlds such as Second Life defy that categorization by including support for both competitive play, purely social interaction and 'e-commerce'. The definition of a MMOG in this work was provided as a means to limit its scope, more than anything. Others may label Second Life, MUDs or turn-based persistent worlds as MMOGs.

As of 2009, virtually all MMOGs are client-server systems, where the world is simulated at a trusted server infrastructure and then served to client (player) machines which either just display the server-provided results as-is or derive a smoother graphical rendition from them. This results in a recurring server-side computing, communication and administrative cost which can only be afforded by a few. This contrasts with other Internet applications such as the World-Wide Web (WWW) which have thrived on the client-server paradigm in such a way that almost anyone is able to host a massively-popular website, as far as the monetary cost of hiring space on website servers is of concern.

FreeMMG 2 is an alternative distribution architecture for MMOGs. Like the many MMOG architectures evaluated in Chapter 2, FreeMMG 2 follows the *peer-to-peer* design paradigm. When applied to distributed computing architectures, this peer-to-peer paradigm results in distributed computing systems which are more decentralized than client-server systems. FreeMMG 2 is hybrid: though most costly tasks are to be delegated to a network of players and other kinds of volunteer nodes, the servers retain ultimate control of the game. Our goal was not to remove control over the application from a central operator, which may be among the leading goals of other peer-to-peer systems such as FreeNet (CLARKE et al., 2001). Instead, our goal was to lower the costs of being a game operator, thus lowering the barrier of entry in such a way that hosting a MMOG would be an endeavor as accessible as it is to host a website.

Since our work on the original FreeMMG (CECIN et al., 2004) we have chosen to view a running MMOG as a large distributed system in which thousands of concurrent

users commit modifications in real-time to a valuable shared piece of state. And due to the competitive nature of games it can be expected that many users will want to modify or bypass the state transformation rules, and act to do so if the task seems feasible. Thus, if what the players are fighting each other over for has value, then the main technical challenge of a peer-to-peer MMOG seems to lie in maintaining the persistence and integrity of the world's state in the long run. This state integrity cannot be taken for granted by a peer-to-peer MMOG, or any other decentralized stateful system for that matter.

The originality of FreeMMG 2 lies in it being a peer-to-peer MMOG architecture for avatar-based games (MMORPGs) which offers both significant server-side cost reduction and strong integrity and persistence of game state even in the presence of malicious nodes and network-level attacks such as DoS and DDoS attacks. Even if DoS and DDoS attacks are disregarded as threats, FreeMMG 2 also presents some unique features such as a light-weight inter-cell synchronization mechanism which supports both object transfers and interaction across cell borders. In FreeMMG 2 we have also dealt with the problem of maintaining the global game consistent across cell borders, such as ensuring that an unique object does not disappear or is duplicated, which is a problem rarely considered in the literature.

FreeMMG 2 is a cell-based, replication-based, multiple arbiter, hybrid of client-server and peer-to-peer model that supports MMOGs. As discussed in Chapter 5, existing results in the literature already show that multiple arbiter approaches can be significantly resistant to collusion attacks intended as facilitators to state cheating. Through network simulation, we have shown that FreeMMG 2 is also bandwidth-efficient. Thus, we consider our hypothesis verified: the multiple arbiter approach for peer-to-peer MMOG support not only is naturally state cheating resistant, but can be made to work with realistic broadband connectivity such as peers on ADSL connections, and in a scenario where no global IP multicast is available.

## 7.1 Summary of contributions

We have identified the following contributions in this work:

- Distributed simulation and peer-to-peer games: we provide a background discussion that identifies a synergy between distributed simulation and peer-to-peer games. That is because in both problems there is the need for multiple distributed processes to synchronize in applying modifications (events) to a shared state.

- Client-server game protocol review: we provide a didactic overview of a broad range of techniques used to build client-server game protocols and to make them mask or compensate for network inconveniences such as delay, jitter and packet loss, and thus be able to support action games such as FPS games;

- An assessment of security issues for peer-to-peer MMOGs: we provide a simplified and focused assessment of the security threats to MMOGs. We focused on the threats that could possibly cause the most damage to a MMOG which, as argued, are cheats and attacks that can cause the shared state of a MMOG to be illegally altered or lost;

- A review of peer-to-peer MMOG architectures: we have reviewed several recent peer-to-peer MMOG architectures, focusing on security and level of processing

and communication decentralization. In Chapter 2, we fit most of them in one of four categories (player mesh, fat server-side, single arbiter and multiple arbiter). In Chapter 6, we perform a more detailed analysis of a few selected works, comparing them to FreeMMG 2;

- A new cell event and state synchronization protocol suitable for a *multiple arbiter* MMOG architecture: we have developed the Baseline State Synchronization (BSS) protocol, which is a simple modification of the Trailing State Synchronization (TSS) protocol (CRONIN et al., 2004a). BSS combines the security and verifiability of state replication based on conservative event ordering with the responsivity of optimistic event execution: all replicas agree on the *conservative* state snapshot, but may disagree on the *optimistic* state snapshot;

- A new hybrid peer-to-peer and client-server MMOG architecture: Chapter 4 describes in detail the FreeMMG 2 architecture. FreeMMG 2's originality is in being highly resistant to important security threats and in achieving significant decentralization of services. In its current form, FreeMMG 2 already provides security guarantees comparable to the ones provided by default by a client-server architecture, while allowing smaller game companies or researchers to operate MMOGs;

- Verification that 'multiple arbiter' replication approaches can be bandwidth-efficient: Chapter 5 shows, through network simulation, that a feature-filled, peer-to-peer MMOG model based on 'multiple arbiter' (or consensus) cell replication can indeed work on realistic bandwidth constraints and without relying on IP multicast. Together with the natural resistance of replication approaches to collusion-based cheating attempts, multiple arbiter approaches are shown to be promising, warranting more research into other aspects of these models such as their interaction latency.

## 7.2 Future work

FreeMMG 2 can be improved in several directions. Below we discuss some of these directions. These mix both add-on and redesign work opportunities.

### 7.2.1 Dealing with hot-spots

Hot-spots are areas (cells) of the virtual world which, at any given time, host an unusually large amount of player avatars. In FreeMMG 2, a clear case of hot-spot would be a cell hosting 100 or 200 avatars. The only way that this kind of concentration would not be a problem is if the hosting cell is comprised of 'super-peers' with server-grade bandwidth available to them. The neighbour cells would also have to cope with this kind of network load if the 'hot' cell has many avatars near one or more of the cell borders, since neighbor cells would receive large update packets from the hot cell due to the large amount of object status updates on each packet incoming from the hot cell.

FreeMMG 2 does tolerate hot-spots. By 'tolerate' we mean that FreeMMG 2's simulator nodes and the overlay network don't just collapse. As the avatar count increases, the available bandwidth on the network stays the same, so the overall frequency of updates, be it between cells or from cells to player machines, decreases proportionally. This decrease of update frequencies increases the latency perceived by players and decreases the simulation consistency as very due updates make remote avatars 'jump' from one predicted

position to the correct position far away from the prediction, for instance.

FreeMMG 2 uses a static mesh of simulator nodes, and the inability to handle hot-spots stems directly from that. More dynamic methods such as Asynchronous Synchronization (AS) (BAUGHMAN; LEVINE, 2001) are more appropriate for tolerating hot-spots. The AS is a good example because it also employs lock-step (conservative) simulation, just as FreeMMG 2. However, AS does this only between avatars that intersect their spheres of influence (SoI), that is, only when the avatars get close enough to each other. In FreeMMG 2, hot-spots are much easier to cause due to the coarse-grained 'cell' data distribution. A 'cell' can be visualized as a 'map' of an FPS game such as Quake (ID SOFTWARE, 2008) or Counter-strike (WIKIPEDIA, 2008a). So, one could easily have 100 avatars that are far away from each other but they happen to be at the same 'cell' so they are forced to engage in lock-step synchronization.

In FreeMMG 2 we have chosen the static cell division mainly because it is simpler. For one thing, it treats dynamic avatars, NPCs and environment equally. All are modeled as game-dependent simulation 'state', and the FreeMMG 2 middleware needs very little insight on the actual representation of game objects. At most, it needs timestamps and 'state' of cells or individual objects as serialized blocks of bytes. We think that this particular benefit that arises from the simplicity of static 'cell' division already offsets the inconveniences caused by a lack of adequate support for hot-spots.

One way to mitigate the hot-spot problem in FreeMMG 2 would be to detect when a cell has a certain large count of avatars and then start moving the more potent ('super-peer') volunteer nodes to that cell. These nodes, having more bandwidth, would be able to serve more player machines, and would be able to send and receive larger 'tick' messages on the peer-to-peer overlay network of the cell. However, this would modify the scenario for collusion-based cheats: a cheater that could afford to donate several high-capacity nodes to the network could draw those nodes to a cell by somehow gathering a large mob of players into a cell. We have not simulated this super-peer optimization for hot-spots, nor the possibilities of collusion cheating that can occur due to that optimization. This could be done as future work.

Another strategy for coping with hot-spots would be increasing the cell node count of a particular cell when it starts hosting many avatars. Having more cell nodes, more bandwidth to serve player machines is available at that cell. This could be coupled with the super-peer allocation strategy mentioned above for maximum effect. However, this strategy can back-fire as this increases the size of the full-mesh peer-to-peer network formed between all cell nodes, which does not scale well. The cell node count is a sensitive parameter, and in FreeMMG 2 we assume that a fixed number of nodes for all cells should be chosen prior to deploying the game. But maybe there will be some gain in allowing it to vary during each cell's lifetime. However, the model will need to be revised to see if this affects inter-cell synchronization, for instance. Exploring the effect on varying cell node count for hot-spots is also left as future work.

### 7.2.2 Informed choice of nodes from the volunteer pool

In FreeMMG 2, the virtual world cells have to be served by groups of volunteer nodes, which are drawn from a pool of volunteers (see Figure 4.1 in Chapter 4). Currently, volunteer selection is completely random: all nodes in the pool have the same chance of being selected when a new cell node is needed by the network. It is possible that more informed choices would yield better results, such as decreasing the chance of collusion cheating and reducing the occurrence of vandalism (nodes that fail on purpose).

For instance, nodes that have just been removed from a cell that failed could have a lowered chance from being selected in the future, or they could be outright banned for several hours or days from being selected. However, care must be taken to avoid the entire pool from being 'banned'. For instance, a large group of colluding nodes could purposefully sacrifice a few of their own cheater nodes to ban a large number of honest nodes. Simulations could be performed to determine safe levels of punishment to apply to cells that have failed, as well as the rate in which the punishments expire.

Several other pieces of information could be used for informed choices. For instance, the IP addresses of the volunteer nodes could also be checked. Peers hosted on the same sub-network could be barred from participating on the same cell, and IP addresses could be matched against IPs or IP address masks present in public blacklists (e-mail spamming, etc). Blacklisting volunteers would not damage the network as it damages an e-mail network, since false positives only result in a shrinked pool of volunteer nodes to choose from. However, if a particular game deployment starts running out of volunteers, it might be necessary to use blacklisted nodes or risk putting cells off-line (off the game).

In the end, the several pieces of information that can be inferred from a volunteer could be used to compute a probability of that node being selected: history of being present in cells that failed, IP address history, and even game-play behavior if that volunteer is tied to a particular player account. Tuning those heuristics so that measurable gains in cheat-proofing, vandalism (griefing) avoidance and overall network stability are obtained is a good opportunity for future work.

### 7.2.3 Improving the inter-cell synchronization protocol

Cells know about the state of objects owned by neighbour cells due to incoming update packets. However, at each tick, two neighbour cells only exchange one such packet. It might be the case that a significant gain is achieved if two or more packets are exchanged per tick between neighbour cells. That would help minimize the impact of packet loss, delay and packets being dropped on purpose by the sender or the receiver.

If additional packets are sent by different primary cell nodes on the same tick to the same neighbour cell, then there is the issue that these nodes can and probably do have different views of the world on their optimistic cell snapshots. So, the receiving cell will, on average, receive two or more distinct updates corresponding to the same time tick and for the same neighbour cell. Obviously, one of them will be selected and the others discarded. The trivial solution is to consider the first one that is committed to conservative ordering at the receiving cell and discard the others. If many such packets are committed to the same conservative step at the receiving cell, then one of them is selected based on some tie-breaking criteria. This resulting scheme does minimize packet loss, delay and intentional drops, but we think that the single-packet inter-cell synchronization design must be tested and ruled out first.

This simple increase in outgoing packets does not help to decrease interaction latency between 'neighbour objects' – objects owned by two different cells which are neighbours to each other. To significantly decrease interaction latency between, the number of hops would have to be cut. The inter-cell packets currently travel one hop between cells and one hop inside the receiving cell which is when the foreign update packet is disseminated as cell input for deterministic execution. These two hops in particular could be condensed into one of the originating cell sent the update directly to all primary cell nodes of the destination cell. All these would have to be sent by the same sender node at the originating cell, since the multiple update packets would have to be identical. However, this

might not be feasible without assuming IP multicast support or without raising the minimum requirements for cell node upload bandwidth. In the current scheme, the inter-cell synchronization overhead weights to each primary cell node roughly the same as serving one additional player. Having each primary cell node broadcast an inter-cell packet each turn would raise that overhead to serving $N$ additional players, where $N$ is the number of neighbour cells. Considering that the average peer currently has ADSL connectivity with low upload bandwidth, that additional cost may forbid peers from serving updates to any players at all after performing the inter-cell synchronization task.

There may be several other ways to improve the quality of inter-cell interaction. For example, the FreeMMG 2 player nodes only upload one stream of short command packets to the cell that currently owns its avatar, which is the same that client nodes of client-server MOGs do. They have idle upload bandwidth which could be exploited in some way to accelerate interaction. For instance, players could exchange updates directly between each other, or they could send commands to multiple cells at once, or receive updates from multiple cells at once. We didn't develop that for two main reasons. The first one is that the focus of this thesis is in securing the state of a MMOG and such schemes open even more security issues which would then have to be addressed or at the very least discussed. The second, and main reason, is that we wanted player nodes to remain low-profile for multiple sub-reasons. We wanted player nodes to, perhaps, simultaneously play and serve as cell nodes. We also wanted to allow any player to play the game regardless of its available bandwidth. And finally, we wanted the application or the users themselves to have extra player bandwidth to implement other features such as peer-to-peer voice communication (TRIEBEL et al., 2009). Nevertheless, that extra player bandwidth is there to be tapped into, perhaps by an improvement of inter-cell interactions.

### 7.2.4 Detection strategies for protocol-level cheating and griefing

In FreeMMG 2, the focus was on protecting the state of each and every cell among thousands of cells. Unfortunately, FreeMMG 2 does not directly address most protocol-level cheats with the exception of inconsistency cheats (WEBB; SOH, 2007). We left protocol-level cheats such as time cheats to be addressed by the application or an additional middleware layer that would provide higher-level services. Alternatively, future work could extend FreeMMG 2 to deal with cheats that we have not addressed.

Regardless of software engineering concerns, a detection approach for the remaining cheats is probably better suited to FreeMMG 2 than prevention. There are works that fit as add-ons and that detect protocol-level cheats quickly and with low false positive rates, such as the AC/DC algorithm (FERRETTI; ROCCETTI, 2006). Other authors of *multiple arbiter* MMOG models also suggest that protocol-level cheats can be dealt with adequately by add-on modules (KABUS; BUCHMANN, 2007).

Some works (CHAN; HU; JIANG, 2008; KABUS; BUCHMANN, 2007; WEBB et al., 2007) that detect protocol-level cheats rely on the cryptographic property of *non-repudiation*. Non-repudiation is the ability to prove beyond doubt the origin of a message. This translates to being able to prove that a peer has misbehaved with an effective zero false-positive rate. An example of this is the inconsistency cheat, where a peer sends two different packets that are supposed to be identical. If the packets are required to be digitally signed by the sender then there is no way that the sender can claim that it did not send two distinct packets. Thus, a peer that attempts an *inconsistency cheat* can be detected and banned by live or post-mortem auditing of protocol-level (packet) activity.

Finally, some cheat detection schemes cause the level of abstraction (the generality) of

FreeMMG 2 to decrease. By offering protection to some protocol-level cheat, for instance time-based cheats, we are assuming more about the meaning of the event data exchanged between players and cell nodes. For instance, we might assume that players are able to timestamp outgoing events. We leave these and other optimizations of FreeMMG 2 to specific games, game genres, or interaction models to future work.

### 7.2.5 Implementing a MMOFPS over FreeMMG 2

FreeMMG 2 was originally an attempt at peer-to-peer support for MMOFPS games. Supporting a peer-to-peer MMOFPS that is more than a shooter over a large area – that is, one that also has a virtual economy and perhaps mutable terrain – turned out to be a daunting task when considering other limitations such as an Internet without widely-deployed IP multicast and peers with ADSL upload bandwidth. Also, supporting low-latency MMOFPS games with players from all over the globe may be difficult to realize simply due to the sheer geographic distance even if network delay were to be reduced to the minimum required by the light speed barrier (BOSSER, 2004). The FreeMMG 2 model described in this thesis maximizes state integrity and low bandwidth usage above everything else, which resulted in several trade-offs elsewhere, including in player interaction latency. However, we believe that FreeMMG 2 could be used to serve MMOFPS games in a limited way. For instance, if it is deployed in the same country or region only. Coupled with an evolution of consumer-grade broadband technology and some extra work, MMOFPS support could be achieved. In any case, significant work over FreeMMG 2 is required before MMOFPS game support can be claimed.

### 7.2.6 MMOG virtual currency as an incentive to volunteering as cell node

In Chapter 6 it was shown that, among similar proposals, FreeMMG 2 is the one that requires the largest amount of volunteer nodes to provide the distributed simulation of the virtual world where active players are to play the game. Many peer-to-peer MMOG architectures assume that a P2P MMOG should work like most file-sharing networks where the bulk of peers 'serving' (files) is provided by active users of the service which are also 'consuming' (files). In the case of MMOGs, this means that the players serving the game are the same ones that are also playing the game. However, having peers serving and playing is sub-optimal because of the combined CPU cost of playing and serving. Also, any pending outgoing packets at an active player's machine will increase the interaction latency of that player since packets in the socket architecture are non-preemptable.

Ideally, the volunteers that serve as cell nodes would be continually provided by players while they are not playing the game. This is feasible since, unlike modems over a telephony network, most broadband users are not billed for connection duration or traffic. However, there are a few downsides. One are other costs for serving such as electricity: it is cheaper if you just turn your PC off after playing. Second, players may want their bandwidth for other things while they're not playing. And third, players from the same region would likely play at the same peak hours so most serving peers would be located across the globe at that time, resulting in a prohibitive average latency between active players and the volunteering cell nodes.

As we see it, the definitive solution for this problem is to reward players automatically with in-game currency or virtual items for the computational resources they provide to the overlay. This is feasible since, as we have shown, virtual wealth is actually worth real money (LEHTINIEMI; LEHDONVIRTA, 2007). This would open a new category of player: the *legitimate* Gold Farmer (DIBBELL, 2007). It would not be surprising if

a business model emerged out of providing dedicated and high-quality overlay nodes in return for in-game wealth. This may actually be the greatest strength of hybrid client-server and peer-to-peer MMOG architectures: allowing for a centralized party to create a *concept*, virtual wealth, which then motivates swaths of player-provided machines to band together to run the virtual world as a decentralized (distributed) simulation.

# REFERENCES

AGGARWAL, S.; BANAVAR, H.; KHANDELWAL, A.; MUKHERJEE, S.; RANGARAJAN, S. Accuracy in dead-reckoning based distributed multi-player games. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 3., 2004. **Proceedings...** [S.l.: s.n.], 2004. p.161–165.

AHMED, D.; SHIRMOHAMMADI, S. A Dynamic Area of Interest Management and Collaboration Model for P2P MMOGs. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON DISTRIBUTED SIMULATION AND REAL-TIME APPLICATIONS-VOLUME 00, 2008., 2008. **Proceedings...** [S.l.: s.n.], 2008. p.27–34.

AHMED, D.; SHIRMOHAMMADI, S.; OLIVEIRA, J. Improving gaming experience in zonal MMOGs. In: MULTIMEDIA, 15., 2007. **Proceedings...** [S.l.: s.n.], 2007. p.581–584.

ALEMI, F. An avatar's day in court: a proposal for obtaining relief and resolving disputes in virtual world games. **UCLA Journal of Law & Technology**, [S.l.], v.11, n.2, p.1–54, 2007.

ARENANET. **Guild Wars**. http://www.guildwars.com/.

ASPIN, R.; LE, K. Augmenting the CAVE: an initial study into close focused, inward looking, exploration in ipt systems. In: IEEE INTERNATIONAL SYMPOSIUM ON DISTRIBUTED SIMULATION AND REAL-TIME APPLICATIONS, 11., 2007. **Proceedings...** [S.l.: s.n.], 2007. p.217–224.

ASSIOTIS, M.; TZANOV, V. A distributed architecture for MMORPG. **Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games**, [S.l.], 2006.

BACKHAUS, H.; KRAUSE, S. Voronoi-based adaptive scalable transfer revisited: gain and loss of a voronoi-based peer-to-peer approach for mmog. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 6., 2007. **Proceedings...** [S.l.: s.n.], 2007. p.49–54.

BALAN, R.; EBLING, M.; CASTRO, P.; MISRA, A. Matrix: adaptive middleware for distributed multiplayer games. **Lecture notes in computer science**, [S.l.], v.3790, p.390, 2005.

BARCELLOS, M.; FACCHINI, G.; MUHAMMAD, H. Bridging the gap between simulation and experimental evaluation in computer networks. In: SYMPOSIUM ON SIMULATION, 39., 2006. **Proceedings...** [S.l.: s.n.], 2006. p.286–293.

BARDZELL, J.; JAKOBSSON, M.; BARDZELL, S.; PACE, T.; ODOM, W.; HOUSSIAN, A. Virtual Worlds and Fraud: approaching cybersecurity in massively multiplayer online games. In: DIGRA 2007 CONFERENCE, 2007. **Proceedings...** [S.l.: s.n.], 2007. p.451–742.

BAROSSO, D.; BARTLE, R.; CHAZERAN, C.; ZWART, M. de; DOUMEN, J. M.; GORNIAK, S.; KAŹMIERCZAK, M.; KASKENMAA, M.; LÓPEZ, D. B.; MARTIN, A.; NAUMANN, I.; REYNOLDS, R.; RICHARDSON, J.; ROSSOW, C.; RYWCZYOSKA, A.; THUMANN, M. **Security and privacy in massively-multiplayer online games and social and corporate virtual worlds**. Crete: [s.n.], 2008. ENISA Position Paper. (1).

BAUER, D.; ILIADIS, I.; ROONEY, S.; SCOTTON, P. Communication architectures for massive multi-player games. **Multimedia Tools and Applications**, [S.l.], v.23, n.1, p.47–66, 2004.

BAUGHMAN, N.; LEVINE, B. Cheat-proof playout for centralized and distributed online games. **INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE**, [S.l.], v.1, 2001.

BAUGHMAN, N.; LIBERATORE, M.; LEVINE, B. Cheat-proof playout for centralized and peer-to-peer gaming. **IEEE/ACM Transactions on Networking (TON)**, [S.l.], v.15, n.1, p.1–13, 2007.

BERNIER, Y. Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization. **Proc. of Game Developers ConferenceŠ01**, [S.l.], 2001.

BESKOW, P.; VIK, K.; HALVORSEN, P.; GRIWODZ, C. Latency reduction by dynamic core selection and partial migration of game state. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 7., 2008. **Proceedings...** [S.l.: s.n.], 2008. p.79–84.

BETTNER, P.; TERRANO, M. 1500 archers on a 28.8: network programming in age of empires and beyond. **Presented at GDC2001**, [S.l.], v.2, p.30p, 2001.

BEZERRA, C.; GEYER, C. A load balancing scheme for massively multiplayer online games. **Multimedia Tools and Applications**, [S.l.], 2009.

BHARAMBE, A.; PANG, J.; SESHAN, S. Colyseus: a distributed architecture for online multiplayer games. In: NSDI'06: PROCEEDINGS OF THE 3RD CONFERENCE ON NETWORKED SYSTEMS DESIGN & IMPLEMENTATION, 2006, Berkeley, CA, USA. **Anais...** USENIX Association, 2006. p.12–12.

BLIZZARD ENTERTAINMENT. **WORLD OF WARCRAFT REACHES NEW MILESTONE: 10 million subscribers**. http://www.blizzard.com/us/press/080122.html.

BOSSER, A. Massively multi-player games: matching game design with technical design. In: ACM SIGCHI INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTER ENTERTAINMENT TECHNOLOGY, 2004., 2004. **Proceedings...** [S.l.: s.n.], 2004. p.263–268.

BRANDT, D. **Networking and scalability in EVE online**. Keynote presentation at the 4th ACM SIGCOMM workshop on Network and system support for games (NetGames 2005).

BRUN, J.; BOUSTEAD, P.; SAFAEI, F. Fairness and playability in online multiplayer games. In: IEEE INTERNATIONAL WORKSHOP ON NETWORKING ISSUES IN MULTIMEDIA ENTERTAINMENT (NIME'06), CCNC, 2., 2006. **Proceedings...** [S.l.: s.n.], 2006.

BURO, M. ORTS: a hack-free rts game environment. **Lecture notes in computer science**, [S.l.], p.280–291, 2003.

CALDWELL, P. **Azeroth spreads out**. http://gamespot.com/pc/rpg/worldofwarcraft/news.html?sid=6153338.

CECIN, F. **FreeMMG**: uma arquitetura cliente-servidor e par-a-par de suporte a jogos macicamente distribuidos. Universidade Federal do Rio Grande do Sul. Instituto de Informática. Programa de Pós-Graduação em Computação.

CECIN, F.; BARBOSA, J.; GEYER, C. Freemmg: an hybrid peer-to-peer, client-server, and distributed massively multiplayer game simulation model. In: BRAZILIAN WORKSHOP ON GAMES AND DIGITAL ENTERTAINMENT (WJOGOSŠ 03), 2., 2003. **Proceedings...** [S.l.: s.n.], 2003.

CECIN, F.; BARBOSA, J.; GEYER, C. VFreeMMG: eliminando a trapaça de fabricação de estado no freemmg. In: WORKSHOP ON PARALLEL AND DISTRIBUTED PROCESSING (WSPPD) AT UFRGS, 3., 2005. **Anais...** [S.l.: s.n.], 2005.

CECIN, F.; GEYER, C.; RABELLO, S.; BARBOSA, J. A peer-to-peer simulation technique for instanced massively multiplayer games. **Proceedings of the Tenth IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'06)-Volume 00**, [S.l.], p.43–50, 2006.

CECIN, F.; REAL, R.; MARTINS, M.; JANNONE, R.; BARBOSA, J.; GEYER, C. FreeMMG: a scalable and cheat-resistant distribution model for internet games. In: IEEE DISTRIBUTED SIMULATION AND REAL-TIME APPLICATIONS, 2004. **Anais...** [S.l.: s.n.], 2004. p.83–90.

CECIN, F.; REAL, R.; OLIVEIRA JANNONE, R. de; GEYER, C.; MARTINS, M.; BARBOSA, J. FreeMMG: a scalable and cheat-resistant distribution model for internet games. **8th IEEE International Symposium on Distributed Simulation and Real Time Applications**, [S.l.], 2004.

CHAMBERS, C.; FENG, W.-c.; FENG, W.-c. Towards public server MMOs. In: NETGAMES '06: PROCEEDINGS OF 5TH ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 2006, New York, NY, USA. **Anais...** ACM, 2006. p.3.

CHAMBERS, C.; FENG, W.; FENG, W.; SAHA, D. Mitigating information exposure to cheaters in real-time strategy games. In: NETWORK AND OPERATING SYSTEMS SUPPORT FOR DIGITAL AUDIO AND VIDEO, 2005. **Proceedings...** [S.l.: s.n.], 2005. p.7–12.

CHAN, L.; YONG, J.; BAI, J.; LEONG, B.; TAN, R. Hydra: a massively-multiplayer peer-to-peer architecture for the game developer. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 6., 2007. **Proceedings...** [S.l.: s.n.], 2007. p.37–42.

CHAN, M.; HU, S.; JIANG, J. An efficient and secure event signature (EASES) protocol for peer-to-peer massively multiplayer online games. **Computer Networks**, [S.l.], v.52, n.9, p.1838–1845, 2008.

CHEN, A.; MUNTZ, R. Peer clustering: a hybrid approach to distributed virtual environments. **Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games**, [S.l.], 2006.

CHEN, J.; WU, B.; DELAP, M.; KNUTSSON, B.; LU, H.; AMZA, C. Locality aware dynamic load management for massively multiplayer games. In: PPOPP '05: PROCEEDINGS OF THE TENTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING, 2005, New York, NY, USA. **Anais...** ACM, 2005. p.289–300.

CHEN, K.; JIANG, J.; HUANG, P.; CHU, H.; LEI, C.; CHEN, W. Identifying MMORPG bots: a traffic analysis approach. In: ACM SIGCHI INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTER ENTERTAINMENT TECHNOLOGY, JUNE, 2006., 2006. **Proceedings...** [S.l.: s.n.], 2006. p.14–16.

CHEN, K.-T.; HUANG, P.; LEI, C.-L. How sensitive are online gamers to network quality? **Commun. ACM**, New York, NY, USA, v.49, n.11, p.34–38, 2006.

CHUNG, S. Real Taxation of Virtual Commerce. **Virginia Tax Review**, [S.l.], v.28, n.3, 2008.

CLARKE, I.; SANDBERG, O.; WILEY, B.; HONG, T. Freenet: a distributed anonymous information storage and retrieval system. **Lecture Notes in Computer Science**, [S.l.], p.46–66, 2001.

CLAYPOOL, M.; CLAYPOOL, K. Latency and player actions in online games. **Communications of the ACM**, [S.l.], v.49, n.11, p.39–45, 2006.

COOKE, E.; JAHANIAN, F.; MCPHERSON, D. The zombie roundup: understanding, detecting, and disrupting botnets. In: USENIX SRUTI WORKSHOP, 2005. **Proceedings...** [S.l.: s.n.], 2005. p.39–44.

CORMAN, A.; DOUGLAS, S.; SCHACHTE, P.; TEAGUE, V. A Secure Event Agreement (SEA) protocol for peer-to-peer games. **Proc. ARES**, [S.l.], p.34–41, 2006.

CORMAN, A.; SCHACHTE, P.; TEAGUE, V. A Secure Group Agreement (SGA) protocol for peer-to-peer applications. In: ADVANCED INFORMATION NETWORKING AND APPLICATIONS WORKSHOPS, 2007, AINAW'07. 21ST INTERNATIONAL CONFERENCE ON, 2007. **Anais...** [S.l.: s.n.], 2007. v.1.

CRIPPA, M.; CECIN, F.; GEYER, C. Peer-to-peer support for instance-based massively multiplayer games. **Proceedings of the Brazilian Symposium on Computer Games and Digital Entertainment (SBGAMES'07)**, [S.l.], 2007.

CRONIN, E.; FILSTRUP, B.; JAMIN, S. Cheat-proofing dead reckoned multiplayer games. In: INTERNATIONAL CONFERENCE ON APPLICATION AND DEVELOPMENT OF COMPUTER GAMES, 2., 2003. **Proceedings...** [S.l.: s.n.], 2003.

CRONIN, E.; FILSTRUP, B.; KURC, A.; JAMIN, S. An efficient synchronization mechanism for mirrored game architectures. In: NETWORK AND SYSTEM SUPPORT FOR GAMES: PROCEEDINGS OF THE 1 ST WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 2002. **Anais...** [S.l.: s.n.], 2002.

CRONIN, E.; KURC, A.; FILSTRUP, B.; JAMIN, S. An Efficient Synchronization Mechanism for Mirrored Game Architectures. **Multimedia Tools and Applications**, [S.l.], v.23, n.1, p.7–30, 2004.

CRONIN, E.; KURC, A. R.; FILSTRUP, B.; JAMIN, S. An Efficient Synchronization Mechanism for Mirrored Game Architectures. **Multimedia Tools Appl.**, Hingham, MA, USA, v.23, n.1, p.7–30, 2004.

CURTIS, P. Mudding: social phenomena in text-based virtual realities. **March**, [S.l.], v.5, p.1999, 1992.

DANEZIS, G.; LESNIEWSKI-LAAS, C.; KAASHOEK, M.; ANDERSON, R. Sybil-resistant DHT routing. **Lecture Notes in Computer Science**, [S.l.], v.3679, p.305–318, 2005.

DARLAGIANNIS, V.; HECKMANN, O.; STEINMETZ, R. Peer-to-peer applications beyond file sharing: overlay network requirements and solutions. **e & i Elektrotechnik und Informationstechnik**, [S.l.], v.123, n.6, p.242–250, 2006.

DFC INTELLIGENCE. **Online Game Market Forecasted to Reach $13 Billion by 2011**. http://www.dfcint.com/wp/?p=52.

DI CHEN, B.; MAHESWARAN, M. A cheat controlled protocol for centralized online multiplayer games. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 3., 2004. **Proceedings...** [S.l.: s.n.], 2004. p.139–143.

DIBBELL, J. The life of the Chinese gold farmer. **New York Times**, [S.l.], v.17, 2007.

DOLEV, D. The Byzantine Generals Strike Again. **J. of Algorithms**, [S.l.], v.3, p.1430, 1980.

DOUCEUR, J. The sybil attack. In: PEER-TO-PEER SYSTEMS: FIRST INTERNATIONAL WORKSHOP, IPTPS 2002, CAMBRIDGE, MA, USA, MARCH 7-8, 2002, REVISED PAPERS, 2002. **Anais...** [S.l.: s.n.], 2002. p.251.

EGAN, J. **Game developer sued for virtual losses**. http://www.massively.com/2008/05/20/game-developer-sued-for-virtual-losses/.

EL RHALIBI, A.; MERABTI, M. Agents-based modeling for a peer-to-peer MMOG architecture. **Computers in Entertainment (CIE)**, [S.l.], v.3, n.2, p.3–3, 2005.

ENDO, K.; KAWAHARA, M.; TAKAHASHI, Y. A distributed architecture for massively multiplayer online services with peer-to-peer support. **Proceedings of NetCon'05**, [S.l.], 2005.

ENDO, K.; KAWAHARA, M.; TAKAHASHI, Y. A proposal of encoded computations for distributed massively multiplayer online services. In: ACM SIGCHI INTERNATIONAL CONFERENCE ON ADVANCES IN COMPUTER ENTERTAINMENT TECHNOLOGY, 2006., 2006. **Proceedings...** [S.l.: s.n.], 2006.

ENDO, K.; KAWAHARA, M.; TAKAHASHI, Y. Encoding for secure computations in distributed interactive real-time applications. **Computer Communications**, [S.l.], 2007.

ENDO, K.; YANG, Y.; ZHANG, Z. **RAP: reliable peer to peer network gaming environment using overlapped map tessellation**. [S.l.]: Technical Report (MIT Course Project 6.824), 2006.

Ensemble Studios. **Age of Empires II**. http://www.microsoft.com/games/Age2/.

ERIKSSON, O.; ODENHAMMAR, B. VDSL2: next important broadband technology. **Ericsson Review**, [S.l.], v.1, p.36–46, 2006.

FAN, L.; TAYLOR, H.; TRINDER, P. Mediator: a design framework for p2p mmogs. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 6., 2007. **Proceedings...** [S.l.: s.n.], 2007. p.43–48.

FENG, W. **What's Next for Networked Games?** http://www.thefengs.com/wuchang/work/cstrike/netgames07_keynote.pdf.

FERRETTI, S.; PALAZZI, C.; ROCCETTI, M.; PAU, G.; GERLA, M. FILA, a Holistic Approach to Massive Online Gaming: algorithm comparison and performance analysis. In: ACM INTERNATIONAL CONFERENCE IN COMPUTER GAME DESIGN AND TECHNOLOGY (GDTW 2005), LIVERPOOL, UK, 3., 2005. **Proceedings...** [S.l.: s.n.], 2005. p.68–76.

FERRETTI, S.; ROCCETTI, M. AC/DC: an algorithm for cheating detection by cheating. In: NETWORK AND OPERATING SYSTEMS SUPPORT FOR DIGITAL AUDIO AND VIDEO, 2006., 2006. **Proceedings...** [S.l.: s.n.], 2006.

FERRETTI, S.; ROCCETTI, M.; PALAZZI, C. An Optimistic Obsolescence-based Approach to Event Synchronization for Massively Multiplayer Online Games. **International Journal of Computers and Applications**, [S.l.], v.29, n.1, p.33–43, 2007.

FITCH, C. **Cyberspace in the 21st century: part five, scalability with a big 's'**. http://www.gamasutra.com/features/20010226/fitch_01.htm.

FRITSCH, T.; RITTER, H.; SCHILLER, J. The effect of latency and network limitations on MMORPGs: a field study of everquest2. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 4., 2005. **Proceedings...** [S.l.: s.n.], 2005. p.1–9.

FROHNMAYER, M.; GIFT, T. The TRIBES engine networking model. In: GAME DEVELOPERS CONFERENCE, 2000. **Proceedings...** [S.l.: s.n.], 2000.

FUJIMOTO, R. **Parallel and Distributed Simulation Systems**. [S.l.]: John Wiley & Sons, Inc, 2000.

FUJIMOTO, R. **Parallel simulation: parallel and distributed simulation systems**. [S.l.]: IEEE Computer Society Washington, DC, USA, 2001.

FUNG, Y.; LUI, J. Hack-proof synchronization protocol for multi-player online games. **Multimedia Tools and Applications**, [S.l.], v.41, n.2, p.305–331, 2009.

GAUTHIERDICKEY, C.; ZAPPALA, D.; LO, V.; MARR, J. Low latency and cheat-proof event ordering for peer-to-peer games. In: INTL. WORKSHOP ON NETWORK AND OPERATING SYSTEMS SUPPORT FOR DIGITAL AUDIO AND VIDEO (NOSSDAV), 2004, New York, NY, USA. **Anais...** ACM Press, 2004. p.134–139.

GAUTHIERDICKEY, C.; ZAPPALA, D.; LO, V.; MARR, J. Low latency and cheat-proof event ordering for peer-to-peer games. **Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video**, [S.l.], p.134–139, 2004.

GAUTIER, L.; DIOT, C. Design and Evaluation of MiMaze, a Multi-Player Game on the Internet. In: IEEE INTERNATIONAL CONFERENCE ON MULTIMEDIA COMPUTING AND SYSTEMS, 1998, Washington, DC, USA. **Anais...** IEEE Computer Society, 1998. p.233.

GAUTIER, L.; DIOT, C. 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, 1998, Washington, DC, USA. **Anais...** IEEE Computer Society, 1998. p.233.

GENOVALI, L.; RICCI, L. AOI-Cast Strategies for P2P Massively Multiplayer Online Games. In: IEEE CONSUMER COMMUNICATIONS AND NETWORKING CONFERENCE, 2009. CCNC 2009, 6., 2009. **Anais...** [S.l.: s.n.], 2009. p.1–5.

GILES, K.; MARCHETTE, D.; PRIEBE, C. On the spectral analysis of backscatter data. In: HAWAII INTERNATIONAL CONFERENCE ON STATISTICS AND RELATED FIELDS, 2004. **Anais...** [S.l.: s.n.], 2004.

GLINKA, F.; PLOSS, A.; GORLATCH, S.; MÜLLER-IDEN, J. High-level development of multiserver online games. **International Journal of Computer Games Technology**, [S.l.], v.2008, n.5, 2008.

GOODMAN, J.; VERBRUGGE, C. A peer auditing scheme for cheat elimination in MMOGs. In: NETGAMES '08: PROCEEDINGS OF THE 7TH ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 2008, New York, NY, USA. **Anais...** ACM, 2008. p.9–14.

GORLATCH, S.; GLINKA, F.; PLOSS, A.; MULLER-IDEN, J.; PRODAN, R.; NAE, V.; FAHRINGER, T. Enhancing Grids for Massively Multiplayer Online Computer Games. **Lecture Notes in Computer Science**, [S.l.], v.5168, p.466–477, 2008.

GUO, Y.; BARNES, S. Why people buy virtual items in virtual worlds with real money. **SIGMIS Database**, New York, NY, USA, v.38, n.4, p.69–76, 2007.

HAMPEL, T.; BOPP, T.; HINN, R. A peer-to-peer architecture for massive multiplayer online games. **Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games**, [S.l.], 2006.

HU, S.; LIAO, G. Scalable peer-to-peer networked virtual environment. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 3., 2004. **Proceedings...** [S.l.: s.n.], 2004. p.129–133.

HUANG, G.; HU, S.; JIANG, J. Scalable reputation management with trustworthy user selection for P2P MMOGs. **International Journal of Advanced Media and Communication**, [S.l.], v.2, n.4, p.380–401, 2008.

ID SOFTWARE. **Quake**. http://www.idsoftware.com/games/quake/quake/.

IIMURA, T.; HAZEYAMA, H.; KADOBAYASHI, Y. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 3., 2004. **Proceedings...** [S.l.: s.n.], 2004. p.116–120.

ITO, S.; SAITO, H.; SOGAWA, H.; TOBE, Y. A Propagation of Virtual Space Information Using a Peer-to-peer Architecture for Massively Multiplayer Online Games. In: IEEE INTERNATIONAL CONFERENCEWORKSHOPS ON DISTRIBUTED COMPUTING SYSTEMS, 26., 2006. **Proceedings...** [S.l.: s.n.], 2006. p.44.

IZAIKU, T.; YAMAMOTO, S.; MURATA, Y.; SHIBATA, N.; YASUMOTO, K.; ITO, M. Cheat detection for MMORPG on P2P environments. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 5., 2006. **Proceedings...** [S.l.: s.n.], 2006.

JANG, S.; YOO, J. An Efficient Distributed MMOG Server Using 2Layer-Cell Method. **LECTURE NOTES IN COMPUTER SCIENCE**, [S.l.], v.4469, p.864, 2007.

JHA, S.; KATZENBEISSER, S.; SCHALLHART, C.; VEITH, H.; CHENNEY, S. Enforcing semantic integrity on untrusted clients in networked virtual environments. In: IEEE SYMPOSIUM ON SECURITY AND PRIVACY, 2007., 2007. **Proceedings...** [S.l.: s.n.], 2007. p.179–186.

JIANG, J.; CHIOU, J.; HU, S. Enhancing Neighborship Consistency for Peer-to-Peer Distributed Virtual Environments. **Proceedings of the 27th International Conference on Distributed Computing Systems Workshops**, [S.l.], 2007.

JIANG, X.; SAFAEI, F.; BOUSTEAD, P. An approach to achieve scalability through a structured peer-to-peer network for massively multiplayer online role playing games. **Computer Communications**, [S.l.], v.30, n.16, p.3075–3084, 2007.

JOHN, A.; LEVINE, B. Supporting P2P gaming when players have heterogeneous resources. In: NETWORK AND OPERATING SYSTEMS SUPPORT FOR DIGITAL AUDIO AND VIDEO, 2005. **Proceedings...** [S.l.: s.n.], 2005. p.1–6.

KABUS, P.; BUCHMANN, A. Design of a cheat-resistant P2P online gaming system. In: DIGITAL INTERACTIVE MEDIA IN ENTERTAINMENT AND ARTS, 2., 2007. **Proceedings...** [S.l.: s.n.], 2007. p.113–120.

KABUS, P.; TERPSTRA, W.; CILIA, M.; BUCHMANN, A. Addressing cheating in distributed MMOGs. **Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games**, [S.l.], p.1–6, 2005.

KAMVAR, S.; SCHLOSSER, M.; GARCIA-MOLINA, H. The eigentrust algorithm for reputation management in p2p networks. In: WORLD WIDE WEB, 12., 2003. **Proceedings...** [S.l.: s.n.], 2003. p.640–651.

KAWAHARA, Y.; AOYAMA, T.; MORIKAWA, H. A Peer-to-Peer Message Exchange Scheme for Large-Scale Networked Virtual Environments. **Telecommunication Systems**, [S.l.], v.25, n.3, p.353–370, 2004.

KELLER, J.; SIMON, G. Toward a peer-to-peer shared virtual reality. In: DISTRIBUTED COMPUTING SYSTEMS WORKSHOPS, 2002. PROCEEDINGS. 22ND INTERNATIONAL CONFERENCE ON, 2002. **Anais...** [S.l.: s.n.], 2002. p.695–700.

KELLER, J.; SIMON, G. Solipsis: a massively multi-participant virtual world. In: PDPTA, 2003. **Proceedings...** [S.l.: s.n.], 2003. p.262–268.

KNUTSSON, B.; LU, H.; XU, W.; HOPKINS, B. Peer-to-peer support for massively multiplayer games. **IEEE Infocom**, [S.l.], 2004.

KRAUSE, S. A case for mutual notification: a survey of p2p protocols for massively multiplayer online games. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 7., 2008. **Proceedings...** [S.l.: s.n.], 2008. p.28–33.

KTARI, S.; ZOUBERT, M.; HECKER, A.; LABIOD, H. Performance evaluation of replication strategies in DHTs under churn. In: MOBILE AND UBIQUITOUS MULTIMEDIA, 6., 2007. **Proceedings...** [S.l.: s.n.], 2007. p.90–97.

KU, Y.; CHEN, Y.; WU, K.; CHIU, C. An Empirical Analysis of Online Gaming Crime Characteristics from 2002 to 2004. **Lecture Notes in Computer Science**, [S.l.], v.4430, p.34, 2007.

LAMPORT, L.; SHOSTAK, R.; PEASE, M. The Byzantine Generals Problem. **ACM Trans. Program. Lang. Syst.**, New York, NY, USA, v.4, n.3, p.382–401, 1982.

LAURENS, P.; PAIGE, R.; BROOKE, P.; CHIVERS, H. A novel approach to the detection of cheating in multiplayer online games. In: IEEE INTERNATIONAL CONFERENCE ON ENGINEERING COMPLEX COMPUTER SYSTEMS (ICECCS 2007), 12., 2007. **Proceedings...** [S.l.: s.n.], 2007. p.97–106.

LEE, H.; SUN, C. Load-balancing for peer-to-peer networked virtual environment. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 5., 2006. **Proceedings...** [S.l.: s.n.], 2006.

LEHDONVIRTA, V. Real-money trade of virtual assets: ten different user perceptions. **Proceedings of Digital Arts and Culture (DAC 2005)**, [S.l.], 2005.

LEHTINIEMI, T.; LEHDONVIRTA, V. How big is the RMT market anyway. **Virtual Economy Research Network**, [S.l.], v.3, 2007.

LI, S.; CHEN, C. Interest scheme: a new method for path prediction. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 5., 2006. **Proceedings...** [S.l.: s.n.], 2006.

LIU, H.; LO, Y. DaCAP-A Distributed Anti-Cheating Peer to Peer Architecture for Massive Multiplayer On-line Role Playing Game. In: IEEE INTERNATIONAL SYMPOSIUM ON CLUSTER COMPUTING AND THE GRID, 2008. CCGRID'08, 8., 2008. **Anais...** [S.l.: s.n.], 2008. p.584–589.

LOBATO, R. S.; ULSON, R. S.; SANTANA, M. J. A Revised Taxonomy for Time Warp Based Distributed Synchronization Protocols. In: DS-RT '04: PROCEEDINGS OF THE 8TH IEEE INTERNATIONAL SYMPOSIUM ON DISTRIBUTED SIMULATION AND REAL-TIME APPLICATIONS, 2004, Washington, DC, USA. **Anais...** IEEE Computer Society, 2004. p.226–229.

MALIK, K. **Proximity: a scalable p2p architecture for latency sensitive massively multiplayer online games**. 2005. Tese (Doutorado em Ciência da Computação) — The University of British Columbia.

MANNINEN, T.; KUJANPÄÄ, T. The Value of Virtual Assets–The Role of Game Characters in MMOGs. **International Journal of Business Science and Applied Management**, [S.l.], v.2, n.1, p.21–33, 2007.

MARSHALL, D.; DELANEY, D.; MCLOONE, S.; WARD, T. Exploring the spatial density of strategy models in a realistic distributed interactive application. In: EIGHTH IEEE INTERNATIONAL SYMPOSIUM ON DISTRIBUTED SIMULATION AND REAL-TIME APPLICATIONS, 2004. DS-RT 2004, 2004. **Anais...** [S.l.: s.n.], 2004. p.210–213.

MCGRAW, C.; HOGLUND, G. Online games and security. **IEEE Security & Privacy**, [S.l.], v.5, n.5, p.76–79, 2007.

MDY Industries. **Glider**. http://www.mmoglider.com/.

MERABTI, M.; EL RHALIBI, A. Peer-to-peer architecture and protocol for a massively multiplayer online game. In: IEEE GLOBAL TELECOMMUNICATIONS CONFERENCE WORKSHOPS, 2004. GLOBECOM WORKSHOPS 2004, 2004. **Anais...** [S.l.: s.n.], 2004. p.519–528.

MILLS, D. **Network Time Protocol (Version 3) Specification, Implementation and Analysis**. [S.l.]: RFC 1305, March 1992, 1992.

MÖNCH, C.; GRIMEN, G.; MIDTSTRAUM, R. Protecting online games against cheating. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 5., 2006. **Proceedings...** [S.l.: s.n.], 2006.

MOORE, D.; SHANNON, C.; BROWN, D.; VOELKER, G.; SAVAGE, S. Inferring Internet denial-of-service activity. **ACM Transactions on Computer Systems (TOCS)**, [S.l.], v.24, n.2, p.115–139, 2006.

MULLER, J.; FISCHER, S.; GORLATCH, S.; MAUVE, M. A proxy server-network for real-time computer games. **Lecture notes in computer science**, [S.l.], p.606–613, 2004.

MÜLLER, J.; GÖSSLING, A.; GORLATCH, S. On correctness of scalable multi-server state replication in online games. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 5., 2006. **Proceedings...** [S.l.: s.n.], 2006.

MURRAY, N.; ROBERTS, D. Comparison of head gaze and head and eye gaze within an immersive environment. In: OF 5TH , 2006. **Anais...** [S.l.: s.n.], 2006. p.70–76.

NEUMANN, C.; PRIGENT, N.; VARVELLO, M.; SUH, K. Challenges in peer-to-peer gaming. **ACM SIGCOMM Computer Communication Review**, [S.l.], v.37, n.1, p.79–82, 2007.

NG, B.; WIEMER-HASTINGS, P. Addiction to the Internet and online gaming. **CyberPsychology & Behavior**, [S.l.], v.8, n.2, p.110–113, 2005.

NORDEN, S.; GUO, K. Support for resilient Peer-to-Peer gaming. **Computer Networks: The International Journal of Computer and Telecommunications Networking**, [S.l.], v.51, n.14, p.4212–4233, 2007.

PANTEL, L.; WOLF, L. On the suitability of dead reckoning schemes for games. In: NETWORK AND SYSTEM SUPPORT FOR GAMES, 1., 2002. **Proceedings...** [S.l.: s.n.], 2002. p.79–84.

PAPAGIANNIDIS, S.; BOURLAKIS, M.; LI, F. Making real money in virtual worlds: mmorpgs and emerging business opportunities, challenges and ethical implications in metaverses. **Technological Forecasting and Social Change**, [S.l.], v.75, n.5, p.610–622, 2008.

PELLEGRINO, J. D.; DOVROLIS, C. 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, 2003, New York, NY, USA. **Anais...** ACM, 2003. p.52–59.

RAJAB, M.; ZARFOSS, J.; MONROSE, F.; TERZIS, A. A multifaceted approach to understanding the botnet phenomenon. In: ACM SIGCOMM CONFERENCE ON INTERNET MEASUREMENT, 6., 2006. **Proceedings...** [S.l.: s.n.], 2006. p.41–52.

RIECHE, S.; WEHRLE, K.; FOUQUET, M.; NIEDERMAYER, H.; PETRAK, L.; CARLE, G. Peer-to-Peer-based Infrastructure Support for MMOGs. **Proc. of CCNC, Las Vegas**, [S.l.], 2007.

RITARI, N.; RIVINOJA, J.; CECIN, F.; CONTRIBUTORS. **Outgun**. http://koti.mbnet.fi/outgun/.

ROBERTS, D.; WOLFF, R. Controlling consistency within collaborative virtual environments. In: EIGHTH IEEE INTERNATIONAL SYMPOSIUM ON DISTRIBUTED SIMULATION AND REAL-TIME APPLICATIONS, 2004. DS-RT 2004, 2004. **Anais...** [S.l.: s.n.], 2004. p.46–52.

ROBLES, R.; YEO, S.; MOON, Y.; PARK, G.; KIM, S. Online Games and Security Issues. In: FUTURE GENERATION COMMUNICATION AND NETWORKING, 2008. FGCN'08. SECOND INTERNATIONAL CONFERENCE ON, 2008. **Anais...** [S.l.: s.n.], 2008. v.2.

ROONEY, S.; BAUER, D.; DEYDIER, R. A federated peer-to-peer network game architecture. **IEEE Communications Magazine**, [S.l.], v.42, n.5, p.114–122, 2004.

SANCHEZ-ARTIGAS, M.; LOPEZ, P.; SKARMETA, A. Bypass: providing secure dht routing through bypassing malicious peers. In: IEEE SYMPOSIUM ON COMPUTERS AND COMMUNICATIONS, 2008. ISCC 2008, 2008. **Anais...** [S.l.: s.n.], 2008. p.934–941.

SANGLARD, F. **Quake Engine code review**. http://fabiensanglard.net/quakeSource/quakeSourcePrediction.php.

SCHIANO, D.; WHITE, S. The first noble truth of CyberSpace: people are people (even when they moo). **Proceedings of the SIGCHI conference on Human factors in computing systems**, [S.l.], p.352–359, 1998.

SCHIELE, G.; SUSELBECK, R.; WACKER, A.; HAHNER, J.; BECKER, C.; WEIS, T. Requirements of Peer-to-Peer-based Massively Multiplayer Online Gaming. **Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid**, [S.l.], p.773–782, 2007.

SCREEN DIGEST. **Subscription MMOGs: life beyond world of warcraft**. http://www.screendigest.com/reports/09subscriptionmmogs/pdf/09SubscriptionMMOGs-pdf/view.html.

SEMPSEY, J. Psyber Psychology: a literature review pertaining to the psycho/social aspects of multiuser dimensions in cyber-space. **Journal of MUD Research**, [S.l.], v.2, n.1, 1997.

SHEN, X.; ZHOU, J.; EL SADDIK, A.; GEORGANAS, N. Architecture and evaluation of tele-haptic environments. In: EIGHTH IEEE INTERNATIONAL SYMPOSIUM ON DISTRIBUTED SIMULATION AND REAL-TIME APPLICATIONS, 2004. DS-RT 2004, 2004. **Anais...** [S.l.: s.n.], 2004. p.53–60.

SHI, X. bin; FANG, L.; LING, D.; XIAO-HONG, C.; YUAN-SHENG, X. A cheating detection mechanism based on fuzzy reputation management of P2P MMOGs. **SNPD 2007**, [S.l.], v.1, p.75–80, 2007.

SHI, X.; GAO, J.; DU, L. A coordinator election algorithm for P2P MMOG. In: VISUAL INFORMATION ENGINEERING, 2008. VIE 2008. 5TH INTERNATIONAL CONFERENCE ON, 2008. **Anais...** [S.l.: s.n.], 2008. p.265–270.

SONY ONLINE ENTERTAINMENT. **PlanetSide**. http://www.planetside.com.

SRIVATSA, M.; LIU, L. Vulnerabilities and security threats in structured overlay networks: a quantitative analysis. In: COMPUTER SECURITY APPLICATIONS CONFERENCE, 2004. 20TH ANNUAL, 2004. **Anais...** [S.l.: s.n.], 2004. p.252–261.

STEINMAN, J. Scalable parallel and distributed military simulations using the SPEEDES framework. In: ELECTRONIC SIMULATION CONFERENCE (ELECSIM95), INTERNET, 2., 1995. **Proceedings...** [S.l.: s.n.], 1995.

STEINMAN, J. S. Breathing Time Warp. In: PADS '93: PROCEEDINGS OF THE SEVENTH WORKSHOP ON PARALLEL AND DISTRIBUTED SIMULATION, 1993, New York, NY, USA. **Anais...** ACM, 1993. p.109–118.

STUTZBACH, D.; REJAIE, R. Understanding churn in peer-to-peer networks. In: ACM SIGCOMM CONFERENCE ON INTERNET MEASUREMENT, 6., 2006. **Proceedings...** [S.l.: s.n.], 2006. p.189–202.

THAWONMAS, R.; KASHIFUJI, Y.; CHEN, K. Detection of MMORPG bots based on behavior analysis. In: INTERNATIONAL CONFERENCE IN ADVANCES ON COMPUTER ENTERTAINMENT TECHNOLOGY, 2008., 2008. **Proceedings...** [S.l.: s.n.], 2008. p.91–94.

THOMAS, J.; ESSAAIDI, M. Problems of Security in Online Games. **Information Assurance and Computer Security**, [S.l.], p.168, 2006.

THOMPSON, Z. The in-game economics of Ultima Online. In: COMPUTER GAME DEVELOPERŠS CONFERENCE, SAN JOSE, CA, 2000. **Anais...** [S.l.: s.n.], 2000.

TRIEBEL, T.; GUTHIER, B.; PLOTKOWIAK, T.; EFFELBERG, W. Peer-to-Peer Voice Communication for Massively Multiplayer Online Games. In: IEEE CONSUMER COMMUNICATIONS AND NETWORKING CONFERENCE, 2009. CCNC 2009, 6., 2009. **Anais...** [S.l.: s.n.], 2009. p.1–5.

TYCHSEN, A.; HITCHENS, M. Ghost Worlds-Time and Consequence in MMORPGs. **Lecture Notes in Computer Science**, [S.l.], v.4326, p.300, 2006.

Valve Software. **Source Multiplayer Networking**. http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking.

VAN DEN BOSSCHE, B.; VERDICKT, T.; DE VLEESCHAUWER, B.; DESMET, S.; DE MULDER, S.; DE TURCK, F.; DHOEDT, B.; DEMEESTER, P. A platform for dynamic microcell redeployment in massively multiplayer online games. In: NETWORK AND OPERATING SYSTEMS SUPPORT FOR DIGITAL AUDIO AND VIDEO, 2006., 2006. **Proceedings...** [S.l.: s.n.], 2006.

VIK, K. Game state and event distribution using proxy technology and application layer multicast. In: ACM INTERNATIONAL CONFERENCE ON MULTIMEDIA, 13., 2005. **Proceedings...** [S.l.: s.n.], 2005. p.1041–1042.

VIK, K.; GRIWODZ, C.; HALVORSEN, P. Applicability of group communication for increased scalability in MMOGs. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 5., 2006. **Proceedings...** [S.l.: s.n.], 2006.

VIK, K.; GRIWODZ, C.; HALVORSEN, P. Dynamic group membership management for distributed interactive applications. In: IEEE CONFERENCE ON LOCAL COMPUTER NETWORKS, 2007. LCN 2007, 32., 2007. **Anais...** [S.l.: s.n.], 2007. p.141–148.

VLEESCHAUWER, B. D.; BOSSCHE, B. V. D.; VERDICKT, T.; TURCK, F. D.; DHOEDT, B.; DEMEESTER, P. Dynamic microcell assignment for massively multiplayer online gaming. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 2005, New York, NY, USA. **Anais...** ACM Press, 2005. p.1–7.

WALDO, J. Scaling in games and virtual worlds. **Commun. ACM**, New York, NY, USA, v.51, n.8, p.38–44, 2008.

WALDO, J.; WYANT, G.; WOLLRATH, A.; KENDALL, S. A note on distributed computing. **Lecture Notes in Computer Science**, [S.l.], p.49–64, 1997.

WARNER, D.; RAITER, M. Social context in massively-multiplayer online games (MMOGs): ethical questions in shared space. **International Review of Information Ethics**, [S.l.], v.4, n.7, 2005.

WEBB, S.; SOH, S. Cheating in networked computer games: a review. In: DIGITAL INTERACTIVE MEDIA IN ENTERTAINMENT AND ARTS, 2., 2007. **Proceedings...** [S.l.: s.n.], 2007. p.105–112.

WEBB, S.; SOH, S.; LAU, W. Enhanced mirrored servers for network games. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 6., 2007. **Proceedings...** [S.l.: s.n.], 2007. p.117–122.

WEBB, S.; SOH, S.; LAU, W.; PERTH, W. RACS: a referee anti-cheat scheme for p2p gaming. In: NOSSDAV, 2007. **Proceedings...** [S.l.: s.n.], 2007. p.37–42.

WEBB, S.; SOH, S.; TRAHAN, J. Secure Referee Selection for Fair and Responsive Peer-to-Peer Gaming. In: PRINCIPLES OF ADVANCED AND DISTRIBUTED SIMULATION, 2008. PADS'08. 22ND WORKSHOP ON, 2008. **Anais...** [S.l.: s.n.], 2008. p.63–71.

WHITE, E. **Massively Multiplayer Online Fraud:** why the introduction of real world law in a virtual context is good for everyone. Northwestern Journal of Technology and Intellectual Property, Northwestern University School of Law.

WIERZBICKI, A.; KASZUBA, T. Practical trust management without reputation in peer-to-peer games. **Multiagent and Grid Systems**, [S.l.], v.3, n.4, p.411–428, 2007.

WIKIPEDIA. **Half-Life (video game) — Wikipedia, The Free Encyclopedia**. [Online; accessed 30-July-2008].

WIKIPEDIA. **Interactive fiction — Wikipedia, The Free Encyclopedia**. [Online; accessed 30-July-2008].

WIKIPEDIA. **Game Developers Choice Awards — Wikipedia, The Free Encyclopedia**. [Online; accessed 30-July-2008].

WIKIPEDIA. **MUD — Wikipedia, The Free Encyclopedia**. [Online; accessed 30-July-2008].

WIKIPEDIA. **PLATO (computer system) — Wikipedia, The Free Encyclopedia**. [Online; accessed 30-July-2008].

WIKIPEDIA. **MUD - Wikipedia**. [Online; accessed 30-July-2008].

WOODCOCK, B. An analysis of MMOG subscription growth. **MMOGCHART. COM**, [S.l.], v.23, 2008.

WOODCOCK, B. MMOG Subscriptions Market Share by Genre. **MMOGCHART. COM**, [S.l.], v.23, 2008.

WUESTEFELD, K. Do you still use a database? In: OOPSLA '03: COMPANION OF THE 18TH ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES, AND APPLICATIONS, 2003, New York, NY, USA. **Anais...** ACM, 2003. p.101–101.

YAMAMOTO, S.; MURATA, Y.; YASUMOTO, K.; ITO, M. A distributed event delivery method with load balancing for mmorpg. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 4., 2005. **Proceedings...** [S.l.: s.n.], 2005. p.1–8.

YAN, J.; RANDELL, B. A systematic classification of cheating in online games. In: ACM SIGCOMM WORKSHOP ON NETWORK AND SYSTEM SUPPORT FOR GAMES, 4., 2005. **Proceedings...** [S.l.: s.n.], 2005. p.1–9.

YANG, B.; GARCIA-MOLINA, H. Designing a super-peer network. In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING, 2003. **Proceedings...** [S.l.: s.n.], 2003. p.49–62.

YEUNG, S.; LUI, J.; LIU, J.; YAN, J. Detecting cheaters for multiplayer games: theory, design and implementation [1]. In: IEEE CONSUMER COMMUNICATIONS AND NETWORKING CONFERENCE, 2006. CCNC 2006, 3., 2006. **Anais...** [S.l.: s.n.], 2006. v.2.

YONEKURA, T. A Proposal of Dead Reckoning Protocol in Distributed Virtual Environment based on the Taylor Expansion. In: INTERNATIONAL CONFERENCE ON CYBERWORLDS, 2006., 2006. **Proceedings...** [S.l.: s.n.], 2006. p.107–114.

YOON, U. South Korea and indirect reliance on IP law: real money trading in mmorpg items. **Journal of Intellectual Property Law & Practice**, [S.l.], v.3, n.3, p.174, 2008.

YU, A.; VUONG, S. MOPAR: a mobile peer-to-peer overlay architecture for interest management of massively multiplayer online games. In: NETWORK AND OPERATING SYSTEMS SUPPORT FOR DIGITAL AUDIO AND VIDEO, 2005. **Proceedings...** [S.l.: s.n.], 2005. p.99–104.

ZHOU, R.; HWANG, K. Powertrust: a robust and scalable reputation system for trusted peer-to-peer computing. **IEEE Transactions on parallel and distributed systems**, [S.l.], v.18, n.4, p.460, 2007.