UNIVERSIDADE DA BEIRA INTERIOR
Covilhã | Portugal

Re-engineering Jake2 to Work on a Grid using the GridGain Middleware

Ricardo Jorge da Cruz Alexandre

Submitted to University of Beira Interior in partial fulfillment of the requirement for the degree of
Master of Science in Engenharia Informática

Supervised by Prof. Dr. Abel Gomes

Departmento de Informática
Universidade da Beira Interior
Covilhã, Portugal
http://www.di.ubi.pt

Resumo

Com o advento dos Massively Multiplayer Online Games (MMOGs), os engenheiros e projectistas de jogos depararam-se com várias questões, tais como: Como conseguir uma grande quantidade de clientes a jogar simultaneamente no mesmo servidor? Como garantir uma boa qualidade de serviço (QoS) para um grande número de clientes? Quantos recursos são necessários? Como otimizar estes recursos ao máximo? Uma resposta possível e abrangente a estas questões baseia-se na utilização de *grid computing*.

Tendo em conta a natureza parallela e distribuída da *grid computing*, pode dizer-se que *grid computing* garante à partida maior escalabilidade relativamente a um crescente número de jogadores, garante que o tempo de comunicação entre os clientes e os servidores é tão baixo quanto possível e garante também uma melhor gestão dos recursos disponíveis (e.g., memória, CPU, utilização balanceada dos *cores*, etc.) do que a computação sequencial.

No entanto, o tema central desta dissertação não é *grid computing*, mas sim o processo de re-engenharia de software de um jogo FPS (*First Person Shooter*) multi-jogador, designado por Jake2, de modo a transformá-lo num MMOG que, afinal de contas, é executado numa *grid*.

Abstract

With the advent of Massively Multiplayer Online Games (MMOGs), engineers and designers of games came across with many questions that needed to be answered such as, for example, "how to allow a large amount of clients to play simultaneously on the same server?", "how to guarantee a good quality of service (QoS) to a great number of clients?", "how many resources will be necessary?", "how to optimize these resources to the maximum?". A possible answer to these questions relies on the usage of *grid computing*.

Taking into account the parallel and distributed nature of grid computing, we can say that grid computing allows for more scalability in terms of a growing number of players, guarantees shorter communication time between clients and servers, and allows for a better resource management and usage (e.g., memory, CPU, core balancing usage, etc.) than the traditional serial computing model.

However, the main focus of this thesis is not about grid computing. Instead, this thesis describes the re-engineering process of an existing multiplayer computer game, called Jake2, by transforming it into a MMOG, which is then put to run on a grid.

Contents

Resumo iii

Abstract v

Contents vii

List of Figures xi

List of Tables xiii

List of Algorithms xv

1 Introduction 1

1.1 Motivation and Objectives 2
1.2 Research Contributions 3
1.3 Organization of the Thesis 4
1.4 Publications 4
1.5 Target Audience 4

2 MMOGs: The State-of-the-Art 5

2.1 A Brief History 5
2.2 MMOGs Definition and Features 8
2.3 Platform-based Classification of MMOGs 9
2.3.1 PC MMOGs 10
2.3.2 Browser-based MMOGs (BBMOGs) 10
2.3.3 Console MMOGs 11
2.3.4 Mobile MMOGs 12
2.4 Genre-based Classification of MMOGs 12
2.4.1 Massively Multiplayer Online First Person Shooters 14

2.4.2 Massively Multiplayer Online Role-Playing Games 14
2.4.3 Massively Multiplayer Online Real-Time Strategy Games . . . 15
2.5 MMOGs Issues and Solutions 16
2.5.1 Networking Issues and Solutions 17
2.5.2 Security Issues and Solutions 18
2.5.3 Persistence Issues and solutions 19
2.5.4 Scalability Issues and Solutions 19
2.6 Network Topologies 21
2.6.1 Client-Server 21
2.6.2 Client-Master-Server 23
2.6.3 Client-Master-Slave 25
2.6.4 Server-Clustering 27
2.6.5 Peer-To-Peer (P2P) 29
2.6.6 Mirrored-Server 34
2.6.7 Peer-Clustering 35
2.6.8 Comparison of Networking Architectures 36
2.7 Computing Models 39
2.7.1 Parallel Computing 39
2.7.2 Distributed Computing 40
2.7.3 Cluster Computing 40
2.7.3.1 Cluster Configurations 41
2.7.3.2 Cluster Categories 42
2.7.4 Grid Computing 43
2.7.5 Computing Models Comparison 45
2.8 Summary 46

3 Grid-based Re-engineering of Jake2 49
3.1 Software Re-Engineering 49
3.2 Grid Middleware: A Brief Analysis 50
3.3 GridGain 52
3.3.1 GridGain Architecture 52
3.3.2 Split/Reduce Mechanism 55
3.3.3 Map/Reduce Mechanism 56
3.3.4 Network Protocols 56
3.3.5 GridGain Middleware Installation and Functioning 58

3.4 Reverse engineering of Jake2 . . . 58
3.4.1 Jake2 code . . . 59
3.4.2 Jake2 Playing Modes . . . 59
3.4.3 Jake2 Networking . . . 60
3.4.4 Jake2 Persistence . . . 61
3.4.5 Jake2 Game World . . . 62
3.4.6 Jake2 Scalability . . . 62
3.5 Module Re-engineering of Jake2 . . . 63
3.5.1 Client Module Changes . . . 65
3.5.2 QCommon Module Changes . . . 69
3.5.3 Proxy Module . . . 71
3.6 Data Re-Engineering . . . 73
3.6.1 Database Module . . . 73
3.6.2 Game Logic Module . . . 73
3.7 Control-flow re-engineering . . . 75
3.7.1 GridGain Module . . . 75
3.7.2 Server Module . . . 77
3.8 Discussion . . . 82
3.9 Summary . . . 84

4 Scalability: Experiments and Results 85
4.1 Experiment Methodology . . . 85
4.2 Experimental Scenarios . . . 87
4.2.1 Standard Grid . . . 88
4.2.1.1 Hardware Configurations . . . 88
4.2.1.2 Jake2 versus Jake2Grid . . . 90
4.2.2 Constrained Grid . . . 93
4.2.2.1 One Machine . . . 93
4.2.2.2 Multiple Machines . . . 97
4.2.3 Singular Grid . . . 98
4.2.4 Comparison of the Test Scenarios . . . 98
4.3 Summary . . . 99

5 Conclusions 101

References **103**

List of Figures

2.1 BBMMOG Sherwood Dungeon (taken from http://www.sherwooddungeon.com/). 10
2.2 Console MMOG Final Fantasy XIV (taken from Final Fantasy XIV). . . 11
2.3 Mobile MMOG Fantasy World Rhynn (taken from http://www.rhynn.com/). 12
2.4 MMOFPS PlanetSide (taken from PlanetSide). 14
2.5 MMORPG World of Warcraft (taken by the author during gameplay). . 15
2.6 MMORTS Shattered Galaxy (taken from Shattered Galaxy). 16
2.7 Client-Server Architecture. 21
2.8 Client-Master-Server architecture. 24
2.9 Client-Master-Slave Architecture. 26
2.10 Server-Clustering Architecture. 27
2.11 Peer-To-Peer (P2P) Architecture. 30
2.12 Mirrored-Server Architecture. 35
2.13 Peer Clustering Architecture. 36
2.14 Parallel Computing architecture. 40
2.15 Distributed Computing Architecture. 41
2.16 Tightly Coupled Cluster Computing Architecture. 42
2.17 Grid Computing Architecture. 43
2.18 Grid networks doing a task division. 44

3.1 GridGain Architecture (taken from GridGain web site at http://www.gridgainsystems.com/wik
3.2 Split-reduce division mechanism. 55
3.3 Map-reduce division mechanism. 56
3.4 A Jake2 snapshot. 58
3.5 Jake2 playing modes: (a) single player mode; (b) multiplayer mode. . . 60
3.6 Jake2 networking architecture on a LAN. 61
3.7 Jake2 networking architecture with master-server in a WAN. 61
3.8 Jake2Grid architecture. 64

3.9 Jake2Grid Functioning. 70
3.10 Jake2Grid grid functioning. 75

4.1 Jake2 Latency measurement example. 87
4.2 Comparison of Jake2 with Jake2Grid in respect to latencies and execution times listed in Tables 4.2 and 4.3. 92
4.3 Comparative graphic of Jake2 versus Jake2Grid scalability measurements using stationary bots. 94
4.4 Comparative chart of Jake2 and Jake2Grid scalability measurements using stationary shooting bots. 95
4.5 Comparative chart of Jake2 VERSUS Jake2Grid scalability measurements using shooting and moving bots. 96

List of Tables

2.1 Comparison of networking architectures. 37
2.2 Comparison of Computing Models. 45

3.1 Grid middlewares. 51

4.1 Hardware configurations. 89
4.2 Execution time and latency measurements of Jake2 running on a single 2 × Quad-Core machine with 1Gb Ethernet card. 90
4.3 Execution time and latency measurements of Jake2Grid running on a single 2 × Quad-Core machine with 1Gb Ethernet card. 91
4.4 Jake2 versus Jake2Grid scalability measurements using stationary bots. 94
4.5 Jake2 versus Jake2Grid scalability measurements using stationary shooting bots. 95
4.6 Jake2 VERSUS Jake2Grid scalability measurements using shooting and moving bots. 96
4.7 Latency (ms) measurements per node using shooting and moving bots on a constrained grid. 97
4.8 Jake2 versus Jake2Grid scalability measurements using shooting and moving bots. 98
4.9 Comparison related to the number of players using shooting and moving bots. 99
4.10 Comparison related to the scalability gain using shooting and moving bots. 99

List of Algorithms

1 Original Jake2 Main Class 65
2 Jake2Grid Client Main Class 66
3 Jake2Grid Client: Client module 67
4 Jake2Grid Client: Qcommon module 69
5 Jake2Grid Server: Game logic module 74
6 Jake2Grid Server Main Class 78
7 Original Jake2: Server module 79
8 Jake2Grid Server: Server module 80
9 Jake2Grid Server: Server module 81

Chapter 1

Introduction

The first network computer games appeared in the 1970's. Examples are SpaceWar [1], Maze Wars [2], MUD [3], etc.. These games, initially developed to run over Ethernet [4, 5] networks or the ARPANET [6] (i.e., the predecessors of the Internet), later evolved to nowadays called online games. Online games began to attract huge amounts of players, leading to a rapid growth of Internet usage. The main limitation of these games was due to the Internet connection speed of most users. However, both these online multiuser (later referred as multiplayer) games and the Internet connection speed evolved, and online games popularity kept growing. With time a new kind of games, that could support hundreds of thousands of players to play in simultaneously at the same time appeared. This category of online games is nowadays referred to as Massively Multiplayer Online Games (MMOGs) [7, 8].

Since the appearance of MMOGs that programmers and developers have investigated new ways of supporting the increasingly large amounts of players, while keeping or improving quality of service (QoS) [9]. Providing QoS is not a trivial task because the number of players in MMOGs is not constant (i.e., it keeps increasing and decreasing over time). Thus, it is important to workout solutions that offer scalability, in terms of quality of service (QoS) and optimize resources usage (i.e., memory consumption, processing time, etc.), for a large amount of players. QoS is directly related with the latency (i.e., measure that indicates the time that a client needs to send and retrieve an instruction to and from the server) and bandwidth (i.e., user's Internet connection speed) required to run MMOGs.

Regardless of their genre (e.g., First Person Shooter, Real-Time Strategy, Role-

Playing Game, etc.), all MMOGs suffer from lack of scalability. In order to improve MMOGs scalability (i.e., reduce latency and bandwidth requirements), programmers and developers are studying network architectures. Through these studies alternative topologies (e.g., p2p, grid computing, etc.) and alternative network protocols (e.g., RTP, GTP, etc.) were and are still being developed for games [10, 11, 12, 13]. These studies are oriented to the improvement of MMOGs, namely, to achieve better QoS and higher numbers of players.

Other study fields related to MMOGs include mobile network architectures for MMOGs [14], the implementation of grid-based computing MMOGs [15], the effects of prolonged exposure to MMOGs (e.g., violent/addictive behaviors related to MMOGs) on players [16], MMOGs business models [17], games usage for education and/or military training purposes [18, 19, 20, 21, 22, 23, 24], and tools [25, 26, 27, 28, 29, 30, 31] to aid in MMOGs development.

One of the most important goals of this thesis is to answer to the following question: "How to make a MMOG more scalable using grid computing?" In order to prove that grid computing is an adequate solution for MMOGs scalability problems, we have re-designed an open source First Person Shooter (FPS) game, called Jake2, and transformed it into a MMOG.

Jake2 game was transformed into a MMOG and was placed to run on a grid using a grid middleware, called GridGain. The result was the design and implementation of a Massively Multiplayer Online First Person Shooter game (MMOFPS), called Jake2Grid, which runs on a grid and that is scalable with the number of players.

1.1 Motivation and Objectives

The main objective of this thesis is the transformation of a multiplayer First Person Shooter (FPS), called Jake2, into a Massively Multiplayer Online First Person Shooter (MMOFPS), called Jake2Grid. A MMOG is a specific type of computer game that, in principle, can be played by hundreds of thousands of players. Usually, a game is called a MMOG if it supports at least 1000 players simultaneously [8, 32, 33].

Real-time applications like online games use a lot of resources, such as, CPU processing, memory and a high network bandwidth. This is not much of a problem for multiplayer games, where we are limited to a maximum of 32 players per game server (i.e., applications that handle a game level/map and its clients procession). The real problem appears when we talk about games that can support hundreds of thousands of players simultaneously, as it is the case of MMOGs.

Scaling a multiplayer game into a MMOG is not a trivial task, as MMOGs need more resources than a multiplayer game. Most multiplayer games can be hosted in the user/player machine. However, this is not a good solution for MMOGs because a user machine does not possess the amount of resources that a MMOG needs to run. Also a multiplayer game supports a small amount of players, whereas a MMOG supports a large amount of players.

In order to provide the necessary scalability with a reasonable quality of service (QoS), we need to use some sort of distributed and parallel computing. For this thesis, we have used grid computing on a single computer grid, as a milestone to transform the multiplayer version of Jake2 into the MMOG version of Jake2Grid. It is worth noting that grid computing has already proved to be efficient when handling scalability problems for non real-time applications. However, there is not much work with real-time applications. Therefore, this is an opportunity to prove that grid computing can be a solution to the lack of scalability of real-time applications like MMOGs.

Possibly, the main motivation for writing this thesis is due to the fact that grid computing is a relatively recent technology and there is not much usage of this technology involving real-time applications, as it is the case of MMOGs.

1.2 Research Contributions

The major contributions of this thesis are:

1) A software re-engineering process that enable us to transform a multiplayer game (Jake2) into a MMOFPS game (Jake2Grid) using grid computing.

2) We have also proved that, in some circumstances, grid computing can be used in real-time applications. In particular, grid computing can be used to improve the scalability of games in general.

1.3 Organization of the Thesis

In addition to the present chapter that introduces the motivation, objectives, research challenges to be solved, contributions, organization of this thesis, publications, and target audience, this document has four further chapters:

- **Chapter** 2. This chapter overviews the state-of-the art of MMOGs, with a focus on their genres, networking topologies and computing models.
- **Chapter** 3. This chapter describes the re-engineering process that enabled us to transform Jake2 into Jake2Grid. This has been accomplished by using software re-engineering principles and grid computing.
- **Chapter** 4. In this chapter, we carry out a battery of experiments in order to achieve the grid configuration for Jake2Grid that guarantees scalability for a growing number of players while keeping real-time. The idea here was to satisfy the scalability and real-time requirements of our MMOFPS.
- **Chapter** 5. This is the last chapter of this thesis, where relevant conclusions are drawn and possible future work is outlined.

1.4 Publications

As a consequence of the work developed in this thesis, we have already published the following papers:

G. Amador, R. Alexandre, and A. Gomes. Re-Engineering Jake2 to work on a Grid. In the Proceedings of the 7th Conference on Telecommunications, Santa Maria da Feira, Portugal, May 3-5, 2009 [34].

R. Alexandre, P. Prata and A. Gomes. A Grid Infrastructure for Online Games. In the Proceedings of the 2009 International Conference on Information Sciences and Interaction Sciences, Seoul, South Korea, November 24-26, ACM International Conference Proceeding Series, Vol. 403, 2009 [35].

1.5 Target Audience

The target audience of this thesis embraces not only the communities of video games, but also those of distributed and parallel computing and networking.

Chapter 2

MMOGs: The State-of-the-Art

This chapter reviews the state-of-the-art of Massively Multiplayer Online Games (MMOGs), from simple online games to complex MMOGs, like the famous World of Warcraft. This includes a classification of MMOGs, the differences between MMOGs and other games, the advantages/disadvantages of MMOGs in relation to other games, protocols, topologies and computational models used by MMOGs, and software engineering issues presented by MMOGs.

2.1 A Brief History

Online computer games appeared in the 1970s. These games can only be played through the Internet. The first network computer game, developed by MIT students (late 60s) and later transformed into a two player network version by Rick Blomme (early 70s), was SpaceWar [1]. SpaceWar was the first action game, that allowed two players to compete over a network (i.e., ARPANET). SpaceWar was developed by Rick Blomme for the Programmed Logic for Automatic Teaching Operations (PLATO) time-sharing system hosted in the University of Illinois. SpaceWar supported two players playing it from different machines simultaneously. Due to the creation of SpaceWar, many more online computer games were developed, which in turn contributed to the growth of Internet usage [1]. The rapid growth and usage of Internet led to the creation of multiuser online games, later called multiplayer online games. These games can be played by more than just two individuals.

The first network First Person Shooter (FPS) game, Maze Wars [2], appeared in

1973. It was developed at NASA's Ames Research Center in California by high school summer internees. The game two-player capability was achieved by two computers connected using serial cables. Since both machines used formatted protocol packets to transfer information between each other, this could be considered the first peer-to-peer (P2P) computer video game. Later, in the fall of 1973, one of the authors of the game (Greg Thompson) re-engineered the game to run on a mainframe. The game's new version allowed up to eight simultaneous players. Thus, Maze Wars became the first client-server game ever.

The first game of its genre, that allowed more than two players to compete simultaneously, was MUD (MultiUser Dungeon or Domain), developed by Roy Trubshaw and Richard Bartle, in the late 1970s, and it is considered as the first multiuser (adventure) online game [3, 36, 37]. Partly, online game development grew very rapidly because Trubshaw and Bartle shared the source code of the MUD they created. MUD growing popularity allowed it to be licensed to CompuServe, which was online until 1999. MUD provided a virtual environment where players use text to create and describe the world they lived in. Virtual worlds/environments can be games, but they are also worlds where players can create characters in contexts such as, for example, Second Life. In virtual worlds, players provide textural proof of who they are, what they look like, and how they act and react to other players in the world. These worlds are worlds where role-playing is valued and players are judged by how well they perform in the world [38]. MUD can be considered the root of Role-Playing Games (RPGs) [38]. MUDs are considered as text-based games that allow players to have a high level of control over how they create and play characters [38]. MUDs first run on university networks and later on bulletin boards, and were based on a client-server architecture [3, 36, 37]. MUD popularity and number of players grew so rapidly, that they evolved in the early 90s [38] to a new kind of games, nowadays referred to as Massively Multiplayer Online Games (MMOGs) [3, 36, 37].

A very important system that led to the creation of MUDs was Habitat, which was developed by F. Randall Farmer and Chip Morningstar in 1986. Habitat was a graphical chat environment that enabled interaction with the environment and supported objects (e.g., chairs, tables and houses) [32]. During the same period of time of the development of MUDs, a new category of games appeared in the market. These new games were based on the principle of hand-eye coordination (e.g., Pong, Super Mario, Sonic, etc.), and evolved to more sophisticated action games (e.g, Doom) [38].

Gary Tarolli developed a flight simulator demonstration program, in 1983. Later in 1984, the simulator followed the steps of Maze Wars. First, the two player functionality was added to the simulator, through serial cables, followed by a client-server version that allowed multiple players over a LAN. However, in 1986 with the addition of UDP support, the simulator now called SGI Dogfight [39], became the first game ever to use the Internet protocol.

In 1984, Sherrick's released the first version of its turn-based strategy game, Trade Wars [40]. The game source was extended in the 90s by several developers to support online gaming over the Internet. Thus, Trade Wars can be called the father of online strategy games.

Around 1990, games that combined advanced graphics with narrative elements originated a new category of games, called Role-Playing Games (RPGs). These games allow a player to experience parts of the game world (i.e., a virtual world where players play and which can be divided into several game levels/maps), solve puzzles, solve mysteries, enter a combat with Non-Player Characters (NPCs) (i.e., characters controlled by the computer), and permits a high sense of interactivity. In these games, players have the role of a main character in the narrative and are allowed to experience the story from a first-person point of view [38, 36].

MUDs are different from RPGs because a MUD player can create and be anything he/she wants, whereas an RPG constrains the player's identity, forcing him/her to take a predetermined role and narrative [38]. With the evolution of MUDs and RPGs, it was only a matter of time before the first MMOG appeared.

The first MMOG was Meridian 59, developed in 1996 [32]. Meridian 59 allowed a maximum number of 250 players to play simultaneously in the same world over the Internet. The first famous MMOG was Ultima Online, which was developed in 1997 by Origin Systems [32]. After Ultima Online has been released, many other MMOGs were produced (e.g., Everquest Online) [32, 38, 36]. Today MMOGs offer players an experience of either simple to complex 2D/3D graphics in large explorable virtual worlds. There are many things that are important in MMOGs apart from the interaction with other players, like the computer controlled characters and other computer controlled objects such as, for example, game scenarios and other game items (e.g., trees, bridges, etc.) [41].

After the success of the first text-based generation of MMOGs (i.e., MUDs), the second generation or interactive 2D graphics-based MMOGs (e.g., Tibia [42]) was created, followed by the third generation or interactive 2D/3D graphics MMOGs (e.g, World of Warcraft [36]). Finally, in 2002 the first game of the fourth generation of MMOGs appeared, named Final Fantasy XI (FFXI) for Sony Playstation 2 [43]. Fourth generation MMOGs allows players to play this type of games in other platforms either than computers, such as mobile phones (e.g., Everquest II [44]) or game consoles (e.g., Final Fantasy XIV for PS3 [45] or The Lord of the Rings: The Battle for Middle Earth II for Xbox 360 [46]), or through a browser (i.e., browser-based MMOGs or BBMMOGs), such as Sherwood Dungeon [47].

2.2 MMOGs Definition and Features

As previously mentioned, MMOGs evolved from MUDs. These types of games can only be played online (i.e., Internet games). They are the only competition games, that allow large numbers of simultaneous players playing in a game world. Some authors consider a game to be a MMOG if it supports 250 or more players (e.g., Meridian 59) [32, 48]. However, other authors consider a game to be a MMOG if it allows 1000 up to hundreds of thousands of players playing simultaneously [8].

In MMOGs, players interact with other players, and with the game world. Players either control several units/characters (i.e., strategy MMOGs) or a single character (i.e., action and adventure MMOGs), called avatar. The player can interact with the game world elements (e.g., creatures, objects, etc.), with other players, and with non-player characters (NPCs) [49, 50]. NPCs are in charge of giving game support (i.e., provide support information, trade objects, etc.).

MMOGs have features that make them unique. These features are divided into two classes: technical and non-technical. Perhaps, the most relevant non-technical features are:

- *Number of Players*: MMOGs are competition games that support a large number of players playing simultaneously (e.g., more than 1000 players) [8].
- *Role Play*: Independently of the genre of the game, every player plays the role of a character in the game [7, 51].
- *Socialization*: Players socialize in a virtual context, so they tend to learn social skills like: how can a person manage a small group?, how can a person coordinate

and cooperate with other people?, and how can a person participate in a sociable interaction with other people? Thus, MMOGs promote the creation of online communities inside the virtual world, because playing the game often requires team-work and cooperation [8, 51].

- *Virtual Economy*: MMOGs runs a virtual economy, i.e., players can trade game items with other players and NPCs [32, 52].
- *Funding*: MMOGs require great funding for maintenance, development, and hardware acquisition (e.g., servers, clusters, etc.). The funding for MMOGs can come from the purchase by the clients of the software. Also, clients may pay a regular fee (e.g., a monthly fee) to play the game (i.e., also known as premium accounts), or to access exclusive extras (e.g., exclusive game areas, faster game progression, etc.). Clients might also buy game items with real money [53].

The most important MMOGs technical features (also called requirements) are:

- *Networking*: MMOGs can only be played through the Internet [7, 54].
- *Security*: Protection against cheats and other types of security attacks [7, 33].
- *Persistence*: MMOGs have persistent worlds [8, 32, 33]. A persistent world is a virtual world that exists regardless of the players being connected to it or not. Thus, the game world might change or evolve even if players are not connected to it. Persistent worlds cannot be paused. The player status only changes when he is playing. However, its actions (i.e., something he/she does that affects the game world or other players) are permanent [7, 8, 52, 33].
- *Scalability*: A MMOG must cope with many players interacting with one another in a game world simultaneously. A game can only be considered a MMOG if the number of players is over 1000, but this number usually goes up to hundreds of thousands [7, 8, 54].

2.3 Platform-based Classification of MMOGs

MMOGs can run on distinct platforms (i.e., laptop, PC, console, mobile devices). MMOGs can also be played through a browser [55]. A brief overview of browser-based, mobile, console and PC MMOGs can be found below.

2.3.1 PC MMOGs

The most played MMOGs nowadays are PC-based (i.e., they were exclusively designed and published for PCs and laptops) such as World of Warcraft (Figure 2.5).

PCs and laptops nowadays include modems which allow players to play MMOGs anywhere at anytime, as long as it is possible to establish a connection to the Internet (i.e., players must connect to the Internet in order to play, this can be done either through cable or wireless connections).

Also, PCs and laptops offer programmers more powerful hardware and software than consoles, which is an advantage as more powerful MMOGs can be programmed (i.e., MMOGs with more sophisticated graphics can be programmed).

2.3.2 Browser-based MMOGs (BBMOGs)

BBMOGs, such as Sherwood Dungeon, can run on any platform (i.e., laptop, PC, console, mobile devices) over a browser [47]. A snapshot of this MMOG is shown in Figure 2.1.

Figure 2.1: BBMMOG Sherwood Dungeon (taken from http://www.sherwooddungeon.com/).

These MMOGs appeared as a first attempt to remove the requirement of installing a client software. Since browsers already possess all the functionalities (e.g., visualization/access to remote content, plugin updates, etc.), required by MMOGs clients, the client software can be replaced by an easy to install and update browser plugin [56, 57].

These MMOGs for server-based topologies (addressed in the following section) provide several advantages, namely: better security on the client side and cost reduction. Security on the client side is increased since all accesses and data are only available on the server side (i.e., client cheating is reduced). There is also a cost reduction on the game support, since client software updates and maintenance are easily deployed as a browser plug-in update. Also, to use different network topologies just the server software must be modified. Thus, the client software requires less maintenance [58].

In BBMMOGs exclusively, funding mostly comes from advertising. However, as in other MMOGs, clients may purchase game content with real money [59, 60, 61].

2.3.3 Console MMOGs

Consoles, such as PS3, possess a modem that allows a player to play with other players over the Internet. This allowed companies to develop MMOGs for consoles, such as Final Fantasy XIV for PS3 [45] or The Lord of the Rings: The Battle for Middle Earth II for Xbox 360 [46]. These MMOGs were exclusively designed and published for consoles. A snapshot of Final Fantasy XIV is shown in Figure 2.2.

Console MMOGs tend to be cheaper when compared to PC MMOGs, because consoles are cheaper than PCs or even laptops (e.g., nowadays a PS3 costs 300 euros, while a laptop or a PC can cost more than 500 euros). However, the console requirements (i.e, consoles require a TV screen in order for a player to view and play the game) restrain users from playing games anywhere whenever they want.

Also, the majority of players prefer playing a game using a keyboard and mouse than with a controller, because the keyboard and mouse allow a player to have a different control experience while playing the game (i.e., the keyboard is used for character movement, while the mouse is used for character actions and game camera view control).

Figure 2.2: Console MMOG Final Fantasy XIV (taken from Final Fantasy XIV).

2.3.4 Mobile MMOGs

The first Mobile MMOG to be released was Fantasy World Rhynn [44]. This MMOG is illustrated in Figure 2.3.

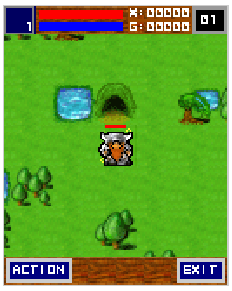


Figure 2.3: Mobile MMOG Fantasy World Rhynn (taken from http://www.rhynn.com/).

Mobile devices portability (i.e., small size and weight) made them the ideal platform to play online from any place at any time. Around 2003, most mobile devices started to support platform independent APIs, such as Java or .NET. These APIs, allowed the development and improvement of games (e.g., such as MMOGs) over mobile networks, for several distinct mobile devices (i.e., cellphones, PDA's, etc.). However, these applications are restricted to the device's memory capabilities. Thus, a MMOG's client application is easily deployed and maintained regardless of the device hardware, as long as the hardware supports the client application [62, 63, 13].

These MMOGs present lower costs for the client when purchasing the device. However, the device portability and, consequently, the playing time is limited to the device's energy supply. Nevertheless, most mobile networks are not as efficient as computer or console networks [64, 65, 66].

2.4 Genre-based Classification of MMOGs

As any video game, the characteristic element of a MMOG is the competition between players. What distinguishes a MMOG from other video games is the number of players. In terms of the number of players, we classify video games as follows:

- *One Player Games*: In this case, the player plays against the computer (e.g., *Solitaire* game).

- *Two Player Games*: These games involve two players playing on the same computer (e.g., *Pong* game).
- *Multiplayer Games*: A multiplayer game involves a small number of players as clients of a server that runs the game over a LAN or even the Internet (e.g., *Quake* game).
- *Massively Multiplayer Online Games (MMOGs)*: MMOGs are played by a large number of players, i.e., at least 1000 players, only over the Internet (e.g., *World Of Warcraft*). In MMOGs, players interact with other players, and with the game world, by means of a 3D character, called avatar [50, 33, 36].

Regardless of the platform they run or if they are browser-based or not, video games including MMOGs, have three main genres/types, namely [8, 67]:

- *Action*: An action MMOG is usually called Massively Multiplayer Online First-Person Shooter (MMOFPS). FPSs are *action* games where players kill other players for points. FPS game worlds are generally small worlds, sometimes even rooms, where players can pick items such as armors, weapons, bullets, health, etc., and can fight against other players.
- *Adventure*: An adventure MMOG is known as Massively Multiplayer Online Role-Playing Game (MMORPG). RPGs are *adventure* games where players choose avatars and walk in an adventure virtual world. This virtual world generally describes the conflict between good and evil. Players choose a side to play and cooperate with other players from the same side to kill players from the opposite side.
- *Strategic*: A strategic MMOG is designated as Massively Multiplayer Online Real-Time Strategy Game (MMORTS). RTSs are *strategic* games where players have an army under their command and try to kill other player armies in the attempt of dominating the virtual world. The virtual world of RTSs generally describes a war world where players construct bases, build up armies and kill other players.

We will now overview each of the types mentioned above, focusing on MMOGs.

2.4.1 Massively Multiplayer Online First Person Shooters

MMOFPSs, such as PlanetSide (see Fig. 2.4) are action MMOGs. In this MMOG genre players individually, or as members of a team, kill other players for points, or to achieve a common goal to the team. In classical First Person Shooters, players compete in small worlds, where players can pick items (e.g., armors, weapons, bullets, health, etc.). The virtual world of a MMOFPS is either a large-scale world where massive number of players compete, or several small worlds where multiple small scale competitions take place simultaneously at the same time [33, 36].

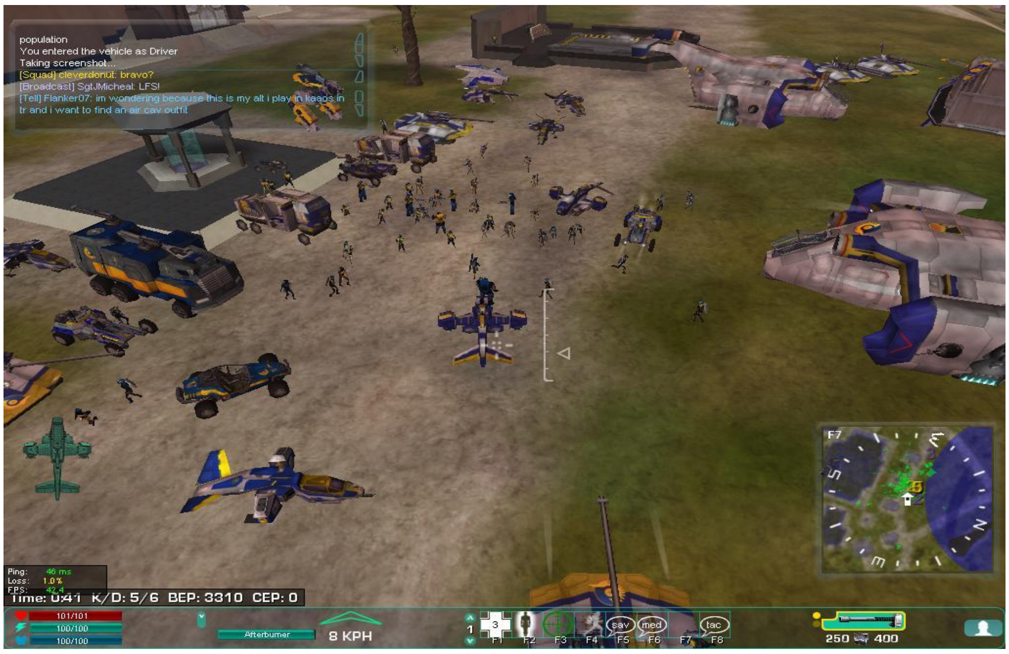

population
You entered the vehicle as Driver
Taking screenshot...
[Squad] cleverdonut: bravo?
[Broadcast] SgtJMicheal: LFS!
[Tell] Flanker07: im wondering because this is my alt i play in kaaos in tr and i want to find an air cav outfit

Figure 2.4: MMOFPS PlanetSide (taken from PlanetSide).

2.4.2 Massively Multiplayer Online Role-Playing Games

Adventure MMOGs (i.e., MMORPGs), such as World of Warcraft (see Fig. 2.5), are the most famous type of MMOGs (i.e., the ones with more players and the most played ones [36]).

In Role-Playing Games (RPGs) players travel through a virtual world. This virtual world generally describes the conflict between good and evil. Players choose a side to play, and individually or by cooperating with players from the same side, strive to kill members (i.e., players or artificial controlled game characters) from the opposite side [8, 68, 36].

Figure 2.5: MMORPG World of Warcraft (taken by the author during gameplay).

The worlds of RPGs usually tell a story of worlds where magic and/or science are present. These worlds can either be entirely fictitious (e.g., futuristic worlds where several races aside humans exist and interact) or based on historical eras (e.g., World War II). In these worlds, players can engage in a huge variety of activities, either with other players or individually. These activities include fights, explorations, team making, chats, buying/selling items, special events or marriages, kill enemies, monsters and level their characters, etc. In RPGs a player is motivated to cooperate with other players, to achieve a higher character level and progress in the game story. Higher character levels mean a stronger character, able to reach more difficult game objectives, and gain access to higher character features [8, 68, 36].

2.4.3 Massively Multiplayer Online Real-Time Strategy Games

Strategic MMOGs, such as Shattered Galaxy (see Fig. 2.4), are designated as Massively Multiplayer Online Real-Time Strategy Games (MMORTS).

MMORTS are war games, where players, unlike other MMOGs genres, control several units (e.g., company of soldiers, tank divisions, etc.). In these games, players choose different pieces/sides, and through military tactics (e.g., alliances, resource control, direct engagement, etc.), they pursue the domination of an entire virtual world. These virtual worlds can either be futuristic universes, historical events (e.g., world wars, ancient history, medieval era, etc.), to mystical non-existent worlds (i.e., worlds where magic and/or science can coexist). In these virtual worlds players, besides controlling their units, are also able to extract resources (e.g., food, wood, etc.), construct game buildings (e.g., support infrastructures, military units training/assembling facilities, etc.), and create more units. Buildings and units are made using the available resources. Thus, controlling resources is also an integrating part of playing [69].

Skarsnik: woot, he had H missles .. ew :P
RebeccaRei: yep
OwnYou: Repocers a3
Visit the War Room at www.sgalaxy.com for the latest galactic news!
RebeccaRei: as of right now i do
HeXoN: if you take out the mants i can help out alot :)

Figure 2.6: MMORTS Shattered Galaxy (taken from Shattered Galaxy).

2.5 MMOGs Issues and Solutions

MMOGs issues have to deal with the technical features reported in section 2.2, namely: networking issues, security issues, persistence issues, and scalability issues, . This section just reviews such issues and their possible solutions. The reader is referred to [70, 71, 54] for further details.

2.5.1 Networking Issues and Solutions

The most important networking issues of MMOGs are: bandwith, latency, and quality of service (QoS).

MMOGs require a high bandwidth, because many player requests must be dealt with simultaneously. In order to optimize bandwidth, three strategies can be applied, namely: reducing the number of messages to send, reducing the size of each message, and reducing the number of hosts to which an information is sent [72, 70]. Reducing the number of messages to send can be achieved using aggregation techniques (i.e., techniques of grouping sets of messages and send them as one large message). By means of compression techniques the size of each message may be reduced. Thus, less information (i.e., in terms of package size) to communicate. Finally if multiple hosts are relatively close, they can be grouped into a single host. Thus, reducing the number of packages to exchange between clients and hosts [72, 70].

Game latency (i.e., the time taken from a client's request until the respective server's response) in MMOGs must be as low as possible. Latency can be affected by several factors such as: network protocol, network architecture, Internet connection speed, quality of the Internet service provider, configuration of firewalls, geographical locations (i.e, servers can be dispersed all around the world), server capacities and its configuration (e.g., computing power, used computing model, etc.) [72, 70]. Acceptable latency levels (i.e., the game must be playable in real-time) can be achieved by using the dead reckoning technique or alternative networking protocols (e.g., RTP, GTP, etc.) instead of the standard ones (i.e., TCP, UDP) [10, 11, 12, 13]. The dead reckoning technique allows a client player to guess the state (e.g., players position) of another player, based on the last known position of that player, when updates are missing [72, 73, 74]. In some games, UDP is used (e.g., EverQuest) instead of TCP (e.g., World of Warcraft) [75], because the package loss rate is not significant to justify the extra time to confirm packages arrival. Other protocols, such as Game Transport Protocol (GTP) [10] and Real-Time Transport Protocol (RTP) [12], were developed specifically for handling MMOGs requirements, in order to surpass limitations of the TCP and UDP protocols. RTP and GTP were developed specifically for the transmission of MMOGs event data by means of specific functions.

Quality of service (QoS) in MMOGs is influenced by the bandwidth and latency (i.e., high bandwidth usage results in large latency and poor QoS). QoS must be ensured in order to provide a real-time experience to the clients. Therefore, to provide a good QoS, we need to improve latency and bandwidth (through previously referred techniques) [9, 76].

2.5.2 Security Issues and Solutions

Players spend a lot of time and money in MMOGs (e.g., to play, to buy game items, to have more resources at their disposal, etc.). In addition to players funding, companies also get funds to maintain an improve their MMOGs from publicity; hence, the need to maintain or increase the number of players. Aside from the networking problems previously discussed, player state in the game is saved in the host side. Thus, as any network service, game security must be guaranteed from those people interested in either stealing personal data (i.e., game software and/or network infrastructure flaw exploitation) or gaining unfair advantage over other players (i.e., cheating) [77, 68, 78, 79, 80, 71, 54].

There are two types of security threats to deal with in MMOGs, namely: cheating (e.g., resource farming, ninja looting, ganking, AFK macroing, and gold buying [77, 68, 78, 79, 80, 71, 54]), and exploitation of flaws on the game software (i.e., server and/or client masquerading and the usage of modified client software) and/or network infrastructure (i.e., distributed denial of service) [81, 77].

Resource farming and ninja looting come from players who only spend time in gathering items, to sell either for real or game money, or to use in other accounts. Ganking consists in directly killing or luring to death lower level players. Ganking is performed to either evolve quicker or get free items. AFK macroing consists in using bots (i.e., artificial intelligence controlled clients) [82, 83] or macros to control a player's character/avatar while they are absent. Gold buyers are players that buy game "gold" (i.e., money used in game to buy items and services), with real life money [68]. Cheating security issues are usually handled either by the company that developed the MMOG or by another company specialized in MMOGs security [68, 84].

Server masquerading is accomplished when an attacker poses as a game server, in order to get access to players private data (e.g., usernames, passwords, CD keys, etc.). Client masquerading is when an attacker impersonates a valid user, to get unauthorized access to the service, in order to obtain or modify clients information. Distributed denial of service (DDOS) is when attackers attempt to disrupt access to the game server(s).

The client software can be modified by an attacker to add functionalities that allow him/her to perform flaw exploits in the game server(s) [81, 77].

Server masquerading can be addressed through the implementation of a digital certificate system. Client masquerading can be solved by the implementation of complex passwords (i.e., hard to crack) chosen either by the client or the MMOGs developer's company. Distributed denial of service (DDOS) can be solved by the introduction of routers, which would play a role when attackers are preventing users from accessing a service (i.e., it would provide the client another way to access the service he/she wants). Another alternative would be providing clients patches, that would allow registered users to access the service that they want to use. Modified client software usage could be addressed by some sort of verification (e.g., client version identification, client version disk occupancy size, runtime verification, mobile guards, etc.) [85, 86].

2.5.3 Persistence Issues and solutions

As previously mentioned, the players and sometimes the game state in MMOGs are stored on the server side. Thus, when a server goes down or crashes for any unexpected reason, the states must be saved [87, 88, 89]. To provide persistence, players and game state can be stored in either files or databases, synchronized among all the game servers. Databases are preferable to files, due to their dynamic data access and modification features (e.g., faster access to data than in files), and their security and failure tolerance techniques (e.g., replication).

2.5.4 Scalability Issues and Solutions

As previously mentioned, MMOGs must support a large number of players. Scalability has three main kinds: networking scalability (i.e., network usage), player scalability (i.e., number of players), and hardware scalability (i.e., number of servers). Networking scalability refers to how to reduce the network usage in order to increase the number of simultaneous communications a network can handle [73, 90]. Player scalability refers to how many simultaneous players a server machine (e.g., a PC, laptop, etc., which runs one or more game servers) can handle, while offering a good QoS [91, 30]. Hardware scalability refers to how many machines are needed to handle a certain amount of simultaneous players and what is the cost (i.e., in the case of MMOG this amount of players is at least 1000) [92, 93]. Server machines can be either large dedicated servers, capable of handling thousands of clients simultaneously, or clustering servers, that are capable of handling hundreds of thousands of players.

Scalability is influenced by aspects, such as: hardware capabilities [70, 94, 54], load balancing [70, 94, 54], and areas of interest (AOI) [95, 93]. The main hardware issue lies in the fact that no single computer can process simultaneously a huge number of MMOGs client requests [70, 94, 54]. The load balancing issue refers to how to correctly (i.e., maintaining a playable QoS) distribute processing of the game world by the server machines. AOI (i.e., circular area centered on a player and is used to design objects close to the player and to calculate collisions) main problem is that players in the same AOI need to send/receive changes to all the other players in the AOI.

To improve scalability, some solutions have been proposed in the literature, namely: distributed architectures [96, 97] and world distribution techniques (i.e. zoning [98, 25, 99, 73] or sharding [44, 99, 100, 101], cloning [100], and instancing [44, 99, 100, 101]). Distributed architectures are networking architectures (e.g., master-servet, P2P, etc.) where the server load is distributed across several entities/computers. The main objective of these architectures is to split the server load into a number of machines that normally work in parallel. An example of a distributed architecture is ScalaMO [97], in which the virtual world is divided into geographical square zones.

There are four ways to divide the world, namely: zoning, sharding, cloning, and instancing. Zoning consists in dividing the whole world into small independent cells (or zones). When splitting a large game world into one or more cells, that can be handled by a single game server, allowing each server to adjust its cells load. Sharding can be applied by running independent copies of the entire game world on multiple servers (i.e., one copy per server), where each copy only supports a small number of players (depending on the server's capabilities). Cloning is very similar to sharding. What makes cloning different is that events (e.g., a maintainance communication to all servers) on all game servers are synchronized (i.e., what happens in one server, happens in all the game servers). Instancing is an hybrid of the zoning and sharding techniques. This technique consists in replicating (i.e., sharding) specific zones of a large game world (i.e., zoning). These zones are limited to a very small group of players (e.g., five players). Using this technique, a group of players from the same zone can only interact with the players on the same shard.

2.6 Network Topologies

Currently, MMOGs use two main network topologies: client-server and its variants (i.e., client-master-server, client-master-slave and server-clustering) [102, 10, 103, 104, 105, 15, 12, 13], peer-to-peer (P2P) and its hybrid variants (i.e, Mirrored-server and Peer-Clustering) [106]. All previously addressed MMOGs genres (i.e., MMOFPSs, MMORPGs, MMORTSs, BBMMOGs) can run on any of the two main network topologies. As a margin note, it is important to refer that these topologies are platform independent (i.e., regardless of whether the games run on consoles, laptops, mobile devices, etc.) [107]. A brief description of each topology is given in sequel, focusing on their capability of addressing the lack of scalability of MMOGs. Finally, a comparative analysis of each topology advantages and disadvantages is provided.

2.6.1 Client-Server

The most common network architecture used by multiplayer games is the client-server architecture [108, 37]. This is a two-tier architecture where the client talks directly to a server, without any intermediate server, as illustrated in Figure 2.7.

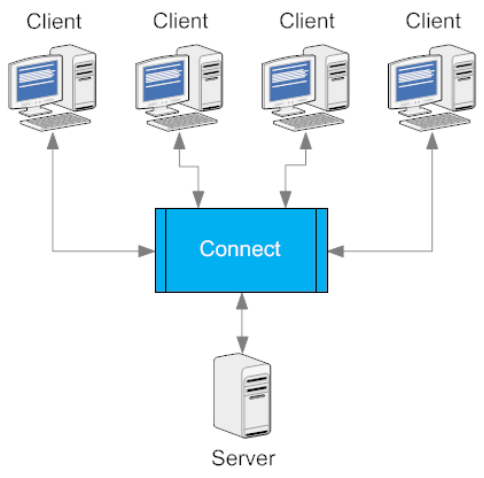


Figure 2.7: Client-Server Architecture.

Clients send requests and receive data updates to/from the server. The server processes all client requests and data, and sends results that occur on the virtual world back to the clients. Typically, this architecture (Figure 2.7) is adequate for small environments, i.e., environments with less than 50 users, and has several advantages, namely [109]:

- It is easy to design, because of its simplicity.
- It retains full access control over the game, because only one entity (server) controls the game.
- It is a predictable model, because it is possible to predict possible failures and to correct them.
- It is easy to update, because only the server sends/receives data to/from all clients.
- There is a single game state which is located on the server, i.e., the server handles the whole game state and sends it to the clients.
- There is one governing authority, i.e, the server that prevents cheating and handles game events.

However, it has important disadvantages, namely:

- *Limited load balancing.* All data is sent to and processed by the server [109, 101, 93].
- *No fault tolerance.* This means that if the server fails, the game will become unavailable because it has only one game server [109, 101, 93].
- *Lack of scalability.* It is costly to maintain the server-side with the increase of the number of players, unless we add more server machines [109, 101, 93].
- *High latency.* In particular for interactive FPS games (e.g. Quake), the gameplay is very sensitive to latencies between the clients and the server, as players must wait at least a round trip time for input events (e.g. key presses and tilts) to be reflected in the client interface. If the number of player increases, the server is not able to give response to many players in time. This causes the server to have a high latency, which in turn can affect player latencies. Also the game server can be geographically far from the clients, which can increase the communication time that takes clients to send and receive responses from the server [91].

A way to solve both latency and scalability issues of the client server is to add more game servers and have these game servers close to clients.

Quake uses the client-server architecture. In this case, the server is in charge of computing the entire game state (e.g. gravity, lighting, map information, and temporary player state) and distribute game state updates to the clients. The server's main task is to maintain and handle the dynamic game entities: avatars, monsters, rockets and backpacks. The server runs an event control loop that detects when a player is hit, uses physics-based models to compute the position of objects, translates key presses into game actions as jumps and weapon fires, etc. In addition to the game state and game play controllability, the server also acts as a session manager that accepts client connections, starts and ends games. The client is responsible for the 3D graphics rendering of the game, and works as the interface between the player and the server so that it passes commands to the server that are triggered by button events and avatar movements from the client player.

2.6.2 Client-Master-Server

The previous client-server architecture is not effective because the server easily becomes overwhelmed when the number of players increases. To properly scale to hundreds (or possibly thousands) of users, we need to move to a three-tier architecture, i.e. a client-master-server architecture (see Figure 2.8). This topology is an variant of the client-server architecture. The client-master-server topology adds an extra layer (i.e., the master or proxy) to the client-server architecture.

The master is a control entity that is in charge of both assigning clients to active game servers and maintaining a list of active game servers available to all clients. This architecture allows direct addition of new game servers, usually distributed around the globe. Also, this architecture allows a player to create its own game server or servers, which will be added to the active servers game list [91].

This architecture allows for several geographical distributed game servers. A MMOG that currently uses this type of architecture is America's Army. America's Army is an MMOFPS used by the USA Army for training purposes of massive amounts of soldiers [110]. Clients connect to one of the masters geographically away from each other. Each master provides to the clients a list of associated game servers, from which the client may choose one to join. This list includes all servers handled by all masters.

After connecting to a game server, the clients send and receive data from/to the server (e.g., player movement, kills, etc.), which is used to update the game on both ends. Master can redirect clients among themselves, if a client joins a server handled by another master [111]. Game servers process data of one or more game worlds at a time (e.g., players interaction with other players and the world, world evolution, etc.).

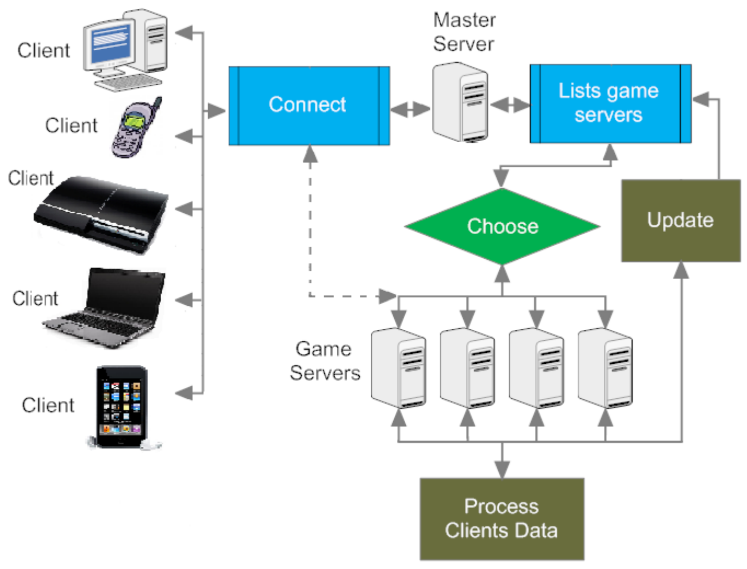


Figure 2.8: Client-Master-Server architecture.

The client-master-server has some advantages [91, 112]:

- *Distribution*: In principle, each server manages a game level/map state and is responsible for processing all data of its connected players. This works as a rudimentary load balancing.

- *Scalability*: Potentially, scalability increases by adding more game servers.

- *Fault tolerance*: Game servers continue to work even if the master fails, because the player only needs the master to join a game server. Besides, when a game server fails, all the others keep running.

- *Reduced latency*: Since there are more servers and they may be distributed across the world, game latency is reduced for closer players.

However, this architecture has some disadvantages [91, 112]:

- *Load Balancing*: Levels are distributed among servers but not their load, i.e, if a server is overloaded and another server is free it cannot use the resources of the second server.

- *No failover*: If the master server fails, players are not able to see which game servers are active and cannot choose one to play, because each master is independent of the other masters. If game server A fails, usually, another game server cannot resume the game level/map that A was running. Usually, another server can run the same game level/map as server A, but this is considered as new game level/map, i.e, game states are reset (game world state is cleared) and players join a new copy of game level/map A.

- *Scalability*: When the number of players increases and all game servers are full, the obvious solution is to add new game servers.

- *High latency*: Although game latency is reduced in this architecture, this may be not enough because clients may connect to game servers that are located very far away, increasing this way the communication time between clients and servers.

The client-master-server architecture is capable of handling MMOG's scalability issues by allowing the addition of more servers. However, this represents a rudimentary scalability solution because server load is not distributed, i.e., game servers can not share resources when needed. Thus, if a server is full/overloaded and another one is free/underloaded, the former cannot use the resources of the latter.

2.6.3 Client-Master-Slave

Although the client-master-server architecture offers a rudimentary scalability solution (i.e., great increase in the number of players can be dealt by adding more servers), it does not ensure servers optimal resource usage (i.e., game load is not distributed). In order to improve the scalability of the client-master-server architecture, a new architecture, the client-master-slave architecture (see Figure 2.9), was proposed [103, 112, 88].

As in the client-master-server, the clients connect to masters that provide a list of available servers (i.e., groups of slaves). Thus, in the client-master-slave, individual machine servers are replaced by groups of individual machine servers (i.e., slaves), scattered all over the globe. However, in the client-master-slave architecture, clients can no longer create their own game servers. Game servers are only added by the company that owns the game commercial exploration rights. The game load (i.e., processing of part of the game world) assigned to specific slaves is equally distributed, among the servers in the same group. This is achieved with the masters new functionality of controlling work division among slaves (i.e., which specific work goes to what specific server in the server group). Servers in the same group can cooperate/communicate among themselves, independently of the master (i.e., they can use each other resources). Therefore, best servers resource usage is achieved, and consequently a MMOG that resorts to this topology becomes more scalable [88]. This architecture offers some advantages [103, 88]:

- *Independent Game State*: Each server has its own game state and is responsible for processing its players data.
- *Improved scalability and flexibility*: Player capacity increases with each game server (i.e, more servers mean more players).
- *Slave Failover*: A slave server is capable of taking over other slave server when the former fails.

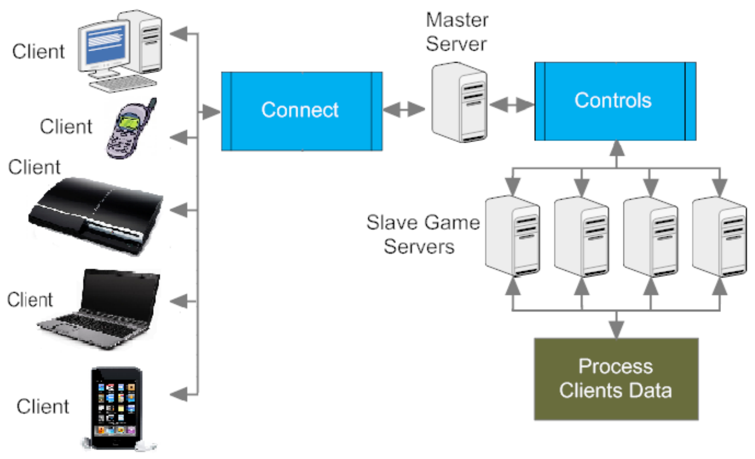


Figure 2.9: Client-Master-Slave Architecture.

However, this architecture as some significant disadvantages [103, 88]:

- *Master Failover*: If the master server fails all slave servers fail.

The client-master-slave architecture scalability can be improved further if the slave servers are not independent and distributed around the globe. In order to achieve a far better scalability a new architecture was developed, the server-clustering architecture.

2.6.4 Server-Clustering

The server-clustering architecture (see Figure 2.10) evolved from the client-master-slave architecture. The server-clustering is the most used network topology for MMOGs such as Everquest [113, 114]. This type of architecture can be seen in Figure 2.10.

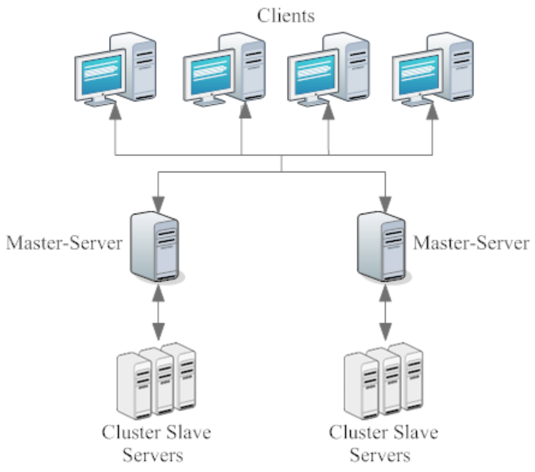


Figure 2.10: Server-Clustering Architecture.

Much like in the previous topologies, the client connects to a master that provides a list of available servers. In this architecture, one or several slave groups are replaced by one or more clusters scattered around the globe. A cluster works as an individual high performance machine that behaves as a group or groups of slave servers connected over a Local Area Network (LAN) [115, 114]. However, instead of having a group or groups of independent slave servers on the same LAN, we have a slave group or groups connected by a high performance bus (i.e., it transfers data between components inside a computer or transfers data between computers). Thus, using a cluster reduces significantly the inter-servers communication time, and consequently MMOGs that use this topology are more scalable (i.e., can respond faster to more clients that on the client-master-slave topology). The master has the same functionalities as in the client-master-slave architecture [116, 117, 118].

An example of a game that uses this topology is the MMORPG Everquest. Everquest is run in over 1500 server machines distributed all around the world, having the same capacity of one of the world's top 100 supercomputers [115, 114]. Sony is the provider of this set of machines. Sony is capable of hosting more than 150000 players from all around the world simultaneously [115, 114]. Everquest is mainly divided into two modules: client and server. Graphics and sound are handled on the client's side. Everything else, such as, players action, data, game world among others are stored on these machines. Everquest is divided into several parallel worlds. These worlds form an interconnected gathering of servers, called a cluster. The size of this cluster depends on how many users can play simultaneously [115, 114]. Each game world is handled by a cluster that is formed by 20 or 30 dual-processor computers. Each processor is responsible for handling parts of the game world (e.g., forests, towns, mountains).

This architecture offers significant advantages to MMOGs:

- *Performance*: More and better computers are used, i.e., the machines that are used have much better resources available such as processing power and memory [115, 114].

- *Scalability and Load balancing (flexibility)*: MMOG simulation is easily distributed among several game servers, which allows more players simultaneously [115, 114]. Game servers share resources, i.e, a high loaded game server can use the resources of another low loaded game server [116, 119].

- *Failover*: If any server fails, another usually, takes its place and resumes the failure server processing, i.e., if slave server A fails, any of the remaining slave servers can resume A's game level/map processing. Usually, in server-clustering more than one cluster node can handle the same game level/map of another node, But now these can be synchronized because slaves are sharing information and resources about what they are doing, i.e., if slave A fails, and slave B is processing the same game level/map A, players who were disconnected from A can simply go to slave B. If the master-server fails, usually another machine of the cluster takes over the master-server [116, 119].

- *Reduced Latency*: Game levels/maps are distributed locally among the cluster slave servers. Usually, game levels/maps can also be distributed around the world, but are distributed in clusters, which can be accessed by master servers.

For example, let us assume that there are 2 master servers, one in England (A) and another in France (B), each one managing 5 game levels/maps. Let us now assume that a Spanish user is at the same distance from A and B, and has finished level A and wants to move to level B. In these circumstances, the player latency does not change that much [116, 119].

- *Software cost*: When the number of players increases and all game levels/maps are full, cluster technologies allow the creation of another node in any cluster machine that still has free resources, i.e., by simply adding more software (nodes) and not hardware, this type of application can support a great number of players with less cost than the client-master-server architecture [116, 119].

However, this architecture has two main significant disadvantages [116, 114]:

- *Hardware cost*: Hardware is more costly when compared to those of other architectures, because it is required to have several computers with high computing power in order to handle thousands of players simultaneously.
- *High bandwidth usage*: A huge number of players access a single machine, which leads to a high bandwidth usage and possibly to its saturation.

2.6.5 Peer-To-Peer (P2P)

In the previous architectures, the game world processing is only done on the server side. MMOGs based on the P2P architecture (see Figure 2.11) delegate work to clients since these clients have available computing resources [115, 120, 93].

This architecture has several advantages, namely [115]:

- *Reduced cost*: Game running cost is reduced because network, bandwidth, storage space and computing power is distributed by all clients.
- *Automatic scalability and load flexibility*: Network capacity increases whenever a new client joins the game network.
- *Fault tolerance*: Even if one peer node fails the network continues with its normal operation.

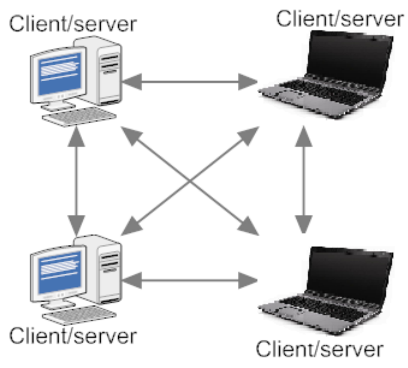


Figure 2.11: Peer-To-Peer (P2P) Architecture.

However, this architecture has several important disadvantages, namely [121, 92]:

- *Game State Distribution*: Game state is distributed among all peer nodes of the network, i.e., there is not a single peer entity that has the whole game state. This is important because players in a game level/map must all have the same state, i.e., one player cannot have a different state. Thus, it is necessary to make sure that the game state of all peer nodes is synchronized.

- *Distribution of updates and patches is troublesome*: Since there is no single entity that controls the game, it is difficult to ensure which peers have the latest updates and which ones do not.

- *Bandwidth*: Bandwidth limitation for MMOGs because users need to send/receive all updates from all the other users and they usually do not have the necessary high end connections.

- *Security*: Harder detection of cheating, since there is no governing authority. An application that is capable of solving the cheating problem is the Referee Anti-Cheat Scheme (RACS), which allows peers to exchange updates directly improving scalability. RACS also has authority over the game state proving good cheat resistance [122].

- *Business*: Harder for companies to explore a P2P game commercially, because not all players are employees of the game company. However, this is only a disadvantage for enterprises.

The P2P architecture offers better scalability than client-server architectures because the adding of a new client peer (player) increases computational power and the storage capacity of the P2P network. However, a standard P2P architecture is not applicable to MMOGs directly. This is so because there are two problems that affect the scalability and performance of MMOGs when they run a P2P network: communication time and load distribution.

In fact, the communication overhead among peers degrades the QoS too much, i.e., the game is not playable in real-time [115, 120, 106, 93]. To achieve lower communication times among peers several solutions were proposed, namely: Donnybrook [71], Voronoi Overlays Networks (VON) [123], N-Trees [124], aggregation techniques [125], and peers@play [126]. Recall that reducing communication time means reducing the game latency.

The second problem has to do with the geographical distribution of players, which is related to the load distribution of peers. It happens that load distribution among peers may be unfair, i.e., some peers may be overloaded while others may be underloaded. In order to cope with this problem, it is often necessary to change the way the P2P network works. Several solutions have been proposed in the literature, namely:

- *P2P-based AOI-Cast*: AOI-Cast is an operation of dissemination of messages to users in the area of interest (AOI). There are two major techniques: VoroCast, which orders AOI neighbors through a Voronoi diagram, where each peer connects to the closest set of peer neighbours; FiboCast dynamically adjusts the range of the message through a Fibonacci sequence, thus allowing peers in the AOI to receive updates more frequently based on their hop counts from the message sender [93].

- *Open Peer-to-peer Network (OPeN)*: This is an architecture developed by Karunasekera et al, where all PCs are peers and there is no central server in charge of interaction coordination. The advantage of this architecture relies in the fact that when peers join/leave the network, game data is easily maintained. [127].

- A CAN-based technique (i.e., structured P2P system based on a d-dimensional coordinate space) approach was used to split the world into disjunctive zones, that were distributed to different nodes of the P2P network [128].

- *Voronoi Diagrams*: This technique was developed for game space partitioning. Voronoi diagrams are a computational geometry technique that allows the partitioning of the game space into regions based on local properties and are supported on a P2P network. The main objective of Voronoi diagrams is to allow each node to construct and maintain a Voronoi diagram based on position information of node neighbours in the node's AOI. Each node connects to the closest neighbor nodes that form the Voronoi diagram [129, 114, 93]. Some applications and techniques examples that use Voronoi diagrams are: Voronoi-Based Adaptive Scalable Transfer (VAST) [130]; Voronoi-based Overlay Network (VON) [123]; Voronoi State Management (VSM) [114]; and Voronoi-Based P2P NVE [129].

- *Peers@play Project*: The peersplay project is a communication engine for P2P-based MMOGs. The main objective of this application is the development of a layer that establishes and manages a P2P network containing all currently active player computers. However, this application tends to maintain a consistent game world, i.e., unlike the pure P2P architecture, the peerplay project architecture does not partition the P2P network, enhancing this way the consistency of the overall game state [126].

- *Donnybrook*: It is a system that allows epic-scalable games without dedicated server resources, even for games with tight latency bounds. It allows better scalability because it reduces bandwidth usage by estimating what players are interested in, thus reducing the frequency at which state updates are sent. It also surpasses resource and interest heterogeneity through dissemination of updates via a multicast system which is designed for special requirements of games, such as games that have multiple sources, games that need good latency, and games which have frequent group membership changes [71].

- *Game State Distribution to Clients*: This technique consists in pushing the cost of running a game to the client, i.e., the clients will be in charge of running parts of the game [109, 95]. An example of an

- *Distributed Hash Tables (DHT)*: DHT are a distributed data placement and lookup algorithm for P2P networks [108, 124]. An example of an application that uses DHT is the *zoned federation model*. This model is based on DHT and on a zoning layer, and was used to construct a cluster of servers on DHT [108]. Another example of an application is Pastry, which is based on an overlay network that uses DHT with support for persistent object storage and event distribution [131].

- *P2P overlays*: Here players who participate in the game form an overlay in which many of the game functions are implemented, i.e., players "offer" their resources (memory, CPU and bandwidth) in order to manage a shared game state [92, 131].

- *Hydra*: Hydra is an architecture that supports a new improved server-client programming model with a protocol that guarantees consistency in the committed messages when peer nodes fail [106].

- *Mediator*: Mediator is an architecture that uses a super-peer network with multiple super-peer roles [27]. In this framework, game server functionalities are distributed using the potential of P2P networks. This enables a MMOG to achieve a better communication and computation scalability.

- *Agents-based Modeling*: This is a architecture that was developed for P2P MMOGs, which is an hybrid solution that starts up as an client-master-server architecture, and then transforms itself into a P2P topology as the number of players increases [120].

Another problem with P2P architectures is the detection and prevention of cheating. This is so because it is difficult to determine which client-peer is cheating. As said above, this problem can be surpassed using a cheating detection and prevention mechanism, named RACS, which was specially designed for P2P networks [122].

However, a pure P2P architecture is not enough to support an application such as a MMOG, because it is necessary to synchronize all the peer nodes, which is a rather difficult task. In order to cope with this problem the pure P2P architecture was improved, leading to the creation and development of hybrid P2P architectures. Two hybrid P2P architectures are found in the literature: mirrored-server and peer clustering, which are described further ahead [91, 132]). Hybrid architectures managed successfully to improve MMOGs scalability, and at the same time to reduce the overall latency, as it is the case of FreeMMG [133].

A MMOG that uses an hybrid P2P architecture was developed by Douglas et al. [115, 127]. This MMOG uses a new P2P middleware that allows the development of complex applications, as it is the case of MMOGs. With this middleware, all machines are peer nodes and there is no central server. The major problem of this middleware is the efficient entity maintenance and interactions in the P2P network. Two possible approaches were taken into account by Douglas et al. in order to solve this problem [115, 127]. The first approach consists in combining the virtual world and the game logic into a database, and then distributing this among the peers nodes. However, it is difficult to maintain efficient entity interactions between strongly related entities in an indirect manner. The second approach consists in separating the dynamic entities into independent processes. This provides efficient communication and processing between strongly related entities. However, it is difficult to maintain the global connectivity. The second approach can be quickly adjusted and used in MMOGs. However, the first approach facilitates the discovery and querying of relevant dynamic entities. Thus, the goal of Douglas et al. [115, 127] was to combine the two techniques in order to have a MMOG working on a P2P network, thus offering a distributed spatial data management to easily discover and query dynamic entities, and combined with the second technique, facilitating real-time interactions [115, 127].

2.6.6 Mirrored-Server

An example of a game that was created with the mirrored-server architecture is a fully operational, distributed Quake system, together with a working synchronization mechanism and a functional multicast protocol, developed by Cronin et al. [91].

The main ideas behind the mirrored-server architecture (see Figure 2.12) are to improve communication on the servers side, to improve fault tolerance by means of data replication, and to reduce cheating by having a centralized server control [91, 132].

In the mirrored-server architecture each peer may be either client or a mirror (i.e., a master or a server as in the master-server architecture) simultaneously. Clients only connect to mirrors and their resources are not harvested to game processing. Thus, from the client point of view this is a client-server architecture. Each client either connects to its closest mirror automatically or chooses one from a mirrors list. Mirrors are connected among themselves through a P2P network. Each mirror possesses a copy of each other's game state, and is in charge of processing a group of players. If a mirror crashes all its associated clients are disconnected, however their data is replicated among other mirrors. Thus, a client game state is persistent [91, 132].

A similar architecture that uses functions of both client-server and P2P architectures

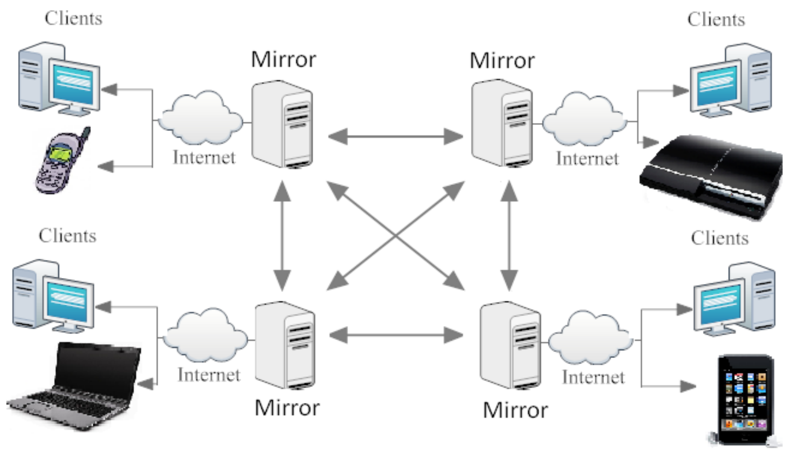


Figure 2.12: Mirrored-Server Architecture.

is the Mirrored Arbiter Architecture [134]. This architecture differs from the mirrored-server in the fact that it has arbitrary agents (i.e., central arbiters). Central arbiters are like the masters in the master-server or slave architecture, i.e., they are in charge of a group of peers and of redirecting clients among themselves when, for example, they leave to another game level. Thus, peers "controlled" by different agents do not communicate directly, which leads to a decrease bandwidth usage and in turn leads to a better communication time.

The main problem with this architecture is that there are many copies of the game state, thus it is difficult to maintain a consistent game state for all players. In order to solve this problem, a synchronization mechanism was developed to ensure that all peers are synchronized. This mechanism makes all peer game states to be synchronized with each other for the sake of the overall state consistency [91].

2.6.7 Peer-Clustering

In the peer-clustering architecture, clients connect to a cluster as in the server-clustering architecture [91, 132], but the clusters in this architecture are connected through a P2P network (see Figure 2.13). The great asset of this architecture is its ability to harvest the clients resources for game processing. If the number of clients associated to a cluster gather enough resources to perform the cluster's job, they will be grouped into a peer network (i.e., a group of clients connected through a P2P network). In this case, the cluster delegates game control processing to the peers and receives updates from them. Peers receive jobs from the cluster and provide updates to all its peer members and to the cluster. When the peers resources are insufficient to handle game processing, the cluster takes over the peer's job [91, 132].

In this architecture, the master-server essentially performs master's work. Unlike the server-clustering architecture, where clusters work individually (i.e., they only process specific clients), in the peer-clustering architecture they cooperate within a P2P network [91, 132].

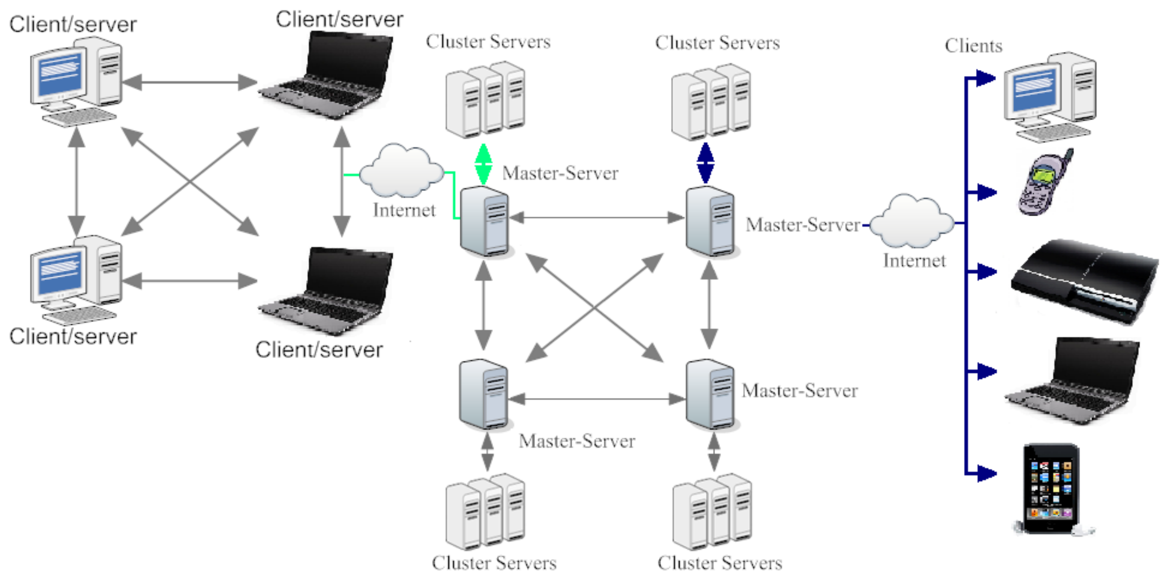


Figure 2.13: Peer Clustering Architecture.

2.6.8 Comparison of Networking Architectures

To better evaluate and compare the previously overviewed architectures, we present the advantages and disadvantages of each architecture [91, 116, 103, 92, 121, 115, 109, 113, 112, 101, 88, 114, 93], as shown in Table 2.1, where C-S stands for the client-server architecture, C-M-S stands for the Client-Master-server architecture, S-C stands for the Server-Clustering architecture, C-M-SL stands for the Client-Master-Slave architecture, P2P stands for the pure P2P architecture, M-S stands for the Mirrored-Server architecture and P-C stands for the Peer-Clustering architecture.

Table 2.1 shows that the client-server architecture is the easiest to design and implement. This is so because the client-server only has the client and server entities. On the contrary, in order to program a client-master-server it is necessary to program a machine that will be in charge of redirecting the clients to active game servers (i.e., this involves some knowledge on network programming). In the client-master-slave and server-clustering, the programming becomes even more intensive, since it is necessary to program a master machine and to program the slaves (i.e., this involves a significant expertise in network programming). Pure P2P and hybrid architectures are even more complex, because besides the needed networking knowledge for programming a P2P or an hybrid architecture, there are also the concerns of architecture security, latency and scalability.

	C-S	C-M-S	C-M-SL	S-C	P2P	M-S	P-C
Easy to design and implement	Yes	No	No	No	No	No	No
Global game processing control entity	Yes	No	No	No	No	No	No
Easy to update	Yes	Yes	Yes	Yes	No	Yes	No
Single game state	Yes	Yes	Yes	Yes	No	Yes	No
Global game access control entity	Yes	Yes	Yes	Yes	No	Yes	No
World Processing Load balancing	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Server Load balancing	No	No	Yes	Yes	Yes	No	Yes
Fault tolerance (Master/Slave fails)	No	Yes	Yes	Yes	Yes	Yes	Yes
Optimized server side resource usage	No	No	Yes	Yes	Yes	Yes	Yes
Low software/hardware cost	Yes	Yes	Yes	No	Yes	Yes	No
High bandwidth usage	No	No	Yes	Yes	Yes	No	Yes
Security	Yes	Yes	Yes	Yes	No	Yes	Yes

Table 2.1: Comparison of networking architectures.

The client-server architecture is the only architecture that has a single entity (i.e., server) with complete control over the game access, whereas in the other architectures we can have multiple masters redirecting the clients or multiple slaves controlling the same game world.

The pure P2P architecture is the only architecture which does not have a single game state, this is due to the fact that each client will have a different state of the game, thus it is difficult to know which client has the latest game state. In order to solve this problem, the hybrid architectures have a centralized control entity for game state synchronizing. In the client-master-server architecture, since each server is independent, each will have its own game state. In the other architectures (i.e., client-master-slave and server-clustering), although the game state can be distributed among several slaves, it will be gathered and synchronized by a master node or nodes.

All architectures with the exception of the pure P2P architecture have an entity solely in charge of controlling the access to the game (i.e., the master). In a pure P2P network, multiple clients might control the access over the game, raising security threats. In order to solve this problem, the hybrid architectures have a centralized control that handles game control access.

Every architecture offers world load balancing, i.e., it is allowed game servers to process small parts of the game world (i.e., often called levels).

The client-server and client-master-server architectures are the only ones that do not allow server load balancing, i.e., server load is only carried out by a single server, whereas in the other architectures high loaded servers can use resources of other lower loaded servers.

The client-server and mirrored-server are the only architectures that do not have failure tolerance, i.e., if the server fails clients will no longer be able to play, whereas in the other architectures, another server takes over the crashed server.

The client-server architecture is also the only architecture that does not have a good resource optimization usage, since the server load is not distributed.

The server-clustering and peer-clustering are the only architectures that have the most costly software/hardware. This is due to the fact that very powerful machines (i.e., usually high performance clusters) are needed for the processing, which may be very expensive, whereas in the other architectures a mid-range computer can cope with the processing load at a reduced cost.

Also client-server architecture offers the best bandwidth usage. This is due to the fact that a server only handles its clients processing, whereas in the client-master-server and client-master-slave, bandwidth can be crucial on the master's side, since all communication passes through it. In the server-clustering architecture, bandwidth is crucial on both master's side and slave server side, because all communication must be redirected across machines. In a pure P2P and hybrid architectures bandwidth is also crucial because of the needed number of updates.

In terms of security, with the exception of the pure P2P, all architectures offer good security mechanisms. The pure P2P architecture has high risks of security because a bad intentioned client can perform illegal action to obtain information from other clients. This is very difficult to handle because it is very hard to determine which client is performing an illegal action. In order to solve this problem, the hybrid architectures have a centralized control that handles security issues.

In relation to the latency, it mostly depends on the location of the game servers and the computing model used. It is clear that if a server is far from clients, the latency will probably be high, whereas a closer server will offer a lower latency. On the other hand, a distributed computing model allows for a better work distribution and faster processing than serial computing; therefore the response to a request is faster.

By observing Table 2.1, some conclusions can be drawn. First, server-clustering is no doubt the best current solution, in terms of performance, for MMOGs. However, it is also the most costly in terms of software and hardware maintenance/upgrade costs, namely, server purchases to support more clients. Also, it is the hardest to implement and design, among all the presented architectures, due to the fact that it often requires the programming and configuration of the cluster machines.

2.7 Computing Models

There are two main categories of computing models we may find in the context of MMOGs: serial model and parallel model. Apart from the serial computing model, we find four computing parallel models in the literature, namely:

- *Parallel computing*;
- *Distributed computing*;
- *Cluster computing*;
- *Grid computing*.

2.7.1 Parallel Computing

Parallel computing is characterized by the fact that several calculations are executed simultaneously. Roughly speaking, large problems are divided into several smaller problems, which are then solved in parallel by different processing units. A processing unit is considered as a conventional processor without cores or, in more modern terms, a core, as illustrated in Figure 2.14 [135]. Parallel computers can be classified according to the parallelism level the hardware supports, namely: multi-core and multi-processor computers having multiple processing units within a single machine or, alternatively, clusters, MPPs (Massively Parallel Processors), and grids using several computers to solve a problem.

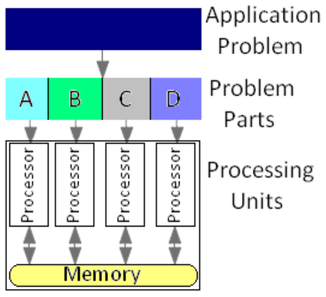


Figure 2.14: Parallel Computing architecture.

An example of a system that uses parallel computing is a supercomputer. A super-computer is a single machine with multiple processing units sharing the same memory, which is capable of doing huge amounts of computations.

2.7.2 Distributed Computing

Distributed computing is a sort of parallel computing that uses several computers. A distributed computer consists of a number of distributed-memory processing units that are connected through a network, i.e., multiple computers interconnected through a network sharing their resources in order to solve a problem in parallel [136]. These machines are not necessarily homogeneous in terms of hardware and computational power. Distributed computing architecture over a LAN is illustrated in Figure 2.15, although it may be extended across a number of networks, including a WAN.

2.7.3 Cluster Computing

Cluster computing is a kind of distributed computing. A cluster is a set of homogeneous computers, usually connected through a LAN or a bus, that work together closely in order to solve a problem. The cluster works as a single logical computer from a client's point of view [102, 117, 137]. Cluster machines are homogeneous for the purpose of guaranteeing an effective load balancing. In cluster computing, a machine plays the

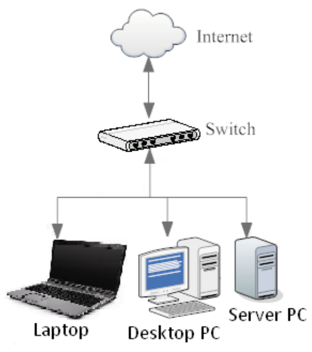


Figure 2.15: Distributed Computing Architecture.

role of a master and the other machines are slaves, i.e., slave machines are connected to the master. Slave machines perform tasks (i.e., a parallelizable instruction or set of parallelizable instructions) delegated from the master and then return the obtained results to the master [102, 117, 137]. This means that cluster computing uses some sort of map/reduce programming mechanism, i.e., the map mechanism for splitting a task across slave nodes and the reduce mechanism to gather the results.

2.7.3.1 Cluster Configurations

Configuring a cluster means defining how tightly nodes are connected to each other. There are three main cluster configurations [138]:

- *Loosely coupled*: In this configuration, slave machines are fully autonomous machines, i.e., each machine has the same version of the operating system and an IP address for accessibility purposes. This means that all the software (i.e., operating system, communication software, cluster middleware, etc.) is installed in the hard disk of each machine by the system administrator manually.

- *Moderately coupled*: In this configuration, slave machines are also accessed by the system administrator directly, but they do not need to have a hard disk because software is located on a network server, which is not necessarily the master. Slaves retrieve needed software from the network server at the booting stage.

- *Tightly coupled*: Similar to the previous configuration, slaves also do not need to have a hard disk and the software installation is also done from a network server.

Slaves do not have a login shell so that the system administrator is not able to access them directly. This configuration is mostly used for latency sensitive applications because the communication time between slaves is low or practically non-existent.

The most used configuration in MMOGs is the tightly coupled (see Figure 2.16), because it provides a small latency for a significant number of users.

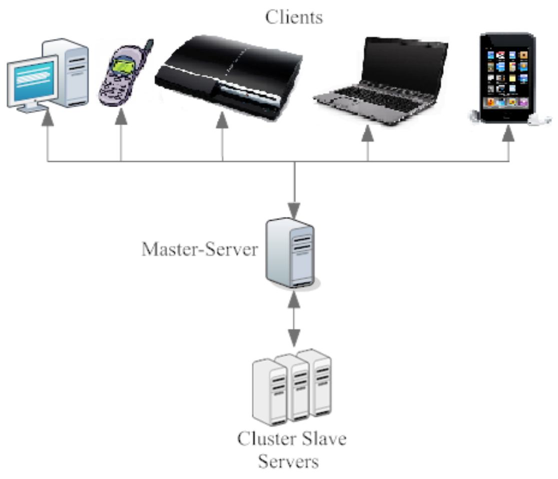


Figure 2.16: Tightly Coupled Cluster Computing Architecture.

2.7.3.2 Cluster Categories

Cluster categories are related to the types of applications, i.e., availability-related applications, database-related applications, or performance-related applications. Thus, there are three categories of clusters, namely:

- *Failover clusters*: Failover clusters, also called *high-availability clusters* (HA), are designed with the objective of improving the availability of cluster services, i.e., clusters with multiple idle nodes that provide the service whenever a node fails. An example of a HA cluster is the Microsoft Cluster Services (MSCS) [118].
- *Load-balancing clusters*: In this case, workload is distributed across several interconnected computers in a balanced manner. Load-balancing clusters prevent the overloading of a single machine as necessary for data-intensive or database applications. An example of a load-balancing cluster is the Oracle RAC which addresses several areas of database management, namely: fault tolerance, load balancing, and scalability [118].

- *Performance clusters*: These clusters, also named *compute clusters*, are primarily designed for compute-intensive applications rather than for handling IO-oriented operations such as web services or databases. Examples of performance clusters are Beowulf clusters [118], Sony PS3 clusters [139], and High-Performance Computing (HPC) clusters [140, 141].

Usually, clusters combine two or all the characteristics of the categories mentioned above, i.e., clusters can combine failover, load-balancing and performance features.

2.7.4 Grid Computing

A grid is a set of geographical distributed computers (usually, over a WAN) that share their resources and processing capacity (see Figure 2.17) [142, 143, 144]. Grid computing is then the most distributed kind of parallel computing because it uses resources of different networked, non-necessarily homogeneous, computers in order to solve a given problem. We may have a grid in a single computer where each processing unit is a node, or have a grid among multiple computers where each computer will run one or more nodes, or also have a grid of clusters where we can have multiple clusters running multiple nodes.

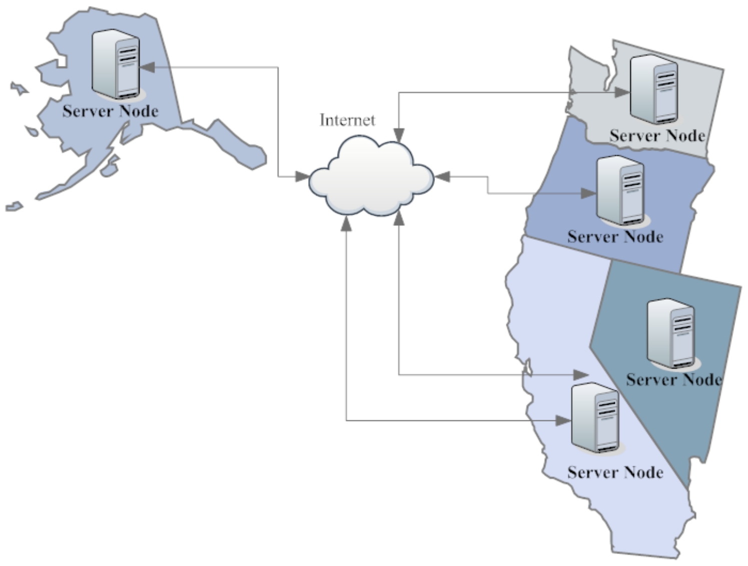


Figure 2.17: Grid Computing Architecture.

In grid computing, communication time between different computers is an important issue when compared to communication time within a supercomputer. Unlike supercomputers, grids are scalable because it is only necessary to add new, but not necessarily homogeneous, network nodes to expand the computing capacity [143, 144].

In grid computing there are two types of nodes: control nodes and work nodes. A grid has several control nodes that ensure task division among work nodes, as well as, the gathering of node results, produced by those work nodes. By definition, a *task* is a set of instructions that is divided into sub-tasks among available (work) grid nodes. These grid nodes run sub-tasks in parallel.

Roughly speaking, and despite other possible grid configurations, a control node sees, and is seen by, all the other control nodes, while a work node only sees the work nodes that are directly connected to its master control node. This is illustrated in Figure 2.18.

1 Servers send tasks to control node for parallelization.

2 Control node divides tasks into sub-tasks.

3 Sub-tasks are sent to available grid nodes.

4 Grid nodes execute sub-tasks and return a result to the control node.

5 Control node gathers results from nodes, and returns a final result to the server.

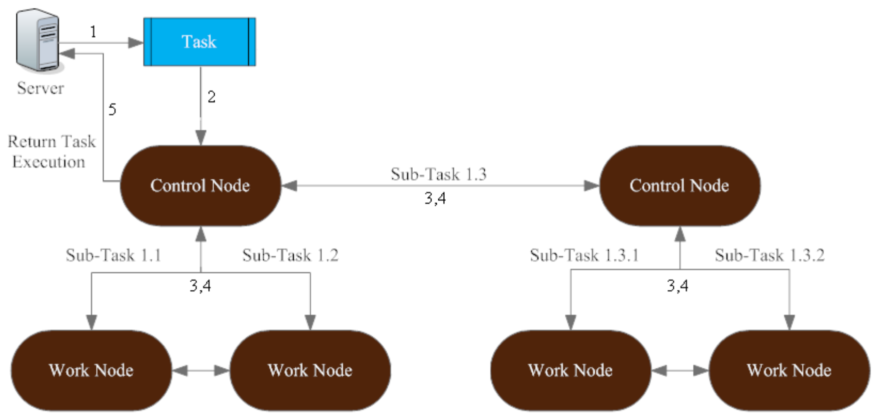


Figure 2.18: Grid networks doing a task division.

The first grid applications were academic-related (e.g., NASA Information Power Grid) [145], but today it is possible to find grid technologies for financial, medical and media applications [143, 144]. In the case of MMOGs, we have identified two grid middlewares: OptimalGrid [15], and RTF/Edutain@Grid [146]. OptimalGrid was used to transform Quake2 into a MMOG running on a grid, while RTF is a framework to develop games and e-learning applications on the Edutain@Grid.

OptimalGrid applies spatial division mechanisms to the whole game world, dividing it into several continuous parts depending on the available existing grid nodes, which involved many source code modifications to both client and server modules (i.e., game world spatial division was not done in Jake2 as it would be necessary to modify a great amount of source code, which was not the objective of this thesis).

2.7.5 Computing Models Comparison

A comparison of the advantages and disadvantages of the previous four computational models described in this chapter is shown in Table 2.2.

	Computing Models			
Features	Parallel Computing	Distributed Computing	Cluster Computing	Grid Computing
Latency	Low	High	Low	High
Scalability	Low	High	High	High
Cost	High	Low	High	Low
Hardware Homogeneity	Yes	No	Yes	No
Management using Middleware	No	No	Yes	Yes

Table 2.2: Comparison of Computing Models.

Let us explain the data in Table 2.2 in respect to the following features:

- *Latency*: Unlike distributed and grid computing, parallel and cluster computing models offer low latency due to the short communication time among nodes.
- *Scalability*: Unlike the other computing models, parallel computing only offers low scalability because we are limited to the hardware power and resources of a single machine.
- *Cost*: Distributed and grid computing are the only computing models that allow the usage of cheap commodity or domestic computers.

- *Hardware Homogeneity*: Parallel and cluster computing require the usage of computers with the same technical specifications (e.g., computing power, software, etc.), whereas the other computing models allow any machine to perform computation.
- *Management using Middleware*: Cluster and grid computing are the only computing models that allow the control and management of nodes through a middleware.

Since MMOGs are latency-sensitive applications and because the number of players is not stable, using cluster computing seems the best solution for the case, as communication time is not significant in a cluster, and because a cluster has enormous processing power, which is capable of handling thousands of players simultaneously. However, cluster computing is not good for companies that have a low budget, because the machines that are required for applications such as MMOGs are very expensive, both in terms of hardware/software purchase and maintainance costs.

Besides, grids were originally thought of as a solution for so many idle computing resources available over networks because any machine, including a user's machine, can become part of the grid and can be used for computing. Thus grid computing offers cheaper hardware/software for low budget companies, because less powerful machines can be used. The major problem of grids is the high latency due to the high communication time among nodes, however, grids can be further programmed to reduce latency among grid nodes either through different networking protocols, networking architectures, or by "cutting/restraining" communication among nodes.

2.8 Summary

In short, the most important problem of MMOGs is the lack of scalability. Scalability problems in MMOGs mainly come from the game's latency and number of players. Latency in MMOGs is affected by the number of players, by the game's architecture and by the computing model that is used because some architectures are better than others and some computing models might be more adequate than others for handling latency issues.

Although there are many solutions that can solve the scalability problem of MMOGs, the most important one is to use computing models such as cluster and grid computing. Grid computing consists in using resources and processing capacities shared by geographical distributed computers in order to solve a problem. These computers form a large single virtual computer, also known as a Grid [142, 143, 144].

Chapter 3

Grid-based Re-engineering of Jake2

This chapter describes the re-engineering process of Jake2 into a grid-based MMOG. We have used GridGain as the middleware for grid computing. This re-engineering process follows a number of software re-design principles that will be approached throughout this chapter.

3.1 Software Re-Engineering

Software re-engineering is the re-designing/re-writing of part or whole the application without changing its functionalities, making the application easier to maintain. Software re-engineering is useful because it is less costly than developing an application from scratch [147]. Software re-engineering involves a number of procedures, namely [147]:

1) *Source code translation*: Translating source code into another programming language. This may be necessary because of hardware platform updates, staff skill shortages, and organizational policy changes.

2) *Reverse engineering*: This procedure consists in discovering the technological principles of a software system by means of the analysis of its structure, functionalities and operation. For example, in this master's work, this procedure involved the analysis of Jake2 source code, i.e., structure and functionality.

3) *Modules re-engineering*: In this procedure, also called program modularization according to Sommerville [147], several related application parts may be grouped together into a single module (e.g., parts related to a client are grouped in a client module), and modules may also be modified in order to either remove unnecessary or add new functionalities. For example, in this master's work, we removed server creation on the client module but added the grid middleware functionality to the server module.

4) *Data re-engineering*: This procedure is used in order to create a manageable data environment, i.e., data structures and values may need to be analyzed and re-organized. For example, in this master's work, we integrated a database in our game for the purpose of storing client's data and persistence.

5) *Control flow re-engineering*: This procedure is called program modularization according to Sommerville [147] and is used to simplify conditions (e.g., while, go to, for, etc.) of an application source code in order to improve their functionality and reading. For example, in this master's work, we parallelized 'for' instructions on the server side in order to obtain a gain of scalability.

Since Jake2 and GridGain were already encoded in Java, there was no need to undertake the first procedure. Thus, the work done in this thesis involved procedures 2 through 5.

3.2 Grid Middleware: A Brief Analysis

In the re-engineering process of Jake2 we had to take into account the features of the available grid middlewares. A grid middleware is a software that allows the sharing of available/idle heterogeneous computing resources over a network.

Usually, a grid middleware needs to be installed in every single machine manually and configured accordingly, i.e., on the operating system. A grid middleware allows for complete control over the grid nodes.

Features	Advanced Resource Connector	GridGain	Globus Toolkit	ICE	ProActive Parallel Suite	Sun Grid Engine
Security	YES	YES	YES	YES	YES	-
Jobs brokering	YES	YES	YES	YES	YES	YES
Resource Management	YES	YES	YES	YES	YES	YES
Services Oriented Architecture	YES	YES	YES	YES	YES	YES
Threads	-	YES	YES	YES	YES	YES
Open Source	YES	YES	YES	YES	YES	-
Language	-	Java	C Java	C++ Java	C/C++ Java	-
Operating System	Lin	Lin Win Mac	Lin Mac	Lin Win Mac	Lin Win Mac	Lin Win Mac
Learning Curve System	Slow	Fast Mac	Fast	Slow Mac	Slow Mac	Slow Mac

Table 3.1: Grid middlewares.

There are several grid middlewares as shown in Table 3.1. These middlewares were compared to each other using the following principal features:

- *Security*: It is recommended that the middleware possesses some security mechanisms in order to, for example, prevent hacks.

- *Job brokering*: This is the an essential feature in every single grid middleware, and has to do with the ability of dividing a task into jobs.

- *Resource Management*: The middleware has much to do with resource management across a network of nodes, i.e., access to resources, control over resources.

- *Services Oriented Architecture*: With the development of web technologies, most grid middlewares now incorporate web services (e.g., XML, SOAP, WS).

- *Threads*: Most middlewares allow for the usage of threads. For example, in GridGain a task can execute multiple jobs, executing them like a set of multi-threaded tasks.

- *Open Source*: Most middlewares go public as open source codes.

We have also considered the following secondary features:

- *Language*: Native programming language or most used programming languages supported (e.g., C, C++, Java) .

- *Operating System*: Supported operating systems (e.g., Windows, Linux, Macintosh).

- *Learning Curve*: How long does it take to learn how to use the middleware.

A complete grid middleware analysis is shown in Table 3.1. The '-' symbol means that information about the feature was either not provided or could not be found. As shown in Table 3.1, four grid middlewares offer all the principal features: GridGain, Globus Toolkit, Internet Communications Engine (ICE) and ProActive Parallel Suite. Unfortunately, the Globus Toolkit is not a fully cross-platform middleware, and this means that no Windows machine can make part of any grid. Remember that, Windows is one of the most important platforms in games. On the other hand, both ProActive and ICE take much more time to learn programming than GridGain. So, we have decided to use GridGain in this master work.

3.3 GridGain

GridGain, as mentioned previously, is a grid middleware written in Java that is easy to use in the development or integration of application using grid computing.

3.3.1 GridGain Architecture

GridGain architecture is shown in Figure 3.1. As shown in Figure 3.1, GridGain has 3 main layers:

- *Grid Public API*: This API is used by the user to execute tasks and retrieve grid information (e.g., topology or events), send direct messages to nodes, etc..
- *Grid Kernal/Kernel*: The Kernel is responsible for providing the implementation of GridGain public API and function delegation to the service provider interface (SPI) layer.
- *Service Provider Interface (SPI)*: A set of Java interfaces that provide implementations for node discovery, communication, topology resolution, job checkpoints, scheduling and job collision resolution, load balancing, fail-over, etc..

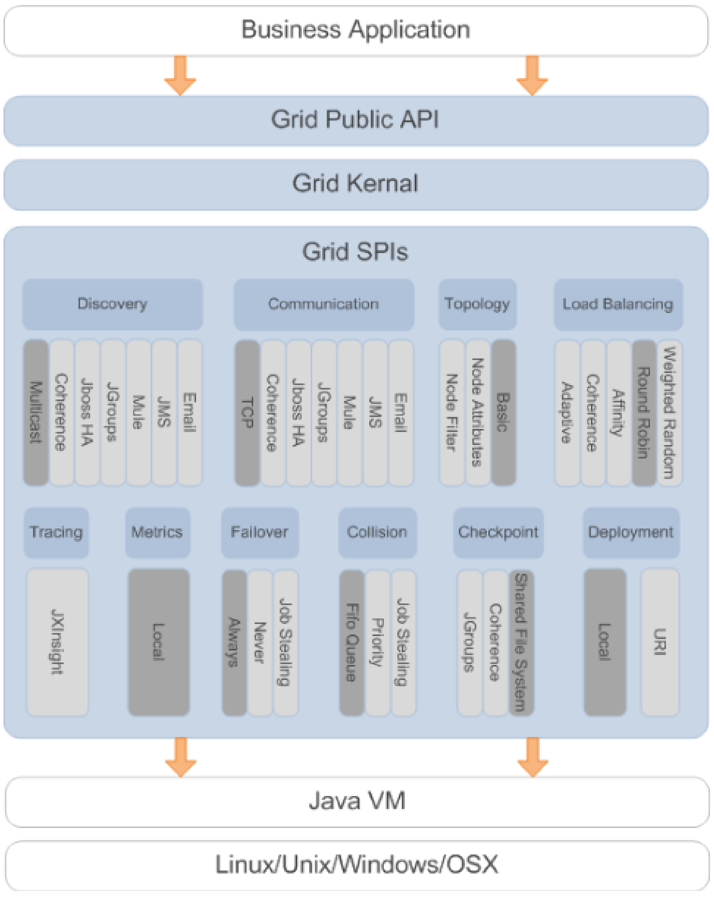


Figure 3.1: GridGain Architecture (taken from GridGain web site at http://www.gridgainsystems.com/wiki/display/GG15UG/GridGain+Architecture)

.

GridGain has the following distinctive features that are related to its architecture (taken from http://www.gridgain.com/product_features.html):

- *Hybrid Cloud Deployment*: An application is developed locally and cloud computing is used to run the whole application, some parts or under certain conditions.

- *Cloud Aware Communication and Deployment*: GridGain runtime can be adjusted to different environments without changing any business logic that is currently running on it.

- *Advanced Affinity Map/Reduce*: The map/reduce mechanism divides a task into several sub-tasks (also called jobs), executes sub-tasks in parallel and gathers results into one final result.

- *Annotation-based grid-enabling with AOP*: GridGain is the only grid computing infrastructure that permits to grid-enable existing code without any modifications.

- *SPI-based integration and customization*: Service Provider Interface (SPI)-based architecture forms the heart of the configuration and customization capabilities of the GridGain middleware. As mentioned previously, SPI allows programmers to customize almost every aspect of GridGain functionalities.

- *Advanced load balancing and scheduling*: GridGain offers early and late load balancing. These processes are defined through load balancing and scheduling resolution SPIs.

- *Pluggable Fault-Tolerance*: GridGain offers a totally pluggable failover logic, i.e, it permits to failover the logic and not only the data.

- *One compute grid - many data grids*: GridGain allows the user to freely choose which caching/data grid solution he/she wants to use with the GridGain compute grid infrastructure.

- *Zero deployment model*: GridGain allows the execution of multiple nodes on the same machine or on the same virtual machine without any changes to the configuration.

- *JMX-based management and monitoring*: GridGain offers a huge collection of JMX MBeans, which allow the user to see the monitoring and statistical information about all grid nodes.

3.3.2 Split/Reduce Mechanism

GridGain computing mainly is structured in tasks and jobs. A task is a main deployment unit in GridGain, i.e., a task is what a user deploys (explicitly or implicitly) for execution. A task is split into jobs whose results are aggregated and returned to the calling task. A job defines a running piece of code, i.e., basically a job travels to remote nodes for execution. The combination of both tasks and jobs is the essence of GridGain and computational grid computing in general. GridGain offers two task division mechanisms: split-reduce and map-reduce. The split-reduce division mechanism divides a task into equal sub-tasks among grid nodes, regardless of which grid node is going to perform the sub-tasks. The problem with this mechanism is that it assumes that each node has the same work capacity and each node must do a sub-task, which can result in nodes repeating sub-tasks. Split-reduce mechanism is illustrated in Figure 3.2

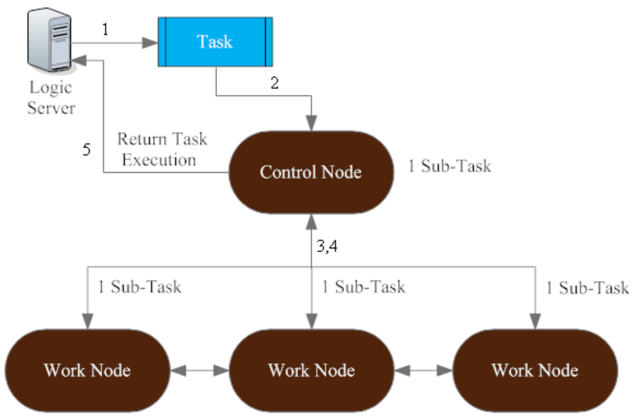


Figure 3.2: Split-reduce division mechanism.

Split-reduce works in the following manner:

1) Parallelizable tasks are chosen from the server side of Jake2.

2) Tasks are assigned to a control node.

3),4) Control node divides the incoming tasks into jobs, assigning these jobs to available grid nodes and gathers results from grid nodes.

5) Control node sends updates back to server.

Split-reduce division mechanism may waste resources by doing repeated sub-tasks, thus it is not good as these resources could be used for other tasks.

3.3.3 Map/Reduce Mechanism

Map-reduce division mechanism divides tasks into different sub-tasks among the grid nodes, by assigning one or more sub-tasks to the grid node that has more resources or to a specific node or nodes according to a pre-defined condition set by the programmer. Map-reduce division mechanism was the chosen mechanism for Jake2Grid because it takes advantage of available resources and it uses resources as they become available. Map-reduce division mechanism is illustrated in Figure 3.3. The map-reduce division mechanism follows the same working steps as split/reduce with a variation in step 3, where one or several specific sub-tasks are sent to a specific node.

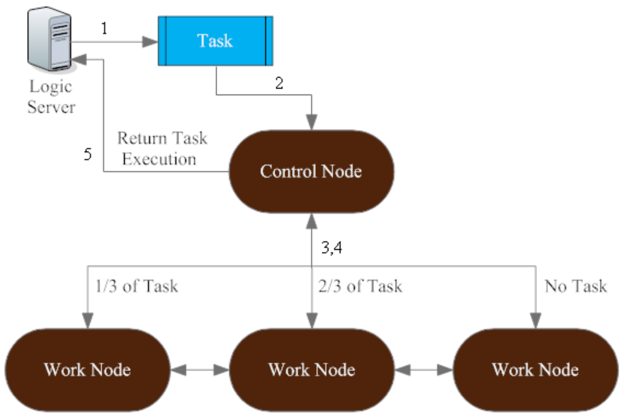


Figure 3.3: Map-reduce division mechanism.

GridGain allows the distribution of jobs according to the resources available on each node through a function called balancer. For example, let us assume that we have 2 jobs (A and B) and 2 nodes (1 and 2). Node 1 has the most resources when job A is sent to the grid. When job B is sent to the grid, if node 1 has finished its job, then any of the nodes can do job B. This happens if the map/reduce mechanism is being used and if balancer discovery is activated.

3.3.4 Network Protocols

It is also important to discuss what type of network protocol GridGain uses and how to use it appropriately. GridGain offers several networking protocols, namely:

- *TCP*: The default communication SPI which uses TCP/IP protocol through multicast to communicate with other nodes. Messages to multiple nodes are sent concurrently.

- *JGroups*: JGroups is an open source product that provides reliable UDP/multicast implementation among other things, i.e., it uses the UDP network protocol for node communication. JGroups is usually used to establish a connection between two geographically independent grids, by transforming them into one huge distributed grid (e.g., a grid constituted by two different LAN grids.).

- *Coherence*: Oracle coherence implementation of GridCommunicationSpi. Oracle coherence offers replicated and distributed data management and caching services on top of a reliable, highly scalable peer-to-peer clustering protocol. GridGain uses coherence for remote node communication.

- *Java Messaging Service (JMS)*: This implementation uses JMS topic and queue to send messages to an individual node or to a group of remote nodes, i.e., it transforms a group of LAN nodes into a local grid, reducing the communication cost between nodes.

- *Mail*: Email communication is provided for cases where nodes from different networks (e.g., even different countries) need to participate in grid task execution together as it can usually penetrate through any firewall. It supports SMTP/POP and IMAP email access protocols and can be used to connect to any public or private email server out there.

- *Mule*: It is P2P-based network protocol used in GridGain for node discovery and communication. A Mule instance may be started before SPI or instantiated during SPI start.

In order to use any of these network protocols we need to know when to use each protocol and which type of protocol suits better to the application. In the case of MMOGs, the best protocols to be used are TCP, JGroups, JMS and Mail. We used the TCP protocol in some experiments because it is the default protocol of the GridGain, but we ended up using JMS because it reduces the communication time among nodes significantly. Although JGroups and Mail would possibly provide a better communication speed among nodes, since they use the UDP protocol, they are most suited for joining multiple networks into one major grid. In our case, nodes are on the same LAN, so we had no need to test either JGroups or Mail protocols.

3.3.5 GridGain Middleware Installation and Functioning

It is also important to understand how GridGain works once it is installed. GridGain middleware needs to be installed in each machine (i.e., the grid software is installed in the machines that the programmer wishes to become part of the grid). After the installation, one or several work nodes are started by simply running the grid software (i.e., they only ensure that tasks are done). A maximum number of 512 work nodes can be started in the same machine. Control nodes can only be started up inside an application, i.e., the programmer needs to add the grid software to his/her application and ensure task division.

The GridGain middleware functions by multicast (i.e., it recognizes all LAN nodes and automatically assigns them a unique grid). Thus, if multicast is blocked, each machine in which the grid middleware is installed becomes a separate grid.

3.4 Reverse engineering of Jake2

After the description of our chosen grid middleware we tackled the second step/procedure of the re-engineering software process. In this step, we analyzed Jake2 in more detail, pointing out the changes made to its source code. Recall that Jake2 is a multiplayer First Person Shooter (FPS) game, i.e., an action game where players kill other players for points. A Jake2 snapshot is shown in Figure 3.4.

Figure 3.4: A Jake2 snapshot.

3.4.1 Jake2 code

Jake2 is a brute force port of the original Quake2 C++ source code to Java. It is divided into eight modules:

1) *Client*: Key handling, visual effects, screen drawing and message parsing.
2) *Game logic*: Monsters, weapons, elevators, explosions, keys and secrets.
3) *Qcommon*: Common support functionality (file system, message buffer, config variables, checksums, Jake2 welcoming/splash screen).
4) *Renderer*: 3D graphics displays (jogl, fastjogl, lwjgl).
5) *Server*: World handling, movement, physics, game control and communication.
6) *Sound*: Sound implementations: joal, lwjgl, and a proprietary native Java sound driver.
7) *Sys*: Methods dealing with the operating system (network, keyboard handling).
8) *Util*: 3D maths, some libc methods, etc.

The transformation process of Jake2 into a MMOG has led to changes to the source code of modules 1), 2), 3), and 5). These changes will be described later on.

3.4.2 Jake2 Playing Modes

When Jake2 starts running, a local client and local server start off. Jake2 has two playing modes:

- *Single player*: The client and (local) server run on the same machine. The client connects to its local server and plays against the computer. This local server is a private server, i.e., other players do not see it and therefore cannot join.
- *Multiplayer*: Normally, the server runs in a remote machine to which clients can connect and play against other players. However, a client can create a logic server that other players may join. A logic server is the client's local server, however, on the contrary to the single player mode, the local server is now visible to other players.

Jake2 diagram of playing modes is shown in Figure 3.5. These playing modes work in the following manner:

- A client chooses one of the two playing modes: single player or multiplayer.
- If client enters single player mode, he/she plays on his/her own local server against the computer.
- If client enters multiplayer mode, he/she can create a logic server or join a remote server and play against other players.
- The client may switch between local server and remote server.
- Each server, either local or remote, processes data of its connected clients.

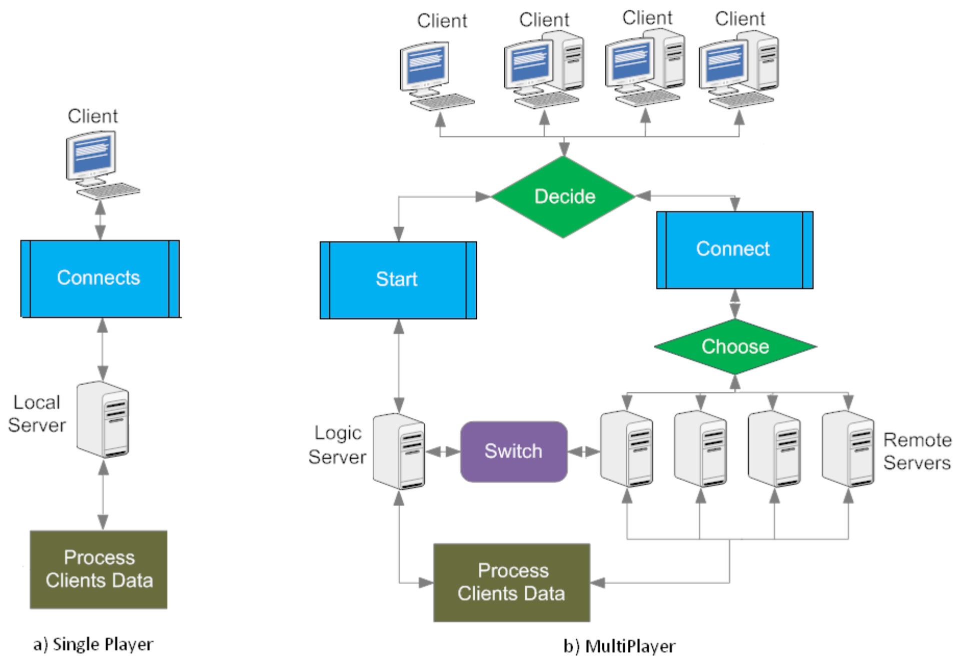


Figure 3.5: Jake2 playing modes: (a) single player mode; (b) multiplayer mode.

3.4.3 Jake2 Networking

Similar to Quake2, Jake2 is also a client-server networking game. If Jake2 is running on a LAN, the list of game servers is directly provided to the client (Figure 3.6). But, if Jake2 is running on a WAN, it is necessary to use a master server in order to provide such a list of game servers to the client (Figure 3.7).

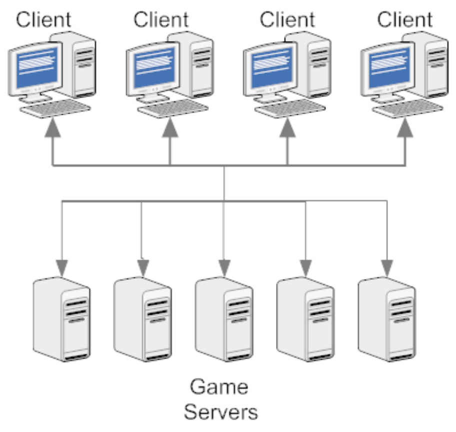


Figure 3.6: Jake2 networking architecture on a LAN.

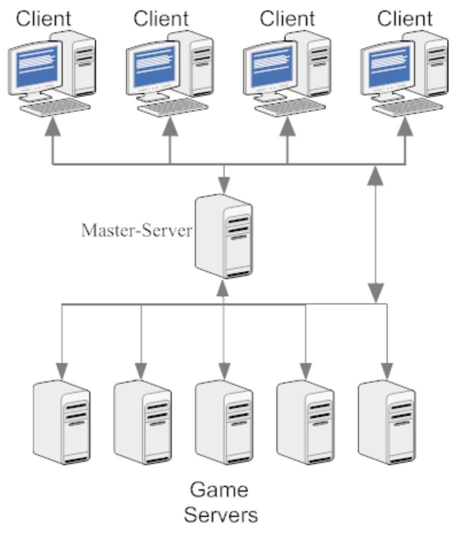


Figure 3.7: Jake2 networking architecture with master-server in a WAN.

It does not matter whether a master server is being used or not to provide the list of existing game servers to clients, once a client has connected to a game server and while he/she is playing on that server, his/her communication will no longer need to pass through the master, i.e., the architecture will be a traditional client-server architecture where client communication is sent directly to game servers.

3.4.4 Jake2 Persistence

Persistence has to do with keeping the game world state, as well as the state of each player. Persistence exists in the single player mode of Jake2, but not in the multiplayer mode. This lack of persistence is due to:

- *Nonexistence of game world state*: In multiplayer mode, the state of the game is not altered at all by any player action. For example, a bomb blast does not cause any damage to the scenery (i.e, bullets do not affect walls).

- *Nonexistence of player state*: In multiplayer mode, player state only persists as long as the player remains in a game level. This state is lost when the player quits the game level or, alternatively, when the game level terminates.

3.4.5 Jake2 Game World

Jake2 uses the zoning technique to divide the game world into game levels, also called sub-worlds or maps. Jake2 possesses a total of 39 game levels. Each game level allows up to a maximum of 32 players playing simultaneously.

As explained in Chapter 4, the limitation concerning the maximum number of players has to do with the need of providing quality of service (QoS) to the clients. Each server handles a single game level at a time, having to be restarted whenever such a game level ends. Each server hosts a game level using one of the following configurations:

- *Time configuration*: The game server runs a game level either indefinitely or for a limited period of time. When this limit is reached, the game level terminates and the game server either starts a new game level, or re-starts the same game level, or terminates execution, depending on the user definitions upon game server creation.

- *Kill configuration*: The game server runs a game level either indefinitely or for a limited number of player kills. When this limit is reached, the game level terminates and the game server either starts a new game level, or re-starts the same game level, or terminates execution, depending on user definitions upon game server creation.

In addition, these two configurations can be combined into an hybrid configuration.

3.4.6 Jake2 Scalability

Jake2 was designed to run over a LAN. According to its developers, in order to keep the game's QoS over such a LAN, the recommended maximum number of players was

256. Recall that each Jake2 game level allows up to 32 players and according to the recommended number of players, there can only be 8 game servers (i.e., $8 \times 32 = 256$), in order to keep the game's QoS. Although Jake2 supports all these players, the game's QoS may not be enough to handle these many players. To solve this problem, Jake2 was modified in order to allow game server load distribution, thus resulting in a new scalable game, Jake2Grid.

3.5 Module Re-engineering of Jake2

This is the third step of the re-engineering software process. In this section, we focus on analyzing and describing Jake2Grid architecture in order to identify the modules or parts of them, where re-engineering will take place. In particular, we interested in to identify possible module changes that will allow us to improve the scalability of the game. The scalability solution adopted in this dissertation is based on grid computing. Grid computing distributes processing tasks over a grid of computers, or nodes that are neither necessarily within the same LAN nor geographically close.

Jake2Grid is the re-designed game using grid computing. From all the eight modules mentioned in section 3.4.1, we only changed four modules: client, game logic, qcommon and server. Our solution is then running a grid on a set of machine servers, as illustrated in Figure 3.8. Our architecture has seven main modules:

- *Client*: This module is in charge of providing the graphical interface for clients.
- *Qcommon*: Clients access this module when their application terminates.
- *Proxy*: Here, one machine manages the database containing player information. Players must first register and identify to play the game.
- *Server*: Here, Jake2 game levels are handled and clients data is processed.
- *Game logic*: This module is accessed by the server to retrieve client data.
- *GridGain*: Here, game server load is distributed among grid nodes. For each game server, there is a control node responsible for load distribution. This layer is transparent to a user, i.e., he/she does not know the game is running on a grid.
- *Database*: This module is in charge of the persistence of the game, i.e, keeping all the players states.

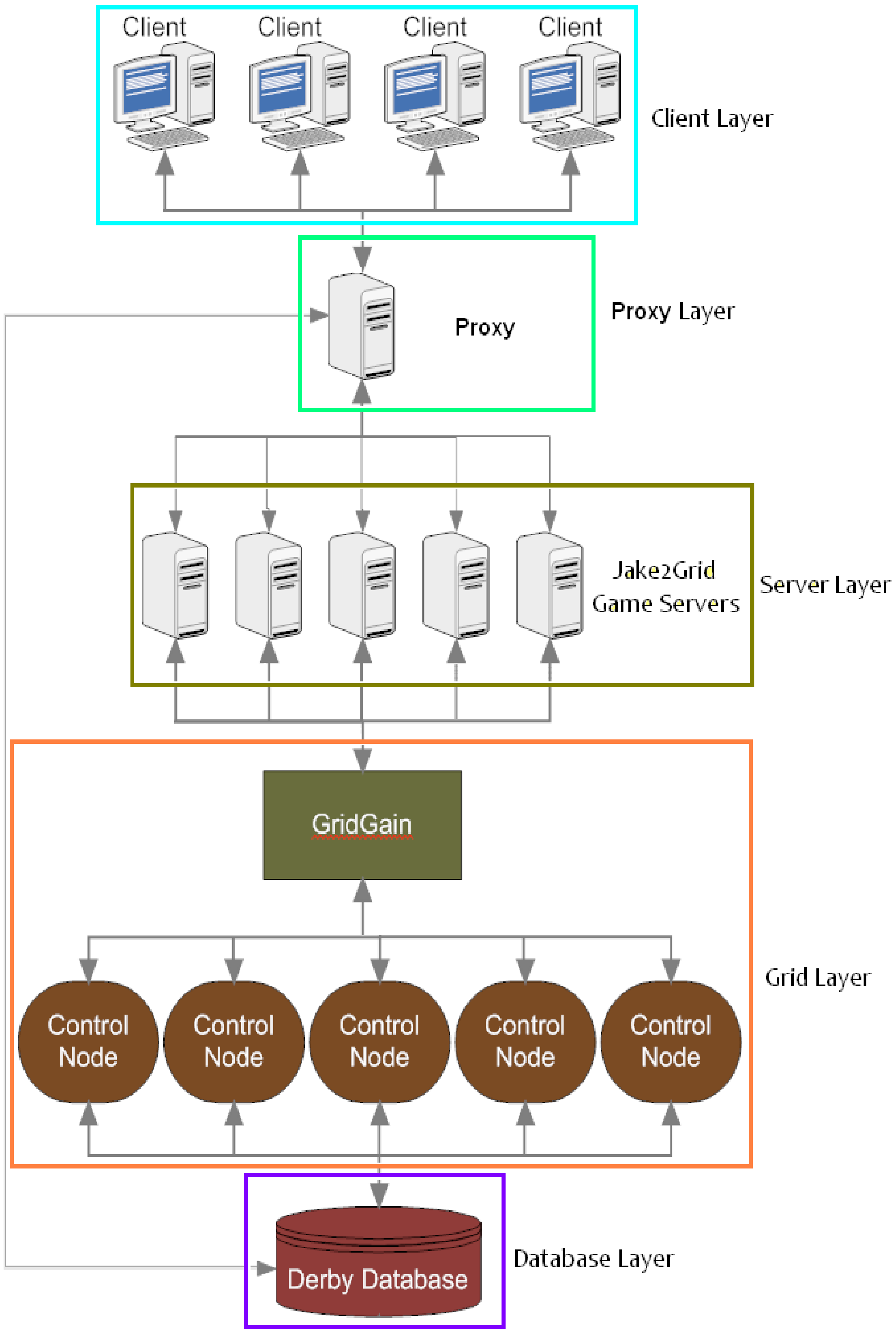


Figure 3.8: Jake2Grid architecture.

The module re-engineering process was applied to the Client, Qcommon and Proxy modules.

Algorithm 1 Original Jake2 Main Class

```
Jake2 {
for $n = 0$ to number of arguments do        ▷ If clients want to start a game server
    if argument at $n$ matches "+set" then
        if next $n \geq$ number of arguments then
            end cycle
        end if
        if argument at $n$ not matches "dedicated" then
            end cycle
        end if
        if next $n \geq$ number of arguments then
            end cycle
        end if
        if argument at $n$ is "1" or argument at $n$ is "\"1\"" then
            user inputed dedicated
        end if
    end if
end for
if $dedicated = true$ then        ▷ If dedicated mode is selected
    A game server is going to start
end if
        ▷ If dedicated mode was not selected, start the graphical client's interface
if $dedicated \neq true$ then
    Start the client's interface so that a client can be started
end if
Depending on the arguments either start a client or a game server }
```

3.5.1 Client Module Changes

A Jake2Grid client has the same behavior as a Jake2 client, i.e., they both join servers and kill other players for points. However, this module was modified in Jake2Grid in order to prevent clients from creating game servers. Jake2 game source code has a special initialization class, called Jake2. This class allows the loading of the client graphical interface or the creation of a game server, by passing an argument through the command line. This is shown in Algorithm 1 (implemented in the original Jake2.java), and deserves a few comments:

- Line 2-17: Checks client's arguments to whether start a client or a server;
- Line 18-25: Client starts a graphical interface or a game server.

If "+set dedicated 1" is passed as an argument, a game server will be started; otherwise, a client graphical interface will be started. This was eliminated on Jake2Grid client, i.e., lines 2-22 and line 24 were eliminated in the Jake2Grid Client version, as shown in Algorithm 2 (implemented in Jake2.java of Jake2Grid Client source code).

Algorithm 2 Jake2Grid Client Main Class

```
1: Jake2 {
2: Start the client's interface
3: Start the client application }
```

This algorithm is explained below:

- Line 2-3: Starts a graphical client's interface and initializes the client application.

In Jake2Grid, the client cannot start a game server. Since Jake2 is a client-server networking game, some work (i.e., rendering and game data reception and delivery) is done on the client's side. Thus smaller changes were made on the client:

1) *Registration*: Clients now need to register/identify themselves in order to play.
2) *Deactivation of single player game option*: Clients play only in multiplayer.
3) *Modification of multiplayer menu*: Clients no longer create servers.
4) *Modification of server menu*: List of servers is now provided.

The main change done to the client was the adding of a registration/identification form for players, i.e., the clients must now register and sign in in order to play the game. Jake2Grid requires the clients sign in to play. Therefore, the client graphical interface was changed in order to add a client's login/registration form, which must be filled by clients if they wish to play Jake2Grid. This is shown in Algorithm 3 (implemented in Menu.Java of Jake2Grid Client source code):

Algorithm 3 Jake2Grid Client: Client module

```
1:  Multiplayer Menu Method {
2:  Added new form {                                             ▷ Login form
3:  "Nick" field;
4:  "Password" field;
5:  "Login" action {
6:  if Nick  Password = empty  then          ▷ Verify if clients have filled both fields
7:      return error                    ▷ If any field is empty, clients must fill new form
8:  else
9:      Create connection to the database
10:     if Player does not exists in database then
11:         return error             ▷ If player does not exists, clients must first register
12:     else
13:         if Password is wrong then
14:             return error      ▷ If player inputs wrong password, clients must fill new form
15:         else
16:             if Player is already connected then
17:                 return error        ▷ If player is connected, clients must fill new form
18:             else
19:                 Update clients registry data on the database and let them proceed to server selection
20:             end if
21:         end if
22:     end if
23:     Terminate connection to the database
24: end if}
```

(Cont. ...)

```
25: Added new "Register" action {
26: Proceed to Registration Menu } }
27: }
28: Register_Menu Method {
29: Added new form {                                          ▷ Register form
30: "Nick" field;
31: "Password" field;
32: "Register" action {
33: if Nick  Password are empty then          ▷ Verify if clients have filled both fields
34:     return error                    ▷ If any field is empty, clients must fill new form
35: else
36:     Create connection to the database
37:     if Player exists in database then
38:         return error            ▷ If player already exists, clients must fill new form
39:     else
40:         Add player register data to database          ▷ If player does not exists, add
    player data to database
41:     end if
42:     Terminate connection to the database
43: end if} }
44: }
```

This algorithm is explained below:

- Line 3-27 (Login Form): Registered players fill in the nick and password fields and sign in order to play the game.

- Line 25-27 (Register action): Unregistered players must first register or they cannot sign in the game.

- Line 28-44 (Register Form): Unregistered players fill up the nick and password fields, thus registering themselves in order to sign in the game and play.

When clients register all their information (netname, password, connected state, client port) is stored in the players registration table of the Derby database, located on the server side.

Passwords are used in order to grant access to player accounts. Connected state is used in order to prevent a player with a same netname from sign in twice. Client port is used in order to allow players from a same LAN to play the game, i.e., in the original Jake2 all players would use the same port (i.e., port 27901) and players from a same LAN would have the same exit IP address (i.e., the LAN IP address). This means that the game server would not allow players from that LAN to join the game, because it could not differentiate the players. In order to cope with this problem, a unique randomly generated port would be assigned to each client upon registration, thus if multiple clients from the same LAN would connect to the server they would have a unique port in order for the server to differentiate them.

When the client signs in the game, his/her personal data are checked and updated in the database. To cope with registration, a new class called PlayerRegister was added to the game logic module. This class initiates the connection to the players registration table of the Derby database (lines 10 and 39 of Algorithm 3), generates clients ports and stores/accesses players information whenever it is necessary.

3.5.2 QCommon Module Changes

Qcommon module was modified in order to avoid client registration data loss when clients are leaving the game. This is shown in Algorithm 4 (implemented in Com.java of Jake2Grid Client source code):

Algorithm 4 Jake2Grid Client: Qcommon module

```
1: Quit Method {                                    ▷ Clients are leaving the game
2: Create connection to the database
3: Update players registration data
4: Terminate connection to the database
5: End client program }
```

When a client is leaving the game, a connection to the database is established in order to update his/her registration data and to avoid registration data loss (lines 2-5 of Algorithm 4).

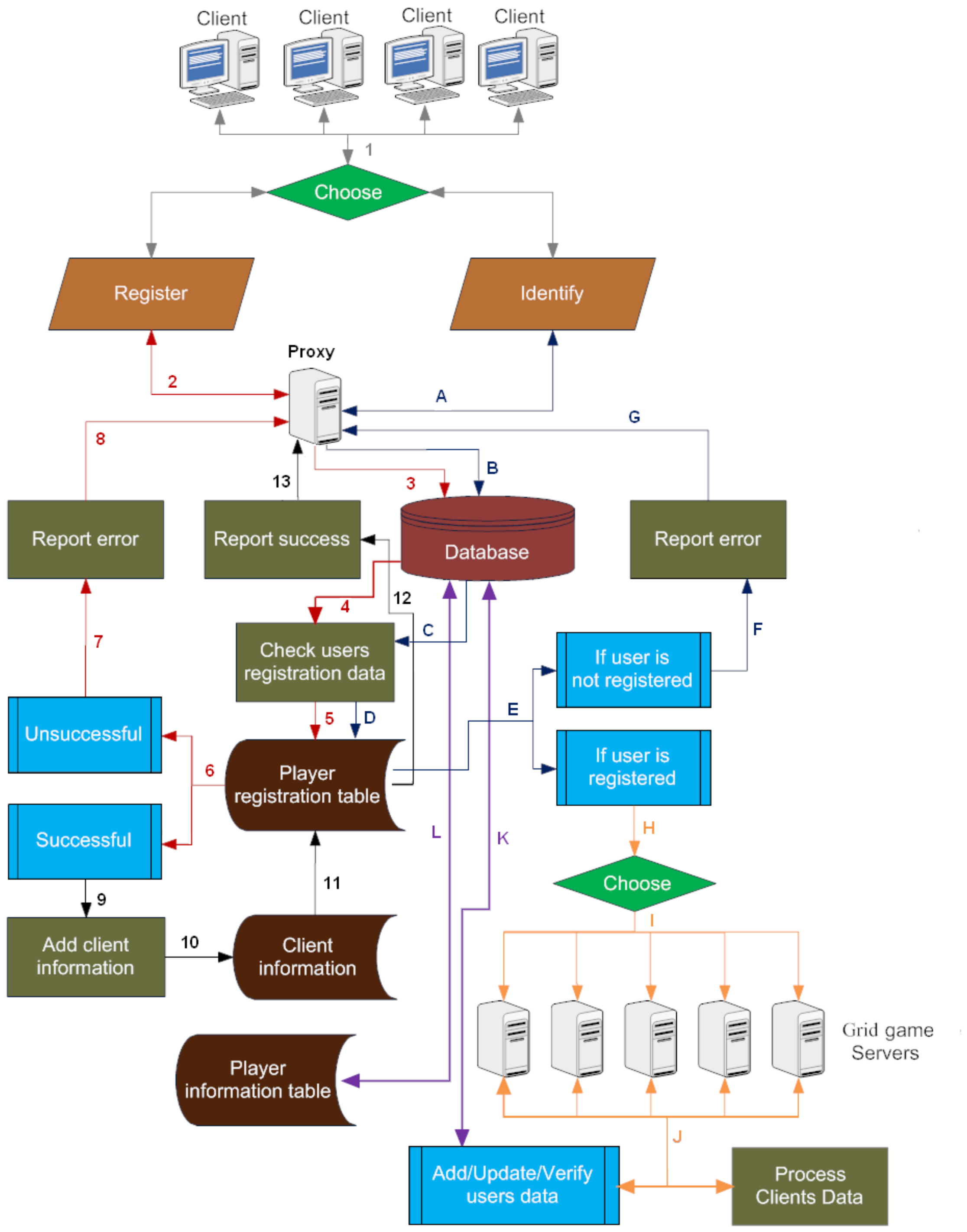


Figure 3.9: Jake2Grid Functioning.

3.5.3 Proxy Module

This is a new module that has been created for Jake2Grid. In this module, one machine is in charge of handling client registration/identification in order for them to play. As mentioned before, clients must register/identify themselves before they can play the game. This is done by using the Derby database and its clients registration table. If the registration/identification is unsuccessful, then the clients will have to re-insert their data and make another attempt in registering/identifying themselves. If the registration/identification is successful, clients will be provided with a list of game servers that are running on the grid. Clients choose one of those servers in which they want to play.

At any time, clients can switch between available game servers over the grid. When this switching occurs, client's data is stored in the Derby database on its client's information table. This happens when clients join/leave a game server. The game server accesses client's information in the Derby database, updates client's information and sends this information to the client that has just connected to the server. According to Figure 3.9, Jake2Grid registration works in the following manner:

1) Clients choose to register.

2) If clients choose to register, they send the register form to the proxy.

3) The proxy communicates with the derby database.

4) The proxy checks received client's data.

5) The proxy compares clients data with data in the players registration table in the database.

6) If client's data already exists, then proceed to step 7). If client does not exist proceed to step 9).

7) The proxy processes an error message to determine what happened.

8) The proxy sends error message back to the clients. Back to step 2).

9) The proxy is going to process the adding of client's data to the database.

10) The proxy transforms client data into SQL format.

11) The proxy adds client's data to the player registration table.

12) The proxy processes a success message.

13) The proxy sends a success message to clients. Back to step 1).

On the other hand, client identification works in the following manner:

1) Clients choose to identify.

A) If clients choose to identify, then they send the login form to the proxy.

B) The proxy communicates with the derby database.

C) The proxy checks received client's data.

D) The proxy compares client's data with data in the players registration table in the database.

E) If a client is not registered, then proceed to step F). If the user is registered proceed to step H).

F) The proxy processes an error message, to determine what happened.

G) The proxy sends error message back to the clients. Back to step 1).

H) If the client is registered then he/she will get a list of active grid game servers in which to play.

I) The client chooses one of the grid game servers in which to play. The client may choose another grid game server if he/she so wishes, thus proceeding back to step H).

J) The grid game servers process their client's data and communicate with the Derby database in order to add, verify, or to update client's data when it is necessary. If the latter occurs proceed to step K).

K) Grid game servers access the Derby database in order to add, verify, or to update client's data. When this access is over, return to step J).

L) Derby database accesses the player information table and performs any of the operations mentioned in step K). When this is over, return to step K).

3.6 Data Re-Engineering

This is the fourth step of the re-engineering software process. This step was applied to the game logic module. The data re-engineering done in this module has to to with keeping player states, i.e., to ensure persistence. Before proceeding to the changing of the game logic source code, we had to add a new module, the database module. This module would be used by the game logic module to store and access player states.

3.6.1 Database Module

This module was added to Jake2Grid and has to do with persistence. In order to do this we used the Derby database system. Derby is much like SQL, but it is easier to use, it uses less resources and it can be easily embedded into an application.

It is necessary to identify the player and the world information to be stored. As already mentioned, the game world is divided in game levels and each game level is run on a separate server independently and indefinitely. Also game level resources are unlimited and the game level is not affected by player's actions, so it is not necessary to store the game world/level state. According to the player's information it is necessary to store in the database the following data:

- Player's name.
- Player's score.
- Player's health.
- Player's weapon.
- Player's inventory.

The player's information is stored/accessed when the player leaves/joins a game level server, i.e., this is done in the game logic module.

3.6.2 Game Logic Module

In order to transform Jake2 into an MMOFPS, and as shown above, it was necessary to save the client's state, i.e., save their current information (Netname, Score, Health, Weapon, Inventory). This change can be seen in Algorithm 5 (implemented in PlayerClient.java of Jake2Grid Server source code):

Algorithm 5 Jake2Grid Server: Game logic module

```
 1: PlayerClient class {
 2: BeginDeathmatch Method {                 ▷ When players want to join a server
 3: if player is not new to the game then
 4:     Get players state data that is stored in the database
 5: else if player is new to the game then
 6:     Create new state data for player
 7: end if
 8: Remaining code execution }
 9: Client Disconnect Method {               ▷ When clients want to leave a server
10: if player has state data in the database then
11:     Update players state data in the database
12: else if player does not have state data in the database then
13:     Save new players data on the database
14: end if
15: Remaining code execution }
16: }
```

This algorithm is explained below:

- Line 3-7: Checks whether the player is a new player or not;
- Line 4: If the player is not a new player, then get his/her state data from the database;
- Line 6: If the player is a new player, then create new state data for him/her;
- Line 8: Player joins the server;
- Line 10-14: Checks whether or not player has state data in the database;
- Line 11: If the player has state data in the database, then update his/her state data;
- Line 13: If the player does not have state data in the database, then store new state data;
- Line 15: Player leaves the server.

3.7 Control-flow re-engineering

This is the last step of the re-engineering software process. This re-engineering step focused on changing parallelizable **for** loops on the server module. Before changing the server module source code, we had to add the GridGain module, which would ensure task division into jobs and job distribution among nodes.

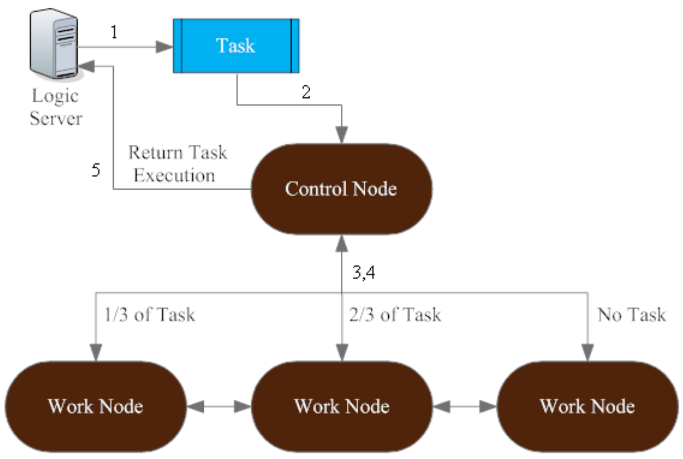


Figure 3.10: Jake2Grid grid functioning.

3.7.1 GridGain Module

This model was added to Jake2Grid and is in charge of task parallelization into jobs and job distribution among grid nodes. Several tasks performed on the server side were parallelized. Tasks were divided into sub-tasks or jobs. Jobs were created according to the number of tasks. Jobs can execute a sub-task or a set of sub-tasks. Jobs are then assigned to available grid nodes for execution. After their execution on the grid, each job returns a result. All jobs results are gathered into a final result using the reduce function. Finally, this final result will be sent back to the gridified method in question. This can be seen in Figure 3.10. In short, Jake2Grid works in the following manner:

1st – A task or several parallelizable tasks are chosen to be gridified.

2nd – Job creation is done according to the number of tasks and available nodes.

3rd – Each job is assigned to the grid node with more available resources.

4th – Each node returns its job execution to the control node.

5th – Control node gathers jobs results and returns them to the game server for updates.

In order to choose the tasks to be gridified, first it is necessary to know how the server works. As mentioned previously, Jake2 game server is responsible for world handling, movement, physics, game control and communication.

The gridified methods were the game control (Server_t method), communication (Calcpings and Ratedrop), movement (Push method) and indirectly some of the psychics methods. World handling methods were not gridified because each server/control node only handles one game level indefinitely and because game level resources are unlimited (e.g., unlimited ammunition, health packs and weapons).

Jake2 game source code was then changed to run on a grid:

1) After selecting the methods, it was necessary to add the grid communication module to the game server, thus creating a grid control node.

2) The source code of the selected methods was changed in order to allow them to be executed on the grid.

3) Creation of a grid task class, task division, job creation, job assignment to grid nodes, gathering of nodes results and return a final result to the game server:

   3.1) Each gridified method has a grid task class associated to it. A task receives required parameters from that method, and returns the results to such a method.

   3.2,3.3) Recall that only **for** instructions were gridified in Jake2Grid. Assuming that a serial **for** loop has *n* iterations, the corresponding parallel task is then divided into *n* jobs at maximum. The distribution of jobs depends on the available number of grid nodes. If the number of jobs is the same or greater than the number of available grid nodes, then the number of grid jobs is equal to the number of jobs. If the number of available grid nodes is lower than the number of jobs, then the number of jobs is equal to the number of grid nodes.

   3.4) After all jobs have been executed by the grid nodes, their results are gathered through the reduce function.

   3.5) After gathering all the results, the overall result is then sent back to the respective gridified method.

4,5) In the gridified method there is a variable that receives the final result from the task. The gridified method uses this variable to do updates.

6) If the gridified method is executed only once, after it has been executed on the grid, the game resumes its normal running state. If a gridified method is executed more then once or is executed as long as the game server is working, than every time the gridified method is executed, all the above steps are re-done till the game server is terminated.

3.7.2 Server Module

In order to transform Jake2 into a MMOG, two major decisions were taken. The first was to assign each game level to a separate game server, i.e, each game server only handles a game level indefinitely. This works as a preliminary server load distribution, and a way to promote the game scalability. The second decision was to integrate a grid middleware and to distribute the server load, as already described above.

After the integration of the GridGain middleware, Jake2's 32 maximum player limitation was surpassed, potentially improving the game scalability. Some important changes were done to the Jake2Grid Server source code:

- The inclusion of a database table, called Jake2Data, to store client information (Netname, Score, Health, Weapon, Inventory) in a persistent manner in the Derby database, as described in section 3.6.
- The creation of a grid node within Jake2Grid Server code in order to set up a connection between the server and the grid node, as described in the previous section.

The inclusion of the GridGain grid middleware improved the game scalability, as now there are several methods of the server side that are done in parallel by the grid nodes. GridGrain middleware changes to the server's source code were:

- Creation and initialization of a control node, which is assigned to the server (i.e., each time a game server runs it starts a control node).
- Selection of parallelized flow control instructions (e.g., for loops).

- Creation of grid task classes. We have encoded a task class for each parallelizable flow control instruction within the server code. This way, the server is capable of sending tasks to its associated control node for execution. Several instructions are processed in this control node, namely:

  a) Reception of tasks from parallelizable flow control instructions.

  b) Selection of task division mechanism.

  c) Grid job creation according to the available grid nodes and tasks.

  d) Grid jobs are assigned to the grid nodes that have more resources.

  e) Grid nodes execute grid jobs and return their results to the control node.

  f) Gathering of grid job results through the *Reduce* function.

After installing the grid middleware, and in order to transform Jake2 into a grid-based game, we had to rewrite/redesigned the main class of the game, as shown in Algorithm 6:

Algorithm 6 Jake2Grid Server Main Class

```
 1: Jake2 Class {                                   ▷ Main class of the game
 2: Create a grid control node
 3: Main Method {
 4: Start a Control node
 5: Remaining code execution
 6:         ▷ If anything went wrong, make sure the programmer knows what it was
 7: Handle and report any errors that might occur
 8:                ▷ if game server stops executing, terminate the control node
 9: Stop Control node }
10: }
```

This algorithm is explained below:

- Line 2: Creates a new grid control node;

- Line 4: Starts a control node;

- Line 5: Start a game server;

- Line 7: Handle possible errors and report them to the programmer;

- Line 9: Stop control node when game server terminates.

The last change to the server was the creation of a grid task class for each **for** loop of each parallelizable method for the codification of a for loop). We successfully parallelize four methods: `SV_CalcPings()`, `SV_RateDrop()`, `SV_Push()` and `server_t()`. Gridification of these methods was carried out through the parallelization of the **for** loops. The gridification of a for loop involved the creation of a class containing the codification the map and reduce functions as shown in Algorithm 9 (implemented in SV_Task_server_tFor.java of Jake2Grid Server source code). This way the parallelization of a for loop reduces to execute a task passing two parameters: (i) the name of the class concerning the for to be parallelized; (ii) a vector of two parameters. The first parameter is the initial value control variable of the for loop and the second concerns the number of iterations of the for loop. Usually, the name of a class argument is the name of the task. Each task class is exactly the same in terms of structure, the only thing that changes is their work and return result. An example of how to apply parallelization is shown in Algorithms 7 (implemented in server_t.java of the original Jake2 source code) and 8 (implemented in server_t.java of Jake2Grid Server source code):

Algorithm 7 Original Jake2: Server module

```
1: server_t Class {
2: ...
3: for $n = 0$ to number of iterations do          ▷ Parallelizable loop
4:     instruction
5: end for
6: Remaining code execution }
```

The **for** instruction in Algorithm 7 was modified (lines 3-5) to be run by the grid nodes. This can be seen in Algorithm 8 implemented in server_t.java of Jake2Grid Server source code):

Algorithm 8 Jake2Grid Server: Server module

```
1: server_t Class { ...
2: Parameters vector                 ▷ Vector that contains required parameters
3:                                   ▷ Creation of a grid task for the for instruction
4: Send For instruction to be executed on the grid.
5: Handle and report any errors that might occur
6: Remaining code execution }
```

This algorithm is as follows:

- Line 2: Creates a parameter vector that contains two values. The first concerns the value of the control variable of the original for loop, while the second concerns how many times the for loop is performed.

- Line 4: Creates a grid task class, send parameters vector to it and updates the models vector by "fetching" the result from the grid task class;

- Line 5: Handle possible errors and report them to the programmer;

- Line 6: Initializes other game server information.

We created a grid task class, according to lines 4-7 of Algorithm 8, for the **for** instruction. The grid task class performs the **for** instruction in parallel, so it needs to receive all the parameters used in the for instruction. After passing the arguments and the tasks to the grid task class, it is necessary to divide tasks into several sub-tasks for parallel execution. This procedure can be seen in Algorithm 9 (implemented in SV_Task_server_tFor.java of Jake2Grid Server source code):

Algorithm 9 Jake2Grid Server: Server module

```
 1: Server_t Task Class {                          ▷ Grid task class for the server_t method
 2: Create GridLoadBalancer                        ▷ Used to find the node with more resources
 3: Global variables
 4: Map Method {                                   ▷ Map-Reduce division mechanism
 5: Create Map of jobs
 6: Put received parameters into separate variables
 7: Create auxiliary variables if necessary
 8: if grid size $\geq$ tasks then                 ▷ grid nodes $\geq$ tasks: grid jobs = tasks
 9:     for $j = n$ to tasks do
10:         Create a grid job that receives sub-task j
11:         Grid job {
12:         executes sub-task j                    ▷ Each node will do exactly one sub-task
13:         return result }
14:         put job in the map of jobs
15:     end for
16: else                                           ▷ grid nodes $<$ tasks: grid jobs = grid nodes
17:     Define how much sub-tasks a node will do
18:     for $k = 0$ to grid size do
19:         Create a grid job that receives begin of the sub-task and the end of the
    sub-task
20:         Grid job {
21:         execute defined sub-task
22:         return result }
23:         put job in the map of jobs
24:         Update what is left to do of the defined sub-task
25:     end for
26: end if
27: return Map of jobs }                           ▷ Send map of jobs to grid nodes for execution
28: Reduce Method {             ▷ Gathers map of jobs and return the final result to the
    gridified method
29: return final result }                          ▷ Final result return to the gridified method
30: }
```

This algorithm is explained below:

- Line 1: Grid task class of the server_t method;
- Line 2: Create a grid load balancer that is used to verify nodes resources;
- Line 3: Creation of global variables to be sent to grid nodes;
- Line 4-27: Map-reduce division mechanism
- Line 5: Creation of a map of jobs (set of jobs) that is executed by the grid nodes;
- Line 6: Parsing of the received parameter vector into separate variables;
- Line 8-15: If the number of nodes is equal or bigger then the number of tasks, the number of created jobs will be equal to the number of tasks. Each job will do exactly one sub-task and will be placed on the map of jobs;
- Line 16-26: If the number of nodes is smaller then the number of tasks, the number of created jobs will be equal to the number of grid nodes. Each job will do one or several sub-tasks and will be placed on the map of jobs;
- Line 27: After all the jobs are in the map of jobs, the map of jobs is sent to the grid nodes for execution;
- Line 28-29: Reduce gather all grid nodes job execution into a final result, which then sends to the gridified method on the game server for updates.

As we can see above each grid nodes does a specific sub-task independent of what it is. Most of the times the number of available grid nodes is always inferior to the number of tasks to be done, leading to each node needing to do several sub-tasks.

3.8 Discussion

The new graphical user interface is much more efficient then the old one, because in the old one players needed to specify (through command line) to which server they wanted to join and play. On the other hand, in Jake2Grid, players have a list of available servers running on a grid. Thus, clients are able to use the list in order to chose a server to play.

In Jake2, players with the same name could join the same server, which became confusing to differentiate them. This was solved in Jake2Grid by adding a players registration table to the Derby database. Also in Jake2, players information is reseted every time a player joined/left a server. This is a very important aspect, because players rapidly abandon a game if they have to start a new game every single time they enter the game. This problem was solved in Jake2Grid by adding a players information table to the database. The fact that players have to be registered and have to identify themselves to play Jake2Grid is a very good idea, as most MMOGs require this in order to be played.

In Jake2Grid it is necessary to provide a client port to each client. This is not very efficient, as there are limited ports a client can use. Although a machine has about 50000 ports a client can use, some of these are reserved for certain protocols and applications and thus can not be used.

It is important to note that the gridification of wrong methods such as sequential, recursive and indirect access methods (methods that access other methods, or variables from other methods) can lead to bad results, which would then lead to wrong conclusions. So it is necessary to be extremely careful when choosing which methods are to be gridified. Before the final tests, we had gridified eight methods, but soon we conclude that only four were parallelizable indeed. This is mainly because some of the methods did not work with more than one node, while others lost execution time and generated higher latency. These methods did not work on more than one node because the game methods have an order on which to be executed, and can be acceded by other methods if needed. So if a method is gridified and another method tries to access the result of the gridified method, one of two things can happen:

1) If the gridified method involves variable instantiation, vector updating with dependent positions and recursive calls, a job can be still in execution while another method tries to access it, thus receiving an error.

2 If gridified method has a call to a non-gridified game method, a job will execute normally but wastes more time accessing the other method.

If either these two things occurs, the execution time and latency of Jake2Grid will degrade and in some cases will be worse than the execution time and latency of the old Jake2 game. In these circumstances, a method cannot be gridified.

3.9 Summary

Re-engineering a multiplayer FPS game in order to run it on a grid is a hard and complex task. However, we succeeded in re-engineering Jake2 with a minimum number of source code changes. We only changed or added a few classes of/to the source code in order to transform Jake2 into an MMOFPS running on a grid. In this process, we have followed the principles and methodology of software re-engineering.

Chapter 4

Scalability: Experiments and Results

In this chapter, we carry out a comparative study about the scalability of both games, Jake2 and Jake2Grid. That is, we intend to show how much scalable is Jake2Grid in comparison to Jake2, but keeping the quality of service (QoS), when the number of players increases. The scalability was subject to assessment against the game latency and execution time of the parallelized game methods.

4.1 Experiment Methodology

The game scalability of a FPS game like Jake2 depends on the network latency used to play. Nevertheless, this network latency must be smaller than the human visual latency to have real-time interaction. Bearing this in mind, we have defined the following experiment methodology to better assess scalability of Jake2Grid in relation to Jake2:

1) To study the problem of network latency of Jake2 against the principles followed by its developers and programmers, trying to explain the rationale behind the maximum number of players. Recall that, the maximum number of players per level is 32 in Jake2.

2) To correlate the network latency to the visual latency. As mentioned above, for the purpose of real-time interaction, the network latency must be smaller than the visual latency. This would allow us to come across a reference network latency for both Jake2 and Jake2Grid, as needed to carry out our experiments.

3) To proceed to experiments to assess the scalability of Jake2Grid. That is, using

the reference network latency achieved in the previous step, we are interested in knowing how many more players Jake2Grid can handle at interactive rates per level.

Before proceeding to the experiments we had to carefully study the latency of Jake2, because it depends on the type of the client's network connection access. The latency data concerning Jake2 was retrieved from

`http://ucguides.savagehelp.com/ConnectionFAQ/Quake2.htm`

on January 27, 2010, but currently it is no longer available:

- *LAN*: Recommended latency of 120 ms.
- *ADSL/Cable/Wireless*: Recommended latency of 120 ms.
- *ISDN Bounded*: Recommended latency of 90 ms.
- *ISDN Single (Stac/Microsoft compression)*: Recommended latency of 90 ms.
- *ISDN Single*: Recommended latency of 80 ms.
- *56K Modem (Stac/Microsoft compression)*: Recommended latency of 60 ms.
- *56K Modem (Hardware compression)*: Recommended latency of 55 ms.
- *V34 Modem (Stac/Microsoft compression)*: Recommended latency of 45 ms.
- *V34 Modem(Hardware compression)*: Recommended latency of 35 ms.

The five last options from above were discarded in our latency study because those technologies are no longer used in practice. We considered the lowest recommended latency of 90 ms, in order to guarantee that players with different connections would be able to play in real-time. However, since delays can occur, a 10 ms of delay compensation was added to the latency, yielding the reference latency of 100 ms. The latency is calculated on the server side and then sent as a scene frame to the client, which is then displayed on the client's graphical interface, as shown in Figure 4.1.

According to Figure 4.1, the current active player has a latency of 100 ms. Interestingly, the reference network latency agrees with the human visual latency, which varies between 80 (for lighter scenes) and 130 ms (for darker scenes) [148]. Note that Jake2 scenes are essentially in dark colors.

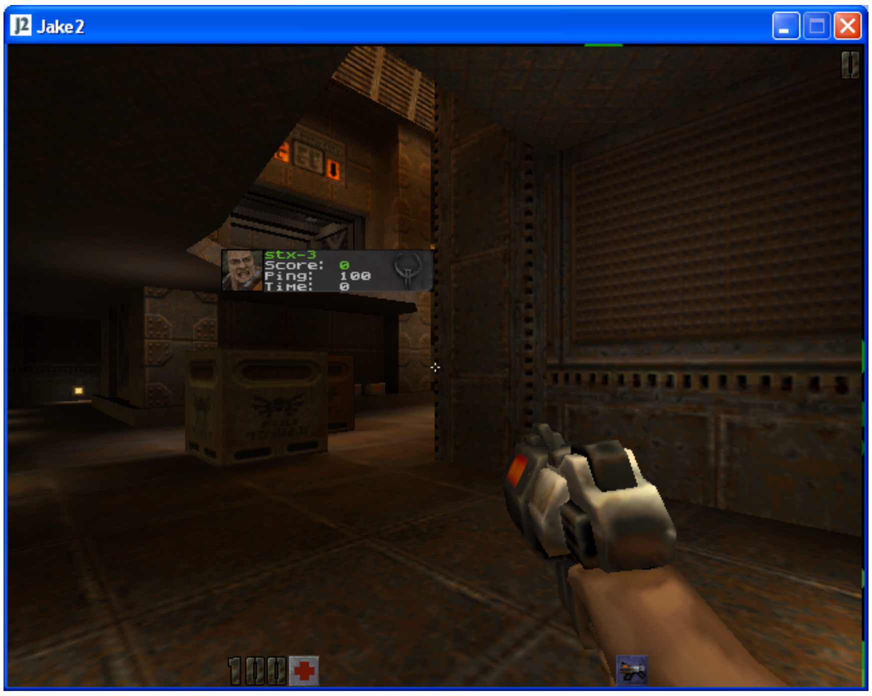


Figure 4.1: Jake2 Latency measurement example.

4.2 Experimental Scenarios

Before proceeding to experiments, we need to recall that Jake2 is structured in game levels that run independently of each other. So, by putting the game levels to run on different machines, we obtain a kind of parallel processing of the game.

To measure the scalability in terms of the increase in the number of players, we are assuming that Jake2 is the reference game. Measurements were done according to the following test scenarios:

- *Standard Grid*: This scenario corresponds to the standard grid, whose grid nodes communicate with others and share resources amongst themselves. Within this scenario, we used 8 CPU cores in order to process a single game level for 32 players (i.e., the admissible maximum number of players for Jake2). The communication amongst nodes is done via TCP/IP and the share of resources employs the map/reduce distribution mechanism.

- *Constrained Grid*: In this scenario, we used 8 machines with 8 CPU cores each to run our grid. Here, grid nodes communicate with others but do not share resources amongst themselves (i.e., jobs of one machine are done locally). Within this scenario, every machine would perform the role of a control node, i.e., each machine would be responsible for handling a game level for an increasing amount of players.

- *Singular Grid*: This test scenario follows the same procedure described in the second test scenario. However, in addition to cut the resource sharing, grid nodes were not able to communicate amongst themselves, i.e., what was done using a firewall.

To better assess the performance of each test scenario, bots (i.e., artificial intelligence agents) were programmed in order to play the role of clients. Note that we have only used one bot per client machine.

4.2.1 Standard Grid

In this first scenario, we have carried out two experiments:

- To determine which is the best hardware configuration to run a single game level on a grid.
- To compare the real-time performance within the maximum latency of both games, Jake2 and Jake2Grid, using the best hardware configuration determined in the first experiment.

4.2.1.1 Hardware Configurations

We used four hardware configurations with the same number of cores/processors, in order to assess which hardware configuration was the best to run a single game level for 32 players:

- *Eight single processor computers, no cores.* The specification of each one of these computers is: Intel Pentium 4 CPU, 2.80 GHz, 1.00 GB of RAM and with Fedora Core 2.6.23 operating system.
- *Four computers, each with a dual-core processor.* The specification of each one these computer is: Intel Pentium D CPU, 3.00 GHz, 1 GB of RAM and with Fedora Core 2.6.20 operating system.

- *Two computers, each with a quad-core processor.* The specification of each one of these computer is: Intel Core 2 Quad CPU, 2.83 GHz, 3.25 GB of Ram. One with Ubuntu 10.4 operating system and the other with OpenSuse 11.2 operating system.

- *One computer, with two quad-core processor.* The specification of this computer are: Quad-Core Intel Xeon with 2.86 GHz, 2 Processors, 8 Cores, 12 MB of L2 Cache, 6 GB of Memory, 16 GHz of Bus speed and with Mac OS X Server operating system.

Taking into account that we are testing a single game level, we only need a single server/control node for each one of the four hardware configurations as shown in Table 4.1. So, we are going to use one game server/control node and 7 work nodes at maximum, working with the TCP/IP protocol and using stationary bots (i.e., bots without shooting and moving activities).

# Grid Machines	Type of Processor	# Processors	# Cores	# Control Nodes	# Work Nodes	Latency (ms)
8	Single Processor	8	0	1	0	132
				1	1	1800
4	Dual Core	4	8	1	0	43
				1	1	500
2	Quad Core	2	8	1	0	41
				1	1	260
1	2 × Quad Core	2	8	1	0	41
				1	1	66
				1	2	81
				1	4	100
				1	5	120
				1	6	140
				1	7	160

Table 4.1: Hardware configurations.

The conclusions to draw from the results presented in Table 4.1 are the following:

- All the four hardware configurations are capable of supporting 32 clients playing simultaneously on a single control node and 0 work nodes.
- The first three configurations are unadvised to be used, as the addition of a single work node dramatically increases the game latency, whereas the addition of work nodes in the last configuration causes an average steady latency increase of 20 ms. Note that we are only adding work nodes and not players. But, it is necessary to take into consideration that the first three configurations use a 10/100Mb Ethernet card, while the last one uses a 1Gb Ethernet card.

In short, as expected, a grid running on a modern 2 × Quad-Core single computer has less latency than the other three hardware configurations. Adding a new work node increases the game latency, but this increasing is smaller when nodes are created in the same machine because the communication between the nodes occurs in the same machine, i.e., there is no need for the communication to pass through a switch/router. Even so, in the case of Jake2Grid, the better configuration seems to be a single control node doing all the job, i.e., it it not recommended to use work nodes.

4.2.1.2 Jake2 versus Jake2Grid

We are now at a position to compare Jake2 to Jake2Grid with 32, 64 and 96 players. Our plan is to use a single 2 × Quad-Core computer configuration with an increasing number of servers/control nodes and at the same time the number of players. Again, measurements were done using stationary bots and the TCP network protocol.

In addition to latency measurement, we also analyzed the execution time of parallelizable **for** instructions of both Jake2 and Jake2Grid. The measurements are shown in Table 4.2 and Table 4.3.

	Jake2		
# Players	32	64	96
# Servers/Control Nodes	1	2	3
Execution Time (ms)	0.184	0.184	0.184
Latency (ms)	85	90	95

Table 4.2: Execution time and latency measurements of Jake2 running on a single 2 × Quad-Core machine with 1Gb Ethernet card.

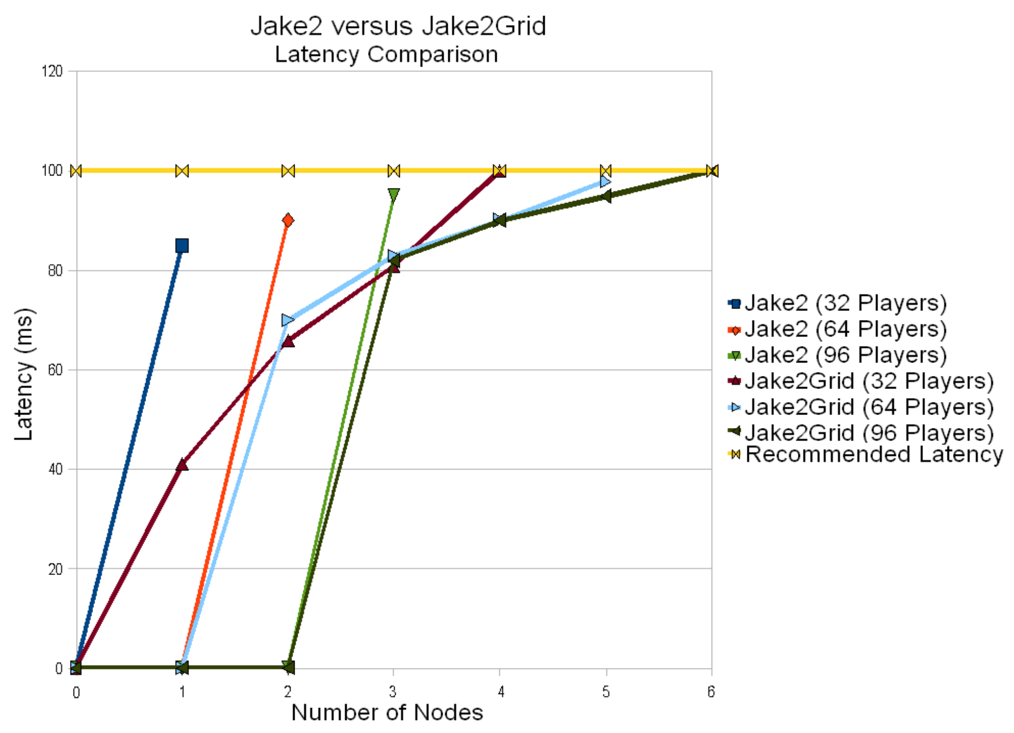
As shown in Table 4.2, the game latency increases an average of 5 ms per server of 32 players. So, it is possible to have 96 clients playing Jake2 simultaneously on a single machine, distributed across 3 servers or game levels. According to the execution time shown in Table 4.2, it is possible to see that the execution time is the same in the all servers. This is due to the fact that the servers are running independently of each other, i.e., they do not share resources.

	Jake2Grid											
# Players	32				64				96			
# Control Nodes	1				2				3			
# Work Nodes	0	1	2	3	0	1	2	3	0	1	2	3
Execution Time (ms)	0.167	0.152	0.141	0.129	0.124	0.118	0.112	0.107	0.110	0.105	0.102	0.098
Latency (ms)	41	66	81	100	70	83	90	98	82	90	95	100

Table 4.3: Execution time and latency measurements of Jake2Grid running on a single 2 × Quad-Core machine with 1Gb Ethernet card.

Similar to Jake2, we can only have up to 3 Jake2Grid game servers/control nodes on the same machine within real-time interactive rate. This can be explained by the fact that Jake2Grid is not parallelized that much. Recall that we only parallelized **for** instructions on the server side.

As shown in Table 4.3, we can have 96 real-time players distributed across 3 game servers and up to 3 work nodes running on the same machine. As expected, more work nodes means shorter execution times but at the cost of higher latency. Taking into consideration the execution times of Jake2 (Table 4.2) and Jake2Grid (Table 4.3), we easily conclude that the best grid configuration is only to use control nodes to distribute jobs to themselves, without the need to have work nodes. Taking also into account that each player takes approximately 0,85 ms (82 ms / 96 players = 0,85 ms) of the overall latency of 82 ms, we come across that we can have 117 clients playing at an interactive rate distributed among the 3 servers. This means that the scalability has increased $(117 - 96)/96 \times 100\% = 21,875\%$. For a more illustrative comparison between Jake2 and Jake2Grid in respect to latency and execution time, the reader is referred to Figure 4.2.



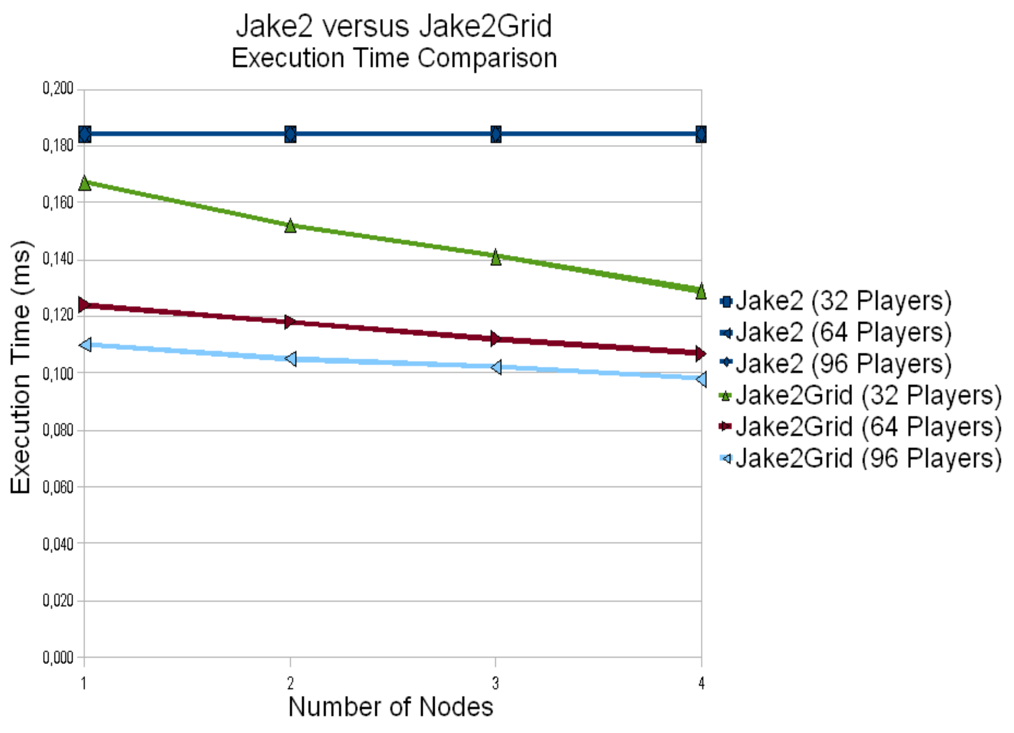


Figure 4.2: Comparison of Jake2 with Jake2Grid in respect to latencies and execution times listed in Tables 4.2 and 4.3.

4.2.2 Constrained Grid

As seen in the previous scenario, the best grid configuration for Jake2Grid is to use a $2 \times$ Quad-Core Xeon machine with only one control node. So, in this test scenario, we neither used work nodes nor the balancer mechanism (i.e., jobs from one machine are exclusively done by its control node). This scenario includes two experiments:

- 1 machine, 1 control node (1 game server);
- *n* machines, *n* control nodes (*n* game servers).

With the first experiment we intended to know how many players a single machine game server supports in real-time. The second experiment aims at checking how the latency is affected by adding a new single control node machine to the grid.

4.2.2.1 One Machine

In this experiment, measurements were then done using three types of clients:

- Stationary bots (i.e., bots do not move);
- Shooting stationary bots (i.e., bots would only shoot but not move);
- Shooting and moving bots (i.e., bots would move and shoot).

These latency measurements were done JMS and TCP networking protocols. Recall that, in principle, JMS is faster than TCP because the communication between nodes are local to the machine.

Stationary Bots

First we began measuring using stationary bots. These measurements are shown in Table 4.4. As shown in Table 4.4 (also see Figure 4.3), Jake2Grid is more scalable in respect to the number of players. Jake2 is capable of supporting a maximum of 40 players playing simultaneously in real-time, while Jake2Grid is capable of supporting 100 simultaneous players in real-time using the TCP protocol and 104 players using the JMS protocol. This means that the scalability using TCP protocol has increased $(100 - 40)/40 \times 100 = 150\%$, while the scalability using JMS protocol has increased $(104 - 40)/40 \times 100 = 160\%$.

Game	Players	TCP Latency (ms)	JMS Latency (ms)
Jake2	32	85 ms	–
	40	100 ms	–
	64	120 ms	–
	100	170 ms	–
Jake2Grid	32	41 ms	37 ms
	64	70 ms	66 ms
	100	100 ms	96 ms
	104	104 ms	100 ms

Table 4.4: Jake2 versus Jake2Grid scalability measurements using stationary bots.

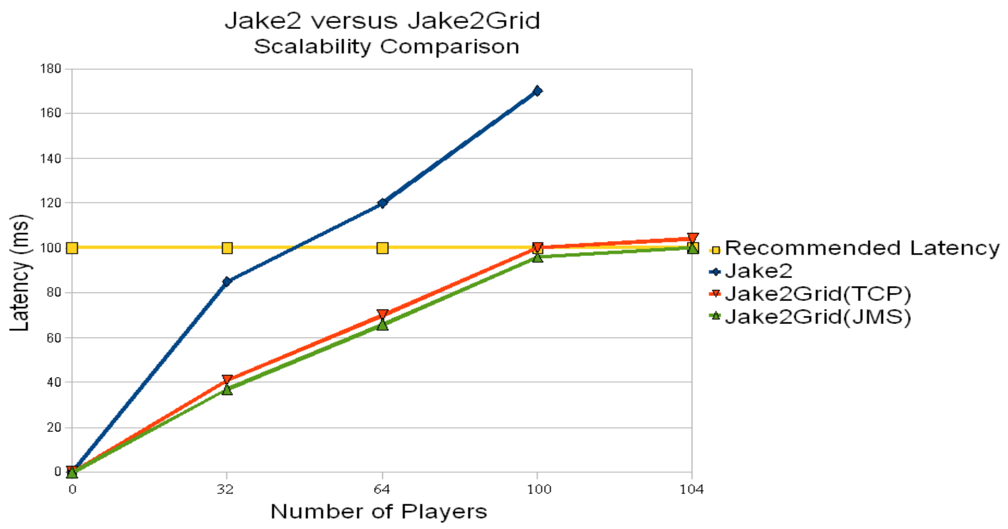


Figure 4.3: Comparative graphic of Jake2 versus Jake2Grid scalability measurements using stationary bots.

Stationary Shooting Bots

In order to make experiment measurements more "realistic", bots would now shoot in this experiment. These measurements are shown in Table 4.5.

Game	Players	TCP Latency (ms)	JMS Latency (ms)
Jake2	32	94 ms	-
	34	100 ms	-
Jake2Grid	32	45 ms	41 ms
	64	74 ms	70 ms
	96	100 ms	96 ms
	100	104 ms	100 ms

Table 4.5: Jake2 versus Jake2Grid scalability measurements using stationary shooting bots.

As shown in Table 4.5 (also see Figure 4.4), Jake2 is capable of supporting up to a maximum of 34 shooting players playing simultaneously in real-time, while Jake2Grid is capable of supporting 96 simultaneous players in real-time using the TCP protocol and 100 players using the JMS protocol. The scalability gains are now $(96 - 34)/34 \times 100\% = 182\%$ for TCP protocol and $(100 - 34/34) \times 100\% = 194\%$ for JMS protocol.

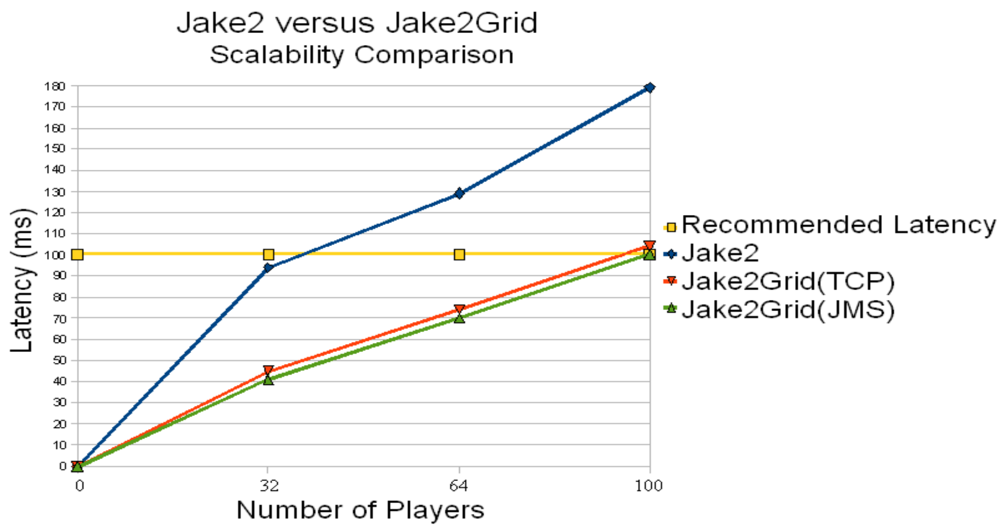


Figure 4.4: Comparative chart of Jake2 and Jake2Grid scalability measurements using stationary shooting bots.

Shooting and Moving Bots

This is the most realistic experiment because bots now move and shoot. The results of this experiment are shown in Table 4.6 and in Figure 4.5.

Game	Players	TCP Latency (ms)	JMS Latency (ms)
Jake2	32	99 ms	–
Jake2Grid	32	46 ms	42 ms
	64	75 ms	71 ms
	95	100 ms	96 ms
	99	104 ms	100 ms

Table 4.6: Jake2 VERSUS Jake2Grid scalability measurements using shooting and moving bots.

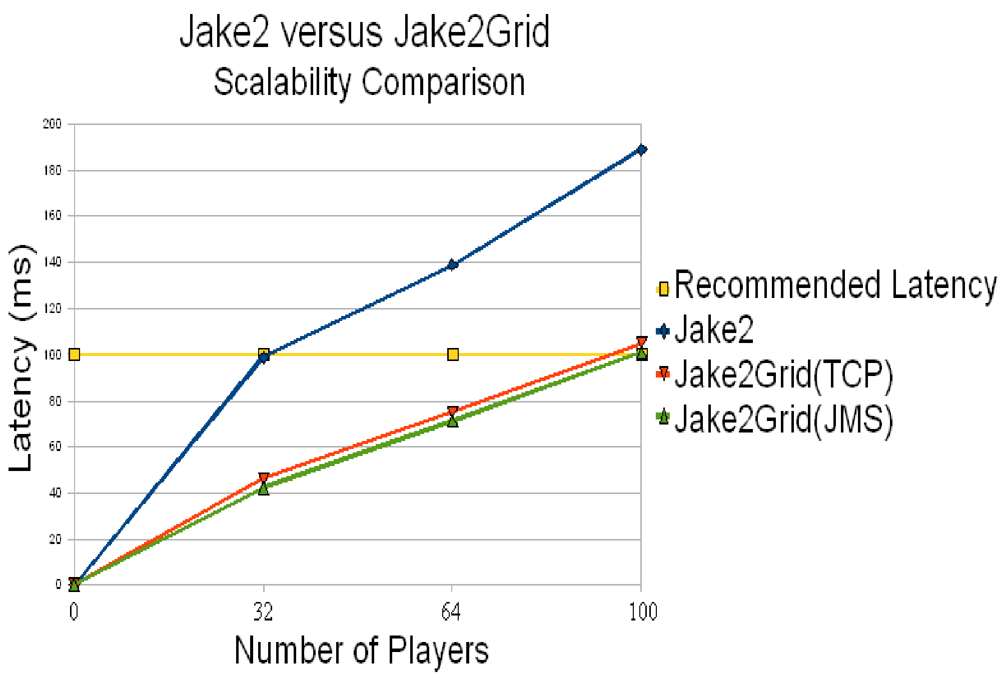


Figure 4.5: Comparative chart of Jake2 VERSUS Jake2Grid scalability measurements using shooting and moving bots.

Jake2 is now capable of supporting up to a maximum of 32 using shooting and moving players simultaneously in real-time, while Jake2Grid is capable of supporting 95 simultaneous players in real-time using the TCP protocol and 99 players using the JMS protocol. The scalability gains are now $(95 - 32)/32 \times 100\% = 197\%$ for TCP protocol and $(99 - 32)/32 \times 100\% = 209\%$ for JMS protocol.

4.2.2.2 Multiple Machines

In this experiment, we used up to 3 machines connected through a LAN network, each running a single game level/control node (and no work nodes). The clients are again shooting and moving bots in order to simulate a more realistic scenario and latency. We only used the JMS protocol because it proved to be faster than the TCP protocol in the previous experiment (Table 4.6) for a single machine.

In the present experiment, we have 3 machines exclusively using JMS, one node per machine. The results are shown in Table 4.7.

Latency/node	32 players/node			64 players/node			99 players/node		
	Node 1	Node 2	Node 3	Node 1	Node 2	Node 3	Node 1	Node 2	Node 3
1 Machine	42 ms	-	-	60 ms	-	-	100 ms	-	-
2 Machines	45 ms	45 ms	-	108 ms	67 ms	-	240 ms	204 ms	-
3 Machines	48 ms	48 ms	48 ms	126 ms	109 ms	105 ms	264 ms	191 ms	191 ms

Table 4.7: Latency (ms) measurements per node using shooting and moving bots on a constrained grid.

As expected, latency increases not only with the number of players (reading along rows for the same node in Table 4.7), but also with the number of machines (reading along columns in Table 4.7) in the same LAN. Like in the previous experiment, Jake2 is now capable of supporting up to a maximum of 32 shooting and moving players/machine simultaneously in real-time, and Jake2Grid is capable of supporting approximately a maximum of 190 players in 3 machines or nodes (64 players/node). Adding 32 players to each node duplicates the latency in average. The most important conclusion is that the best grid configuration is to have each machine in a different LAN, each running one control node, provided that the resulting latency is 100 ms for 99 players (Table 4.7). But this is only feasible for big companies like Sony, Microsoft and so on. Another alternative is described in the third scenario of the next section.

4.2.3 Singular Grid

In this scenario we used 3 computers with the following characteristics: Quad-Core Intel Xeon with 2.86 GHz, 2 Processors, 8 Cores, 6 GB of Memory and with Mac OS X Server operating system. As in the previous scenario, every machine runs one game level/control node. But, unlike the previous test scenario, the control node of each machine is not allowed to communicate and execute jobs for other control nodes (i.e., using a firewall) within the same LAN. The results are shown in Table 4.8.

Latency (ms)		# Players/Machine		
		32	64	100
# Machines	1	42	71	101
	2	42	71	101
	3	42	71	101

Table 4.8: Jake2 versus Jake2Grid scalability measurements using shooting and moving bots.

It is possible to verify through Table 4.8 that in this test scenario, each machine on the same LAN, running an independent control node, will not affect the latency of other machines present in the same LAN. Like in the previous experiment, Jake2 is now capable of supporting up to a maximum of 32 shooting and moving bots simultaneously in real-time and Jake2Grid is capable of supporting 99 players using the JMS protocol. The scalability gain is now $(99 - 32)/32 \times 100\% = 209\%$ for JMS protocol.

4.2.4 Comparison of the Test Scenarios

We compared the scalability gain of all three test scenarios in order to verify which one would be the ideal scenario for our game. This comparison was made according to machine node growth, i.e., what would happen when the number of machine nodes increased. Note that for each machine, Jake2 maximum players grows in stacks of 32. This comparison was made using shooting and moving bots and the JMS protocol for all test scenarios. However, we did not use the balancer mechanism for the second and third test scenarios. We measured that for the first test scenario, the measurements using JMS and one node add an average variation of 8 ms due to the balancer mechanism in comparison to the second test scenario. This measurements can be seen in Table 4.9 and Table 4.10.

	Standard Grid	Constrained Grid	Singular Grids
# Nodes	# Players/Machine		
1	91	99	99
2	52	64	99
3	52	64	99

Table 4.9: Comparison related to the number of players using shooting and moving bots.

	Standard Grid	Constrained Grid	Singular Grids
# Nodes	Scalability Gain (%)		
1	184 %	209 %	209 %
2	63 %	100 %	209 %
3	63 %	100 %	209 %

Table 4.10: Comparison related to the scalability gain using shooting and moving bots.

As shown in Table 4.9 and Table 4.10, a singular grid is the best solution for this type of game. With a singular grid we can have multiple machine on the same network without affect on the game's latency. An equivalent solution is the one provided by the constrained grid where the machines are geographically distributed across different LANs. In either case, the scalability gain is about 209 % (Table 4.10) in relation to the original Jake2.

4.3 Summary

The best configuration for Jake2Grid is to have several singular grids on the same LAN, each running one independent control node. Jake2Grid working with the JMS networking protocol is more scalable than Jake2Grid working with the TCP networking protocol and is more scalable than Jake2. However, it is possible to see that, the more game actions (e.g., shooting and moving) a bot player performs, the more latency it generates and the less gain we get. Also, the more nodes we have, the higher the latency will be and less gain we will get. In short, latency is affected by the number of players, player actions, number of grid nodes, and the sort of network protocol.

Chapter 5

Conclusions

From the results obtained in the previous chapter we concluded that, the usage of a singular grid (i.e., a single computer grid) offers better scalability, performance, resource usage and management in comparison with a standard grid. According to the results, a standard grid loses performance much more faster then a single computer grid. It is true that more work nodes provide a better execution time, but at cost of worsening the client latency, and since this a real-time application, client latency is more important than execution time; hence, the exclusive usage of control nodes in the game grid.

According to the measurements obtained for the old Jake2 game and the new Jake2Grid game, it is possible to see that Jake2Grid offers a better execution time and a better latency to a higher amount of players playing simultaneously on a same server. We can then conclude that Jake2Grid working with the JMS networking protocol is the best solution for supporting a high amount of players simultaneously while maintaining the recommended latency of 100 ms. Although we are adding one extra service that is running on the server side, which might worsen processing time, it allows a smaller latency because instead of the node having to send/get communication to/from another computer or to/from itself through multicast, it uses the internal memory of the computer (i.e., Java virtual machine) in order to send/retrieve needed data for execution.

In short, according to all the measurements and obtained results, it is possible to conclude that grid computing is in fact a possible solution for the scalability problems of MMOGs. Nevertheless, the main conclusions of this thesis come from its contributions. As already mentioned in Chapter 1, the major contributions of this thesis are:

1) A software re-engineering process that enable us to transform a multiplayer game

(Jake2) into a MMOFPS game (Jake2Grid) using grid computing.

2) We have also proved that, in some circumstances, grid computing can be used in real-time applications. In particular, grid computing can be used to improve the scalability of games in general.

Such a software re-engineering methodology can serve as a base for other real-time applications in case we intend to reformulate them using grid computing. This would be done without changing much of the original source code, i.e., only changing the necessary code for the application to work on a grid.

In the future, we mainly intend to check whether the development of an API layer between the server and the GridGain layers is viable. Nevertheless, we hope to more deeply study the problem of persistence in games as well as the scalability of the game against crowds of players, and the implications of the transitions of players from one game level into another.

References

[1] Henry Lowood. Videogames in computer space: The complex history of pong. *IEEE Annals of the History of Computing*, 31(3):5–19, July-September 2009.

[2] Anna Asanowicz. Education in virtual worlds. Whitepaper, 2003.

[3] Gamespy. Roy and richard play in the mud. in 25 smartest moments in gaming. *Gamespy*, (19), July 2003.

[4] Robert M. Metcalfe and David R. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1975.

[5] Don Fedyk and David Allan. Ethernet data plane evolution for provider networks. *IEEE Communications Magazine*, pages 84–89, March 2008.

[6] Gerald D. Cole. Performance measurements on the arpa computer network. In *Proceedings of the ACM Second Symposium on Problems in the Optimizations of Data Communications Systems*, pages 39–45, New York, USA, 1971. ACM.

[7] Alex Jarett and Jon Estanislao. IGDA online games white paper full version. In *Proceedings of the Game Developers Conference 2002*, pages 1–68, San Jose, California, USA, 2002.

[8] Alex Jarett, Jon Estanislao, Elonka Dunin, Jennifer MacLean, Brian Robbins, David Rohrl, John Welch, and Jefferson Valadares. IGDA online games white paper 2nd edition. *International Game Developers Association (IDGA)*, pages 1–140, March 2003.

[9] Tristan Henderson and Saleem Bhatti. Network games - a qos-sensitive application for qos-insensitive users? In *Proceedings of the ACM SIGCOMM 2003 Workshops*, Karlsruhe, Germany, August 2003.

[10] Sangheon Pack, Eunsil Hong, Yanghee Choi, Ilkyu Park, Jong-Sung Kim, and D. Ko. Game transport protocol: A reliable lightweight transport protocol for massively multiplayer on-line games (MMPOGs). In *Proceedings of the SPIE ITCOM Conference*, volume 4861, pages 83–94, Boston, USA, July 2002.

[11] John Chasey. The future of mobile gaming - multiplayer games. *Receiver Magazine*, (11):1–4, 2004.

[12] Tonio Triebel, Benjamin Guthier, Thomas Plotkowiak, and Wolfgang Effelsberg. Peer-to-peer voice communication for massively multiplayer online games. In *Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference (CCNC'09)*, pages 1282–1286, Piscataway, USA, 2009. IEEE Press.

[13] Elina Koivisto. Mobile games 2010. Whitepaper, 2010.

[14] Yung-Wei Kao, Pin-Yen Peng, Sheau-Ling Hsieh, and Shyan-Ming Yuan. A client framework for massively multiplayer online games on mobile devices. In *Proceedings of the 2007 International Conference on Convergence Information Technology (ICCIT)*, pages 48–53, Hydai Hotel Gyeongui, South Korea, November 2007.

[15] G. Deen, M. Hammer, J. Bethencourt, I. Eiron, J. Thomas, and J. H. Kaufman. Running quake ii on a grid. *IBM Systems Journal*, 45(1):21–44, 2006.

[16] Jeffrey Michael Parsons. *An Examination Of Massively Multiplayer Online Role-Playing Games As A Facilitator Of Internet Addiction.* PhD thesis, Graduate College Of The University Of Iowa, Iowa, USA, July 2005.

[17] Morgan Roberts. A study of the massively multiplayer online business model within the interactive entertainment industry. Master's thesis, Faculty Of San Francisco State University, California, USA, December 2005.

[18] Jengchung V. Chen and Yangil Park. The differences of addiction causes between massive multiplayer online game and multi user domain. *International Review of Information Ethics*, 4:53–60, December 2005.

[19] Nicolas Ducheneaut and Robert J. Moore. More than just 'xp': Learning social skills in massively multiplayer online games. *Interactive Technology and Smart Education*, 2(2):89–100, May 2005.

[20] Sanjay Jain and Charles R. McLean. Integrated simulation and gaming architecture fore incident management training. In F. B. Armstrong M. E. Kuhl, N. M. Steiger and J. A. Joines, editors, *Proceedings of the 2005 Winter Simulation Conference*, pages 904–913, Orlando, Florida, 2005. ACM Press.

[21] Constance Steinkuehler and Marjee Chmiel. Fostering scientific habits of mind in the context of online play. In *Proceedings of the 7th international conference on Learning sciences (ICLS)*, pages 723–729, Bloomington, Indiana, 2006.

[22] Mark Griffiths and Sara de Freitas. Online gaming and synthetic worlds as a medium for classroom learning. *Education And Health*, 25(4):74–76, 2007.

[23] Daniel Laughlin, Michelle Roper, and Kay Howell. Nasa eeducation roadmap: Research challenges in the design of massively multiplayer games for education & training. Technical Report Final, Federation of American Scientists, February 2007.

[24] Alicia Sanchez and Peter A. Smith. Emerging technologies for military game-based training. In *Proceedings of the Spring Simulation Multiconference (SpringSim)*, pages 296–301, Norfolk, USA, 2007. ACM Press.

[25] Bruno Van Den Bossche, Tom Verdickt, Bart De Vleeschauwer, Stein Desmet, Stijn De Mulder, Filip De Turck, Bart Dhoedt, and Piet Demeester. A platform for dynamic microcell redeployment in massively multiplayer online games. In *Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video (NOSSDAV)*, Newport, USA, 2006. ACM Press.

[26] Frank Glinka, Alexander Plob, Jens Muller-Iden, and Sergei Gorlatch. Rtf: A real-time framework for developing scalable multiplayer online games. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games (Netgames)*, pages 81–86, Melbourne, Australia, September 2007. ACM Press.

[27] Lu Fan, Hamish Taylor, and Phil Trinder. Mediator: A design framework for p2p MMOGs. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games (Netgames)*, pages 43–48, Melbourne, Australia, September 2007. ACM Press.

[28] Tobias Fritsch, Carsten Magerkurth, Benjamin Voigt, and Jochen Schiller. 4MOG - massive multiplayer middleware for mobile online games. In *Proceedings of The First International Workshop on Intercultural Collaboration (IWIC)*, Kyoto, Japan, January 2007.

[29] Sergei Gorlatch, Frank Glinka, Alexandre Plob, Jens Muller-Iden, Radu Prodan, Vlad Nae, and Thomas Fahringer. Enhancing grids for massively multiplayer online computer games. Technical Report TR-0134, Institute of Computer Science, University Of Munster, Germany, June 2008.

[30] Sergei Gorlatch, Frank Glinka, Alexandre Plob, Jens Muller-Iden, Radu Prodan, Vlad Nae, and Thomas Fahringer. A grid environment for real-time multiplayer online games. Technical Report TR-0133, Institute of Computer Science, University Of Munster, Germany, May 2008.

[31] Nitin Gupta, Alan Demers, and Johannes Gehrke. Semmo: A scalable engine for massively multiplayer online games. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 1235–1238, Vancouver, Canada, June 2008. ACM Press.

[32] Markus Hofer. Massively multiplayer online games and their implications for game-based learning. Whitepaper, January 2005.

[33] Victor Tay. Massively multiplayer online game (mmog) - a proposed approach for military application. In *Proceedings of the 2005 International Conference on Cyberworlds (CW)*, pages 396 – 400, Singapore, 2005. ACM Press.

[34] Gonçalo Amador, Ricardo Alexandre, and Abel Gomes. Re-engineering jake2 to work on a grid. In *Proceedings of the 7th Conference on Telecommunications*, Portugal, 2009. Instituto das Telecomunicações.

[35] Ricardo Alexandre, Paula Prata, and Abel Gomes. A grid infrastructure for online games. In *Proceedings of the 2009 International Conference on*

Information Sciences and Interaction Sciences (ICIS), pages 670–673, South Korea, November 24-26 2009. ACM Press.

[36] Leigh Achterbosch, Robyn Pierce, and Gregory Simmons. Massively multiplayer online role-playing games: The past, present, and future. *ACM Computers in Entertainment*, 5(4):1–33, 2008.

[37] Timothy D. Shields. 'second life' as a teaching tool: Can you have real learning in a virtual world? In *Proceedings of the 2008 Applied Business Research and College Teaching & Learning Conference (ABR & TLC)*, Orlando, USA, 2008.

[38] Douglas Thomas and John Seely Brown. The play of imagination - extending the literary mind. *Games And Culture*, 2(2):149–172, April 2007.

[39] Jack Thorpe. The mario brothers go to war (?). Whitepaper, May 2000.

[40] Brian Ballsun-Stanton. Tagon's toughs and improvisational it: A case study of the symoblic convergence theory with respect to rapidly forming online groups. Master's thesis, Rochester Institute of Technology, New York, USA, 2005.

[41] Richard Slater. What is the future of massively multiplayer online gaming? Master's thesis, University Of Brighton, England, February 2004.

[42] Daniel Voicu. Tibia review. Whitepaper, October 2007.

[43] Jim Whitehead. History of rpg's. Whitepaper, 2007.

[44] Tobias Fritsch, Hartmut Ritter, and Jochen Schiller. The effect of latency and network limitations on mmorpgs (a field study of everquest2). In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games (NetGames)*, pages 1 – 9, New York, USA, October 2005. ACM Press.

[45] Florian Totu. Final fantasy xiv taking beta applications. Whitepaper, December 2009.

[46] GameReplays.org. The history of real time strategy. Whitepaper, August 2008.

[47] Maid Marian Entretainment. Sherwood dungeon. Whitepaper, 2007.

[48] PS3 Magazine. Mag. *Playstation the Official Magazine*, pages 26–27, January 2010.

[49] Gnume. Massive multiplayer online games and their virtual economies. Master's thesis, Troy University of Montgomery, Canada, 2005.

[50] Rui Gil, José Pedro Tavares, and Licinio Roque. Architecting scalability for massively multiplayer online gaming experiences. In *Proceedings of the Digital Games Research Association Conference: Changing Views - Worlds in Play (DiGRA)*, Vancouver, Canada, 2005.

[51] Simon Joslin, Ross Brown, and Penny Drennan. The gameplay visualization manifesto: A framework for logging and visualization of online gameplay data. *ACM Computers In Entertainment*, 5(3):1–19, November 2007.

[52] Christopher Ruggles, Greg Wadley, and Martin R. Gibbs. Online community building techniques used by video game developers. In *Proceedings of the 4th International Conference on Entertainment Computing*, Sanda, Japan, September 2005.

[53] Wei-Hsiang Hsu. Business model developments for the pc-based massively multiplayer online game (MMOG) industry. Master's thesis, School of Economics and Information Systems, University of Wollongong, Australia, March 2005.

[54] Marco Roccetti, Stefano Ferretti, and Claudio E. Palazzi. The brave new world of multiplayer online games: Synchronization issues with smart solutions. In *Proceedings of the 11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 587–592, Orlando, USA, 2008.

[55] Evangelos Simoudis. Video gaming investment opportunities. Whitepaper, 2007.

[56] Stefano Cacciaguerra, Stefano Ferretti, Marco Roccetti, and Matteo Roffilli. Car racing through the streets of the web: a high-speed 3d game over a fast synchronization service. In *Proceedings of the 14th international conference on World Wide Web (WWW)*, pages 884–885, Chiba, Japan, May 2005.

[57] Allegra Mario, Fulantelli Giovanni, Gentile Manuel, La Guardia Dario, and Taibi Davide. On line environments to enhance entrepreneurial mindsets in young students. In *Proceedings of the 4th International Conference on Virtual Learning (ICVL)*, pages 91–97, Romania, 2009.

[58] Pedro de Almeida and Abel Gomes. Advanced strategic browser-based massive multiplayer online game: A game suggestion. In *Proceedings of the International Digital Games Conference (GAMES)*, pages 237–240, Portalegre, Portugal, 2006.

[59] Cartoon Network. Cartoon network launches first AAA, browser-based MMOG for kids. Whitepaper, 2009.

[60] SponsorPlay. Sponsorpay convinces gameforge. Whitepaper, 2009.

[61] Market Access Profile. Moggle, inc. Whitepaper, March 2010.

[62] Michel Simatic, Sylvie Craipeau, Antoine Beugnard, Sophie Chabridon, Marie-Christine Legout, and Eric Gressier. Technical and usage issues for mobile multiplayer games. Whitepaper, 2004.

[63] Shaowen Bardzell, Vicky Wu, Jeffrey Bardzell, and Nick Quagliara. Transmedial interactions and digital games. In *Proceedings of the international conference on Advances in computer entertainment technology (ACE)*, pages 307–308, Salzburg, Austria, June 2007.

[64] Elina M.I. Koivisto and Christian Wenninger. Enhancing player experience in mmorpgs with mobile features. In *Proceedings of the Digital Games Research Association Conference: Changing Views - Worlds in Play (DiGRA)*, Vancouver, Canada, 2005.

[65] Paul Coulton, Klen Copic Pucihar, and Will Bamford. Mobile social gaming. Whitepaper, 2008.

[66] K. Prasetya and Z. D. Wu. Performance analysis of game world partitioning methods for multiplayer mobile gaming. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames)*, pages 72–77, Worcester, Massachusetts, 2008.

[67] Pauline Brutlag. Video game genres and some definitions. In *Proceedings of the SimWorkshops - Designing Compelling Medical Games*, Stanford, California, October 2005.

[68] Sylvie Noel and Sarah Dumoulin. Cheaters, resource farmers, ninja looters, and gankers: If we make collaborative virtual environments more like games, should

our users be worried? In *Proceedings of the Computer Games and Computer Supported Cooperative Work Workshop (ECSCW)*, Paris, France, September 2005.

[69] Simon Egenfeldt-Nielsen. Mapping online gaming: Genres, characteristics and revenue models. *Game Research: The art, business, and science of video games*, May 2006.

[70] Anne-Gwenn Bosser. Massively multi-player games: Matching game design with technical design. In *Proceedings of the international conference on Advances in computer entertainment technology (ACE)*, pages 263–268, Singapore, June 2004.

[71] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *Proceedings of the Special Interest Group on Data Communication (SIGCOMM)*, pages 389–400, Seattle, USA, August 2008.

[72] Jouni Smed, Timo Knuutila, and Harri Hakonen. A review on networking and multiplayer computer games. Technical Report 454, Turku Centre For Computer Science, Finland, 2002.

[73] Nathaniel E. Baughman, Marc Liberatore, and Brian Neil Levine. Cheat-proof playout for centralized and peer-to-peer gaming. *IEEE/ACM Transactions On Networking*, 15(1):1–13, February 2007.

[74] Jiankun Hu and Fabio Zambetta. Security issues in massive online games. *Security Communication Networks*, (1):83–92, 2008.

[75] Kuan-Ta Chen, Chun-Ying Huang, Polly Huang, and Chin-Laung Lei. An empirical evaluation of tcp performance in online games. In *Proceedings of the international conference on Advances in computer entertainment technology (ACE)*, Hollywood, USA, June 2006.

[76] Kuan-Ta Chen, Polly Huang, and Chin-Laung Lei. How sensitive are network quality? *The Communications of The ACM*, 49(11):34–38, November 2006.

[77] Knut Hakon T. Morch. Cheating in online games - threats and solutions. Whitepaper, January 2003.

[78] Jeff Yan and Brian Randell. A systematic classification of cheating in online games. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games (NetGames)*, pages 1–9, Hawthorne, USA., October 2005. ACM Press.

[79] Joel Zetterstrom. A legal analysis of cheating in online multiplayer games. Master's thesis, School Of Economics And Commercial Law, Goteborg University, Sweden, March 2005.

[80] Jouni Smed, Timo Knuutila, and Harri Hakonen. Can we prevent collusion in multiplayer online games? In Honkela, Raiko, Kortela, and Valpola, editors, *Proceedings of the Ninth Scandinavian Conference on Artificial Intelligence*, pages 168–175, Espoo, Finland, October 2006.

[81] Dieter Gollmann. *Computer Security*. John Wiley & Sons., 1999.

[82] Philippe Golle and Nicolas Ducheneaut. Preventing bots from playing online games. *ACM Computers In Entertainment*, 3(3):1–10, July 2005.

[83] Philippe Golle and Nicolas Ducheneaut. Keeping bots out of online games. In *Proceedings of the international conference on Advances in computer entertainment technology (ACE)*, pages 262–265, Valencia, Spain, 2005.

[84] S.F. Yeung, John C.S. Lui, Jiangchuan Liu, and Jeff Yan. Detecting cheaters for multiplayer games: Theory, design and implementation. In *Proceedings of the 3rd IEEE Consumer Communications and Networking Conference (CCNC)*, pages 1178 – 1182, Las Vegas, USA, January 2006.

[85] Margaret DeLap, Bjorn Knutsson, Honghui Lu, and Oleg Sokolsky. Is runtime verification applicable to cheat detection? In *Proceedings of the Special Interest Group on Data Communication (SIGCOMM)*, pages 134–138, USA, August-September 2004.

[86] Christian Monch, Gisle Grimen, and Roger Midtstraum. Protecting online games against cheating. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games (NetGames)*, pages 1–11, Singapore, October 2006.

[87] Daniel James, Gordon Walton, Brian Robbins, Elonka Dunin, Greg Mills, John Welch, Jeferson Valadares, Jon Estanislao, and Steven DeBenedictis. 2004 persistent worlds whitepaper. Whitepaper, 2004.

[88] Robert Greene. Advanced data management for mmog - the versant object database in mmog applications. Versant whitepaper, Versant Corporation, 2007.

[89] Kaiwen Zhang, Bettina Kemme, and Alexandre Denault. Persistence in massively multiplayer online games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames)*, pages 53–58, Worcester, Massachusetts, 2008. ACM Press.

[90] Steven Daniel Webb and Sieteng Soh. Cheating in networked computer games - a review. In *Proceedings of the 2nd international conference on Digital interactive media in entertainment and arts (DIMEA)*, pages 105–122, Perth, Australia, 2007. ACM Press.

[91] Eric Cronin, Burton Filstrup, and Anthony Kurc. A distributed multiplayer game server system, May 2001.

[92] Bjorn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In *Proceedings of the 23rd Conference of the IEEE Communications Society (INFOCOM)*, pages 1–12, Hong Kong, China, 2004.

[93] Jehn-Ruey Jiang, Yu-Li Huang, and Shun-Yun Hu. Scalable aoi-cast for peer-to-peer networked virtual environments. In *Proceedings of the 28th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 447–452, Beijing, China, June 2008.

[94] Sabine Cikic, Sven Grottke, Fritz Lehmann-Grube, and Jan Sablatnig. Cheat-prevention and -analysis in online virtual worlds. In *Proceedings of the 1st International Conference on Forensic Applications and Techniques in Telecommunications (e-Forensics)*, Adelaide, Autralia, January 2008.

[95] Abdennour El Rhalibi, Madjid Merabti, and Yuanyuan Shen. Aoim in peer-to-peer multiplayer online games. In *Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology (ACE)*, pages 1–8, Hollywood, California, June 2006.

[96] Lars Aarhus, Knut Holmqvist, and Martin Kirkengen. Generalized two-tier relevance filtering of computer game update events. In *Proceedings of the 1st workshop on Network and system support for games (NetGames)*, pages 10–13, Braunschweig, Germany, April 2002.

[97] Philippe David and Ariel Vardi. Improving scalability in MMOGs - scalamo: A new architecture. Technical Report Final, Georgia Institute of Technology, April 2006.

[98] Bart De Vleeschauwer, Bruno Van Den Bossche, Tom Verdickt, Filip De Turck, Bart Dhoedt, and Peit Demeester. Dynamic microcell assignment for massively multiplayer online gaming. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games (NetGames)*, pages 1–7, Hawthorne, USA, October 2005. ACM Press.

[99] Michal Hapala. Programming techniques for the development of massive multiplayer on-line games. Whitepaper, June 2006.

[100] Martijn Moraal. Massive multiplayer online game architectures. Whitepaper, January 2006.

[101] Steven Daniel Webb, William Lau, and Sieteng Soh. Ngs: An application layer network game simulation. In *Proceedings of the 3rd Australasian conference on Interactive entertainment*, pages 15–22, Perth, Australia, December 2006. ACM Press.

[102] Amy Apon, Rajkumar Buyya, Hai Jin, and Jens Mache. Cluster computing in the classroom: Topics, guidelines, and experiences. Whitepaper, May 2001.

[103] Craig B. Zilles and Gurindar S. Sohi. Master/slave speculative parallelization. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 85 – 96, Istanbul, Turkey, November 2002.

[104] Sladjan Bogojevic and Mohsen Kazemzadeh. The architecture of massive multiplayer online games. Master's thesis, Department of Computer Science, Lund Institute of Technology, Sweden, September 2003.

[105] Roger Delano Paul McFarlane. Network software architectures for real-time massively-multiplayer online games. Master's thesis, School of Computer Science, McGill University, Canada, February 2005.

[106] Luther Chan, James Yong, Jiaqiang Bai, Ben Leong, and Raymond Tan. Hydra: A massively-multiplayer peer-to-peer architecture for the game developer. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games (NetGames)*, pages 37–42, Melbourne, Australia, September 2007.

[107] Neil J. Gunther. *Guerrilla Capacity Planning*, chapter Gargantuan Computing-GRIDs and P2P, pages 165–177. Springer Berlin Heidelberg, January 2007.

[108] Takuji Iimura, Hiroaki Hazeyama, and Youki Kadobayashi. Zoned federation of game servers: A peer-to-peer approach to scalable multiplayer online games. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games (NetGames)*, pages 1–5, Portland, USA, August-September 2004.

[109] Patric Kabus, Wesley W. Terpstra, Mariano Cilia, and Alejandro P. Buchmann. Addressing cheating in distributed MMOGs. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games (NetGames)*, pages 1–6, Hawthorne, USA, October 2005.

[110] Sara de Freitas. Learning in immersive worlds. Whitepaper, 2007.

[111] Chris Chambers, Wu chang Feng, and Sambit Sahu Debanjan Saha. Measurement-based characterization of a collection of on-line games. Whitepaper, 2005.

[112] Grenville Armitage, Carl Javier, and Sebastian Zander. Measuring a wolfenstein enemy territory master server's respone to game client queries. Technical Report 060410A, Swinburne University of Technology, Melbourne, Australia, 2006.

[113] David Kushner. Engineering everquest: online gaming demands heavyweight data centers. *IEEE Spectrum*, 42(7):34–39, July 2005.

[114] Shun-Yun Hu, Shao-Chen Chang, and Jehn-Ruey Jiang. Voronoi state management for peer-to-peer massively multiplayer online games. In *Proceedings of the*

5th IEEE Consumer Communications and Networking Conference (CCNC), pages 1134 – 1138, Las Vegas, USA, January 2008.

[115] Scott Douglas, Egemen Tanin, Aaron Harwood, and Shanika Karunasekera. Enabling massively multi-player online gaming applications on a P2P architecture. In *Proceedings of the International Conference on Information and Automation*, Colombo, Sri Lanka, December 2005.

[116] Intel. High availability server - extending the benefits of raid technology into the server arena. Whitepaper, 2002.

[117] Gerd Heber, David Lifka, and Paul Stodghill. Post-cluster computing and the next generation of scientific applications. Whitepaper, July 2002.

[118] Edward Whalen. A survey of popular clustering technologies. Whitepaper, 2002.

[119] Inc JBoss. The jboss 4 application server clustering guide - jboss as 4.0.5. Whitepaper, 2006.

[120] Abdennour El Rhalibi and Madjid Merabti. Agents-based modeling for a peer-to-peer MMOG architecture. *ACM Computers In Entertainment*, 3(2):1–19, April 2005.

[121] Madjid Merabti and Abdennour El Rhalibi. Peer-to-peer architecture and protocol for a massively multiplayer online game. In *Proceedings of the Global Telecommunications Conference Workshops (GlobeCom)*, pages 519–528, Texas, USA, 2004.

[122] Steven Daniel Webb, Sieteng Soh, and William Lau. Racs: A referee anti-cheat scheme for p2p gaming. In *Proceedings of the 17th International workshop on Network and Operating Systems Support for Digital Audio & Video (NOSSDAV)*, Urbana-Champaign, USA, 2007.

[123] Shun-Yun Hu, Jui-Fa Chen, and Tsu-Han Chen. Von: A scalable peer-to-peer network for virtual environments. *IEEE Network*, 20(4):22–31, July-August 2006.

[124] Chris GauthierDickey, Virginia Lo, and Daniel Zappala. Using n-trees for scalable event ordering in peer-to-peer games. In *Proceedings of the 15th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 87–92, Skamania, USA, June 2005.

[125] Patrice Muller. Scalable localized histogram aggregation for p2p MMOGs. Master's thesis, Eidgenossische Technische Hochschule (ETH), Zurich, Switzerland, July 2005.

[126] Gregor Schiele, Richard Suselbeck, Arno Wacker, Jorg Hahner, Christian Becker, and Torben Weis. Requirements of peer-to-peer-based massively multiplayer online gaming. In *Proceedings of the 7th International Symposion on Cluster Computing and the Grid (CCGRID)*, pages 773 – 782, Rio de Janeiro, Brazil, May 2007.

[127] Shanika Karunasekera, Scott Douglas, Egemen Tanin, and Aaron Harwood. P2p middleware for massively multi-player online games. In *Proceedings of the 6th International Middleware Conference*, Grenoble, France, 2005.

[128] Simon Rieche, Klaus Wehrle, Marc Fouquet, Heiko Niedermayer, Leo Petrak, and Georg Carle. Peer-to-peer-based infrastructure support for massively multiplayer online games. In *Proceedings of the Consumer Communications and Networking Conference (CCNC)*, pages 763–767, Las Vegas, USA, January 2007.

[129] Shun-Yun Hu and Guan-Ming Liao. Scalable peer-to-peer networked virtual environment. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games (NetGames)*, pages 129–133, USA, August-September 2004.

[130] Helge Backhaus and Stefan Krause. Voronoi-based adaptive scalable transfer revisited: gain and loss of a voronoi-based peer-to-peer approach for MMOG. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games (NetGames)*, pages 49–54, Melbourne, Australia, September 2007.

[131] Thorsten Hampel, Thomas Bopp, and Robert Hinn. A peer-to-peer architecture for massive multiplayer online games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games (NetGames)*, pages 1–4, Singapore, October 2006.

[132] Alvin Chen and Richard R. Muntz. Peer clustering: A hybrid approach to distributed virtual environments. In *Proceedings of 5th ACM SIGCOMM*

workshop on Network and system support for games (NetGames), pages 1–9, Singapore, October 2006.

[133] Fábio Reis Cecin, Jorge Luis Victória Barbosa, and Cláudio Fernando Resin Geyer. A communication optimization for conservative interactive simulators. *IEEE Communications Letters*, 10(9):686–688, September 2006.

[134] Lan Yang and Peerapong Sutinrerk. Mirrored arbiter architecture: A network architecture for large scale multiplayer games. In *Proceedings of the Summer Computer Simulation Conference (SCSC)*, pages 709–716, San Diego, California, 2007. ACM Press.

[135] Azali Bin Saudi. Parallel computing. Whitepaper, 2008.

[136] Dimitri P. Bertsekas and John N. Tsitsiklis. Some aspects of parallel and distributed iteractive algorithms - a survey. *Automanca*, 27(1):3–21, 1991.

[137] Dean E. Dauger and Viktor K. Decyk. "plug-and-play cluster" computing using mac os x. In *Proceedings of IEEE International Conference on Cluster Computing*, pages 430–435, 2003.

[138] Christopher Stanton, Rizwan Ali, Yung-Chin Fang, and Munira A. Hussain. Installing linux high-performance computing clusters. *PowerSolutions*, 2001(4):11–16, 2001.

[139] Alfredo Buttari, Piotr Luszczek, Jakub Kurzak, Jack Dongarra, and George Bosilca. A rough guide to scientific computing on the playstation 3. Technical Report UT-CS-07-595, Innovative Computing Laboratory, University of Tennessee, USA, 2007.

[140] Egan Ford, Brad Elkin, Scott Denham, Benjamin Khoo, Matt Bohnsack, Chris Turcksin, and Luis Ferreira. *Building a Linux HPC Cluster with xCAT*. International Business Machines (IBM), September 2002.

[141] Onur Celebioglu, Ramesh Rajagopalan, and Rizwan Ali. Exploring infiniband as an hpc cluster interconnect. Whitepaper, October 2004.

[142] Luis Ferreira, Viktors Berstis, Jonathan Armstrong, Mike Kendzierski, Andreas Neukoetter, Masanobu Takagi, Richard Bing-Wo, Adeeb Amir, Ryo Murakawa,

Olegario Hernandez, James Magowan, and Norbert Bieberstein. *Introduction to Grid Computing with Globus*. International Business Machines (IBM), 2003.

[143] Stephen Jarvis, Nigel Thomas, and Aad van Moorsel. Open issues in grid performability. *International Journal of Simulation*, 5(5):3–12, 2004.

[144] Nigel Thomas. Challenges and opportunities in grid performability. Technical Report 842, School Of Computing Science, University Of Newcastle Upon Tyne, England, May 2004.

[145] W. E. Johnston, D. Gannon, and B. Nitzberg. Grids as production computing environments: The engineering aspects of nasa's information. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing Power Grid*, Redondo Beach, USA, 1999.

[146] Thomas Fahringer, Christoph Anthes, Alexis Arragon, Arton Lipaj, Jens Muller-Iden, Christopher Rawlings, Radu Prodan, and Mike Surridge. *The Edutain@Grid Project*, volume 4685. Springer Berlin/Heidelberg, August 2007.

[147] Ian Sommerville. *Software Re-Engineering*, chapter 28. Addison Wesley, 2000.

[148] William A. Mackay. *Neurophsysiology without tears*. Sefalotek Ltd, 1997.