JOURNEY, A SHARED VIRTUAL SPACE MIDDLEWARE

by

Alexandre Denault

School of Computer Science
McGill University, Montreal

Fall 2010

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Doctor of Philosophy

Abstract

The complexity of developing multiplayer games, along with their popularity, has grown tremendously in the recent years. The most complex of these, Massively Multiplayer Games (MMOGs), require developers to deal with many issues, such as scalability, reliability and cheat prevention. Although individual solutions to these problems exists, very little academic work has been done to address all these issues simultaneously. In addition, experimentation in these areas can require a significant implementation effort.

In this work, we present Journey, a unified framework that address all these issues in a simple, modular and efficient architecture leveraging replicated objects. Scalability is addressed through the use of a dynamic cell load-balancing strategy while fault tolerance and cheat prevention are achieved by leveraging existing replicated objects in the system. The proposed framework is implemented using numerous enhancements not found in traditional replication, like obstacle-aware partitioning and remote procedure call systems.

The efficiency of this framework is illustrated through the use of Mammoth, a massively multiplayer research framework. Using experimental data from human players, artificial players (NPC) were built and used to stress test and gather performance data. Analysis of this data demonstrated that load balancing provides important scalability benefits while very little overhead is incurred from the fault tolerance and cheat prevention systems.

Résumé

Dans les dernières années, la popularité des jeux multi-joueurs a connu une croissance sans égale. Cette croissance a aussi provoqué une augmentation importante dans la complexité de développement, surtout pour les jeux en ligne massivement multi-joueurs (MMOGs). Ces jeux posent des problèmes sérieux, tel que la croissance de capacité, la fiabilité et la prévention de la tricherie. Quoiqu'il existe de nombreuses solutions pour chacun de ces problèmes, très peu de travail académique adresse tous ces problèmes ensembles. De plus, l'expérimentation dans ces domaines nécessite de grands efforts de développement.

Ce document présente Journey, un cadre de librairies informatiques unifiées qui adresse tous ces problèmes avec une architecture simple, modulaire et efficace tirant parti de la technologie des objets répliqués. Journey utilise un système d'équilibrage de charge avec cellule dynamique pour pallier aux problèmes de capacité. De plus, les défis de tolérance des failles et la prévention de la tricherie peuvent être adressés à l'aide des objets déjà répliqués dans le système. L'outil proposé utilise plusieurs améliorations qui n'existe pas dans la réplication traditionnelle, tel que la division des espaces prenant compte des obstacles et l'exécution de méthode distantes

La performance de Journey est évaluée à l'aide de Mammoth, un outil de recherche pour les environnements massivement multi-joueurs. À l'aide de données expérimentales de joueurs humains, des joueurs artificiels on été construits pour mesurer la capacité et la performance de l'outil proposé. L'analyse de ses données démontre que l'équilibre des charges démontre une grande augmentation de capacité. De plus, les systèmes de tolérance de fautes et de prévention de la tricherie on très peu d'impact sur la performance du système.

Acknowledgments

I would like to sincerely thank my supervisor, Professor Jörg Kienzle, for his guidance, assistance and support. I am extremely grateful for the freedom and flexibility given to me to explore different research directions. I also truly appreciate his patience, considering the strong opinions I sometimes have.

Those words can be found in my Master's thesis, and they deserve to be repeated. Professor Kienzle's faith and confidence in me have never wavered, even throughout the most difficult of times, and for that, I will be forever grateful. I would also like to thank Bettina Kemme, Hans Vangheluwe, Clark Verbrugge and Joseph Vybihal for enabling me to become the researcher I am today.

None of this work would have been possible without the Mammoth team, from the first development team back in 2005 to its current members. Although there are too many members to name, two developers stand out, Marc Lanctot and Michael Hawker. Their outstanding contribution and loyalty were essential to the project's success. I'd also like to thank Christopher Dragert, Christian Lavoie and Gregory Prokopski for their help reviewing this thesis.

Not to be forgotten are McGill University and the School of Computer Science, for providing me with an environment where I could learn and grow, both as a computer scientist and as a person. For that, I am very grateful. I would also like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC), Le Fonds québécois de la recherche sur la nature et les technologies (FQRNT), and Quazal, for providing the funds that made this research possible.

Finally, I would like to thank my friends and family for their patience and help

throughout the years. Surrounding yourself with good people and having their continued support makes the impossible possible. A special thanks goes out to my partner in crime, her unwavering faith in me was essential to the final leg of the race.

Contents

Abstract **ii**

Résumé **iii**

Acknowledgments **iv**

Contents **vi**

List of Figures **xiv**

List of Tables **xvii**

List of Algorithms **xviii**

1 Introduction **1**

1.1 Challenges of Massively Multiplayer Online Games 2

1.1.1 Scalability 2

1.1.2 Consistency 3

1.1.3 Continuity 5

1.1.4 Enjoyability 5

1.2 Contributions 6

1.2.1 Journey, a Unified MMOG Framework 7

1.2.2 Validation of Journey in a Non-Simulated Setting 9

1.3 Section Breakdown 10

2 Related work on Network Virtual Environment **12**
2.1 Frameworks . . . 12
2.1.1 Colyseus . . . 14
2.2 Consistency and Latency . . . 14
2.3 Partitioning and Load Balancing . . . 15
2.4 Fault Tolerance . . . 17
2.4.1 Election and Consensus Problem . . . 18
2.5 Trust . . . 19
2.6 Games and Experimentation . . . 19

3 A look inside Massively Multiplayer Online Games **21**
3.1 Revenue Model . . . 22
3.1.1 Content . . . 24
3.2 MMO Archetypes . . . 24
3.2.1 Sharded Universe . . . 24
3.2.2 Single Universe . . . 25
3.2.3 Hybrid Universe . . . 26
3.3 Technologies . . . 27
3.3.1 Bigworld Technologies . . . 27
3.3.2 Multiverse . . . 28
3.3.3 Project Darkstar . . . 29
3.4 A Closer Look at Eve Online . . . 31
3.4.1 Architecture . . . 31
3.4.2 Backend Bottleneck . . . 32
3.4.3 Node Bottleneck . . . 33

I Building Blocks **34**

4 Replicated Objects **36**
4.1 Quazal's Duplicated Objects / Net-Z . . . 37
4.1.1 Data Definition Language . . . 37

4.1.2 Other Features . . . 39
4.2 Eterna . . . 40
4.2.1 Duplication Space . . . 40
4.2.2 Cells . . . 41
4.3 Replicated Objects . . . 42
4.3.1 Masters and Duplicas . . . 43
4.3.2 Replication Spaces . . . 46
4.3.3 Interest Management . . . 46

5 Obstacle Aware Partitioning 50
5.1 Why do we Partition a Map? . . . 51
5.1.1 Polygon Triangulation and the Obstacle Map . . . 51
5.1.2 Partitioning in Journey . . . 54
5.1.3 Characteristics of a Good Triangulation . . . 54
5.2 Improving the Obstacle Map . . . 55
5.2.1 Dealing with Small Isolated Objects . . . 56
5.2.2 Transforming Walls into Lines . . . 57
5.2.3 Eliminating Very Small Objects . . . 57
5.2.4 Merging Points Close to Each Other . . . 58
5.2.5 Merging Small Overlapping Objects . . . 59
5.2.6 Eliminating Flat Edge Triangles . . . 60
5.2.7 Order is Important . . . 61
5.3 Experiments . . . 62
5.3.1 Quality Metrics . . . 62
5.3.2 Quality Metric Experiments . . . 63
5.3.3 Interest Management . . . 67

6 Extending Remote Procedure Calls (RPC) with Proxies 69
6.1 Existing RPC Infrastructure . . . 70
6.1.1 Open Network Computing (ONC) RPC . . . 71
6.1.2 Common Object Request Broker Architecture (CORBA) . . . 71

6.1.3 DCOM: Distributed Component Object Model 72
6.1.4 Java RMI: Remote Method Invocation 73
6.1.5 Quazal NetZ 74
6.1.6 Other RPC systems 75
6.2 Architecture of the Journey RPC System 75
6.2.1 Using the Journey RPC System 76
6.3 Implementation 78
6.3.1 Asynchronous 78
6.3.2 Proxy Generator 79

II Journey **81**

7 A Unified Approach to Load Balancing, Fault Tolerance and Cheat Detection using Trust and Game Replicas **83**
7.1 Unified Approach 83
7.2 Trust 86
7.2.1 Levels of Trust 86
7.2.2 Acquiring Trust through Time 87
7.2.3 Acquiring Trust through Capacity 87
7.2.4 Acquiring Trust through Honesty 88
7.2.5 Trust Across Sessions 89

8 Load Balancing by Dynamic Adjustment of Cells based over Obstacle-Aware Partitioning **90**
8.1 Load 91
8.1.1 Physical Load 92
8.1.2 Logical Load 92
8.1.3 Logical Load Model 92
8.2 Dealing with Load 93
8.2.1 Dynamic Partitioning 94
8.3 Load Balancing 97

8.3.1 Burst Migration 97
8.3.2 Load Sharing Master Cells 98
8.3.3 Which Tiles to Migrate? 99
8.3.4 Load Sharing Master Objects 101
8.3.5 When a Trusted Node Joins the System 103
8.3.6 Leaving the System 105

9 Fault Tolerance and Cheat Detection within a Replicated Environment 106

9.1 Consistency Model 107
9.1.1 Possible Faults 108
9.1.2 Trust 108
9.1.3 Service Guarantee 109
9.2 Fault Detectors 110
9.3 Faults Handlers 111
9.3.1 Default Recovery of Object Masters 112
9.3.2 Cyclic Recovery of Cell Masters 113
9.3.3 Dealing with Inconsistent Replicas 115
9.3.4 Fault-Tolerant Burst Migration 116
9.4 Auditing 117
9.4.1 What and When to Audit? 117
9.4.2 State History 118
9.4.3 How to Audit 119
9.4.4 The Auditor 120
9.4.5 Dealing with Good and Bad Behaviour 121
9.4.6 Dealing with Cheaters 121

III Mammoth 123

10 The Story of Mammoth 125

10.1 Summer 2005, the First summer 125

10.2 Fall 2005 and Winter 2006 . . . 127
10.3 Summer of Code 2006 . . . 128
10.4 Fall 2006 and Winter 2007 . . . 130
10.5 Summer of Code 2007 . . . 131
10.6 Fall 2007 and Winter 2008 . . . 133
10.7 Summer of Code 2008 . . . 135
10.8 Fall 2008 and Winter 2009 . . . 137
10.9 Fall 2009 and Winter 2010 . . . 138

11 The Architecture of Mammoth 139
11.1 Engines . . . 140
11.1.1 World Engine . . . 140
11.1.2 Graphics Engine . . . 141
11.1.3 Physics Engine . . . 143
11.1.4 Replication Engine . . . 143
11.1.5 Network Engine . . . 144
11.2 Managers . . . 146
11.2.1 Pathfinding Manager . . . 146
11.2.2 NPC Manager . . . 146
11.2.3 Persistence Manager . . . 148
11.3 Implementation . . . 149
11.3.1 Interfaces . . . 149
11.3.2 Listeners . . . 150
11.3.3 XML . . . 151
11.4 Services . . . 153
11.5 Evolving Architecture Example . . . 154

12 Implementing Journey in Mammoth 157
12.1 Implementing Trust . . . 157
12.2 Implementing Load Balancing . . . 158
12.2.1 Migration . . . 158

12.2.2 Load Calculation . . . 159
12.2.3 Rules for Load Balancing . . . 159
12.3 Implementing Fault Tolerance . . . 160
12.3.1 Fault Detector . . . 160
12.3.2 Fault Handlers . . . 161
12.4 Implementing Auditing . . . 161

13 Experiments **163**
13.1 Validation Techniques . . . 163
13.1.1 Role of AI in Games . . . 164
13.1.2 AI for Testing . . . 164
13.2 Experimental Setup . . . 165
13.2.1 Player Behaviour . . . 165
13.2.2 Faulty Behaviour . . . 166
13.2.3 Network Model . . . 167
13.2.4 Physical and Logical Setup . . . 167
13.3 Experiments . . . 168
13.3.1 Load Testing . . . 168
13.3.2 Fault Tolerance . . . 181
13.3.3 Cheating Players . . . 184
13.3.4 Load Balancing, Cheating and Auditing Combined . . . 188

IV Conclusion **190**

14 Summary of Work **191**

15 Future Work **194**
15.1 Improving the Building Blocks of Journey . . . 194
15.1.1 Remote Procedure Calls . . . 194
15.1.2 Triangle-Based Partitioning . . . 195
15.1.3 Network Engine . . . 196

15.2 Improving Journey 196
15.3 Trust Service 196
15.4 Load Balancing and Migration 197
15.5 Fault Tolerance and Auditing 198

Appendices

A Appendix A : Algorithms **199**

Bibliography **204**

List of Figures

1.1 Example of possible inconsistencies in the view of three clients. 4

3.1 Bigworld Architecture 27
3.2 Multiverse Architecture 29
3.3 Darkstar Architecture 30

4.1 Duplication Space, as defined in Eterna. 41
4.2 Duplication Space with Cells, as defined in Eterna. 42
4.3 A simple tomato GameObject. 43
4.4 Two instances of Tomatoe, α and β, replicated over three nodes, A, B, and C. 44
4.5 Example of circular aura interest. 47
4.6 Example of interest based on triangulization. 47
4.7 Using circular interest, difference between view (grey) and interest (red). 48
4.8 Using triangle interest, difference between view (grey) and interest (red). 49

5.1 Example of a trivial game map. Players are drawn in red, obstacle in green. 52
5.2 Example of a game map divided by a quad tree. 53
5.3 Example of an obstacle map. 53
5.4 Example of a game map partitioned using triangles. 54
5.5 Effect on triangulation of transforming small isolated objects into a line. 56
5.6 Effect on triangulization of transforming rectangular wall objects into a line. 57

5.7 Effect of removing small isolated objects on triangulation. 58
5.8 Merging nearby points can greatly simplify the obstacle mesh. 59
5.9 Merging overlapping objects greatly simplifies the obstacle mesh. . . . 60
5.10 Not using the edges of the world as constraints allows the creation of a higher quality mesh. 61
5.11 Town19-4 map 64
5.12 Town20-2 map 64
5.13 Network Traffic (in/out) at Hub during IM tests. 67

6.1 The Architecture of the RPC system 76

7.1 Interactions of Components in Unified Architecture. 84
7.2 Trust level increasing and decreasing over time. 88

8.1 Space partitioned using rectangles. 95
8.2 Partition split in the middle of a hotspot. 96
8.3 Experiments on load distribution algorithm. 100
8.4 Demonstration of shrinking a cell. 102
8.5 Demonstration of load sharing tiles with four level 2 nodes. 104

9.1 Cyclic recovery of 2 orphaned cells. 114
9.2 Example of Remote Method Executing with and without Auditing. . 118

10.1 Mammoth Client in 2005 127
10.2 Wall drawing tool, developed by Yannick Thiel 129
10.3 Mammoth Client in 2006 130
10.4 Mammoth Client in 2007 132
10.5 Content Editor in June 2007 134
10.6 Render of Redpath Museum as drawn by Édouard 136
10.7 Mammoth Client in 2008 137

11.1 Components of the Mammoth Framework 140
11.2 UML Diagram of the WorldEngine and WorldObject 142

11.3 Different layers for the Graphic Engine 143
11.4 Mammoth's current Replication Engine 145
11.5 Mammoth's current PathFinding Manager 147
11.6 Mammoth's NPC Manager. 148
11.7 Mammoth's initial network architecture 155
11.8 Mammoth's current network architecture 156

13.1 RPC Time for Clients in Load Balanced Environment. 171
13.2 CPU Usage on First Server (Halo) in Load Balanced Environment. . 172
13.3 Load on First Server (Halo) in Load Balanced Environment. 174
13.4 Total Network Throughput in Load Balanced Environment. 176
13.5 Load of Servers in Flocking Situation. 177
13.6 CPU Usage of Servers in Flocking Situation. 177
13.7 Distributions of cells during flocking experiment in a 2 servers scenario. 178
13.8 Distributions of cells during flocking experiment in a 4 servers scenario. 179
13.9 Players flocking to the left side of the map. 180
13.10Effects on Network Throughput of Fault Tolerance System. 182
13.11Effect of the loss of a server. 183
13.12Time needed to detect cheating players using Auditing. 185
13.13CPU Usage on main server (Halo) using Auditing. 185
13.14Message Throughput in System using Auditing. 186
13.15Measures taken on Unified Framework. 188

List of Tables

2.1 Maximum acceptable latency depending on type of game. 15

5.1 Characteristics of Town20-2 and Town19-4 63
5.2 Results of Triangulation Experiments. 65

13.1 Scenarios used for Load Balancing Experiments. 169

A.1 Letter conventions for algorithms 199

List of Algorithms

A.1 Burst Migration 200
A.2 Fault Tolerant Burst Migration 201
A.3 Recovery of Lost Master Object 202
A.4 Recovery of Lost Master Cell 202
A.5 Audit of Method Call 203

Chapter 1
Introduction

In the last decade, the video game industry has shown unparalleled growth, both in revenue and in development complexity. With the advent of the Internet, multiplayer and massively multiplayer games have become more and more popular. Compared to a traditional multiplayer game, in which usually up to 16 players [Hen01] play a relatively short-lived game, massively multiplayer games (MMOGs) offer the possibility for thousands of players to play together in a persistent world.

Properly implementing an MMOG requires developers to deal with many issues, such as scalability, reliability and cheat prevention. However, these issues can be difficult to solve given that MMOGs are distributed applications running over unreliable networks with high latency and limited throughput. In addition, these types of games can often be found on platforms with limited computing power, such as consoles. Finally, developers cannot assume that every client is an honest participant, given that cheating is a rampant problem in most popular multiplayer games.

Solutions for addressing these obstacles individually can already be found in the literature. For example, many articles can be found for dealing with consistency problems related to high latency. However, to our knowledge, none of them take into consideration related issues that change the context, such at the limitation in computing power and cheating.

Nonetheless, proper analysis of the literature has revealed a surprising amount of common elements between solutions to the above mentioned problems that are based

on *distributed objects*. This thesis demonstrates that these distributed objects can not only be used as a basis for addressing each of the problems individually, but also as a unifying framework for addressing them collectively. When compared to solutions dealing with these problem separately, not only does the unifying framework result in a simpler architecture, but also provides considerably more functionalities. These functionalities results from the fact that common tasks found in multiple solutions need to be executed only once in the unified architecture. As a result, the proposed framework, *Journey*, also consumes less resources, given that data structures can be shared across different solutions.

Mammoth [KVK+09], a massively multiplayer research framework provides an interesting experimental platform for implementing and testing the proposed solution. It provides a modular architecture where different components, such as the network engine, the replication engine, or interest management, can easily be replaced. Mammoth also offers a modular and flexible infrastructure for the definition of non-player characters (NPC) with behaviour controlled by complex artificial intelligence algorithms. These NPCs allow for the creation of realistic experimental conditions.

1.1 Challenges of Massively Multiplayer Online Games

Some of the concerns typically associated with massively multiplayer games can be broken down into four categories: scalability, consistency, continuity and enjoyability. This section presents these concerns and describes some of the solutions that have been used to address these issues.

1.1.1 Scalability

The biggest challenge in massively multiplayer games is scalability: the aim is to allow as many players as possible to play together in the *same* virtual world. Typically, the number of concurrent players in an MMOG is in the thousands. The machines of the players can be located anywhere on the world, connected to the Internet. As a result, the quality of the network connection to individual nodes varies: some connections

exhibit higher latency than others, meaning that it takes more time for a message to reach its destination. Bandwidth, the maximum throughput of data to and from a given node, is also limited, and varies depending on the quality of the connection. Finally, any one machine on the network has itself limited processing power and memory. On the other hand, with each player that joins the game, a new machine is added to the game, and hence the total available processing power and memory increases (from now on we will call each machine participating in the game simply a *node*).

The most common solution to scalability is *sharding*, as described in chapter 3. When using shards, the deployers of a game can control the maximum number of users that can be connected to a server at any given time. This makes it easier to estimate the maximum number of users that can be found in one area. Another solution is to use *instancing*, first setting a maximum number of players that can be found in a location, and then spawning new instances of that location each time the maximum population is reached. The problem with both of these solutions is that they separate the player-base. As a result, it is impossible for friends to play online with each other if they happen to be assigned to different shards / instances of the virtual world, even if they both play the same game at the same time and move to the same location in the virtual game world.

1.1.2 Consistency

Massively multiplayer games are complex distributed systems. Each player interacts with the game in real-time, and therefore his machine must know about the state of the game world, at least of that part of the world that is somehow visible to the player. Due to the network latency problem, this game state can unfortunately never be 100% up to date, since the world is constantly concurrently modified by other players. The challenge in MMOGs is to nevertheless provide a consistent view of the virtual world to the players, or provide means to tolerate inconsistencies so that they do not negatively affect the game play.

Luckily, existing multiplayer games show that perfect consistency is not always

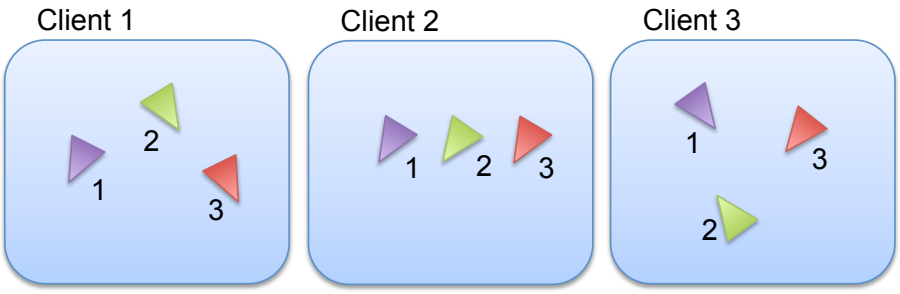


Figure 1.1: Example of possible inconsistencies in the view of three clients.

needed, depending on the types of the game and the actions that are taken by the players. For instance, figure 1.1, the three players all see different positions for their surrounding friends. However, the fact remains that all three players know that they are in the vicinity of each other.

This slight inconsistency can be tolerated with looser rules in the game-play. For example, a sword might require an attacker to be standing about 3 to 4 feet from his opponent. However, to allow looser consistency, a player might be able to attack his foe at 7 or even 10 feet. Let's illustrate the benefit of loosening this constraint with an example. Players A, B and C are out hunting zombies with swords. Player A decides to attack zombie X which he sees at 4 feet from him. Player B sees the distance between A and X as 9 feet, while player C sees this distance as 2 feet. However, the attack is considered valid because the server, which would be the centre of authority in this example, sees this distance as 5 feet, well within the attack threshold. Had the attack threshold been smaller, say 7 feet, player B might be confused by the apparently invalid successful attack.

As such, the definition of loose consistency must be looked at on a case-by-case basis, depending on the game. The consistency requirement for a game can vary greatly depending on the type of action taken. For example, movement might allow for looser consistency, but some actions, such as trading items or simply picking them up from the ground, requires perfect or almost perfect consistency.

1.1.3 Continuity

In massively multiplayer games, players can acquire and improve skills, pick up items and put them into their inventory, acquire rare objects, complete quests, amass fortunes, etc. The state of the game world is constantly evolving as players interact in the game. However, it is the game world itself that is persistent; it continues to change even when players are logged off. These games are meant to be played over a long period of time, with users spending several months or years playing with a single character. Thus, any loss of character progression is unacceptable to a player. Mechanisms to dynamically and constantly record the state of the game are critical in insuring the persistence of the world. However, this can represent a massive computational load. For example, Guild Wars requires between 1500 and 2500 transactions per second on average during the day, going up to 5000 transactions per second during peak period [DBM07].

To decrease the computational load, not all information about a player needs to be permanently recorded to stable storage. For example, in certain MMOGs, the endurance of a character is represented as hit points. That character has a maximum hit point score, which is the highest number of hit points he can have at a given time. He also has a current hit point score, which represents the current health of his character. Maximum hit points only increase through character progression, when a character changes level for example. However, current hit points continuously change, as the player takes damages and receives healing. For this reason, some games only record the maximum hit points to stable storage, restoring a player to full hit points every time he logs on. This decision has a tremendous effect on the game-play of the game, given that players can recharge their energy levels by quitting and returning to the game. However, the gain in performance is important given that hit points constantly change, but would not need to be constantly updated in the stable storage.

1.1.4 Enjoyability

The are many technical factors that can adversely affect the enjoyability of a game. Given that the revenue source of several massively multiplayer online game is based

on subscriptions or micro-transactions (see section 3.1), the loss of players can be damaging to the game's viability. A common technical problem is network latency, or lag. Delays in network transmissions or in processing requests are seen as slowdown to players, and are considered seriously irritating. Inconsistencies in the player views can also occur, causing additional irritations. In addition, cheating when present within a competitive multiplayer game, can quickly be detrimental to non-cheating players, as they see the cheating players as having an unfair advantage.

The solutions to hiding problems with latency are pretty similar to dealing with inconsistencies: loosening the gameplay rules so that timing problems are not noticeable. Since most players already have inconsistent views, a small amount of latency is not a problem. In addition, the effects of long latency can be diminished as long as players are not penalized too much for things that might happen outside their control (accidental death of the character or disconnect). Indeed, players have shown themselves to be very tolerant of latency, as long as it does not have a permanent effect in the game. One example is perma-games, where characters are deleted when they die once. Losing a character because of a high-latency situation would be unacceptable.

Dealing with cheating players is a much more complicated issue. Technical solutions to deal with cheaters are usually very complex and require some time to safely implement. Several game companies prefer to use employees titled *game masters* to police the game environment. By being in the game themselves, these game masters typically have a very fast response rate, they are able to discover exploits as they are being developed by the game population. The classic behaviour of a cheater in an MMOG is very similar to an evil villain in a North American movie, they will like to brag to others about their exploits. Working undercover as players, game masters are well suited to discover these exploits and shut down the cheaters.

1.2 Contributions

The contributions of this work can be broken down into two major topics: Journey itself and its validation in a non-simulated setting.

1.2.1 Journey, a Unified MMOG Framework

- Journey, a middleware for the development of shared virtual spaces

Journey is a middleware that greatly facilitates the development of shared virtual spaces or massively multiplayer online games. It proposes a game architecture based on replicated objects to provide a scalable and reliable way to distribute game state to all players. The various highlights of Journey are listed below:

- A replicated object system that distributes game objects transparently across multiple nodes.

To use Journey, the game developer needs to encapsulate all relevant game state in objects. Once this is done, Journey automatically takes care of distributing the game state to all the players. This is done on a need-to-know basis using interest management techniques: only nodes that that need to know about the state stored in a particular replicated object will receive a copy of it.

One of the copies of the object is designated the master replica. Modifications to the game state are executed sequentially on the master replica and the resulting state change is automatically broadcasted to all replicas. In order to modify the game state, the game developer only needs to call the appropriate update method on the replicated object without needing to worry about contacting the master replica. Journey transparently takes care of calling the master replica (if necessary) using a proxy-based remote procedure call system.

- Cell-based load balancing that automatically migrates game objects from node to node to balance network and processing load.

Journey splits a virtual world into several cells based on an obstacle-aware, triangular partitioning. Management of a cell, which includes calculation of interest regions for all players and game objects currently located within the cell, is assigned to a node. If a node is overloaded during the game, Journey automatically migrates any master replicas that the node may contain to other nodes to decrease load. If

the overloaded node is managing a cell, Journey automatically shrinks the cell by reassigning triangles from the border of the cell to the neighbouring cell. Using this technique, Journey can tolerate flocking behaviour, i.e. situations where many players gathers in a small part of the virtual world.

- Built-in fault tolerance capable of tolerating node failures.

Thanks to the replicated object architecture, Journey can tolerate any number of player node failures by re-electing a new master replica. The fault tolerance algorithms Journey uses can also recover automatically from crashes of nodes that are currently managing cells, provided that no other nodes fail until the cell has been assigned to another node.

- A peer auditing system that detects cheating nodes.

The load balancing component of Journey can decide to migrate master replicas even to player nodes. In order to prevent cheating, Journey can be configured to periodically audit player nodes in order to verify that they do not update the game state of their master replicas in an unfair way. This is done by executing a small percentage of remote method calls not only on the master replica of a game object, but as well on a trusted node that has a replica of the game object. If the outcomes of the calls are not identical, then it is possible that the player is cheating.

Other minor contributions that resulted from working on Journey are:

- A proxy generator that enables the automated creation of remotely executable methods for replicated objects.

Replicated objects must be augmented in a certain fashion to allow methods to be remotely executed. Although several augmentation techniques were explored, a proxy-based approach was found to be most efficient and practical. As such, a proxy generator was designed and implemented that analyzes the objects to be replicated and generates the appropriate proxy, which provides the remote execution functionalities.

- A distributed trust system.

Journey keeps track of the behaviour of nodes in the system. It then assigns a trust value to each node based on the node's reliability and history of failed audits. This trust value is used by the load balancing and fault tolerance components of Journey in order to determine where to migrate or recover master replicas and cells. Exceptional well trusted nodes can be used as auditors, and validate the work of other nodes.

- Multiple triangulation optimization algorithms.

Since the quality of the obstacle-aware triangularization used to partition the shared virtual world has a high impact on the performance of Journey, several optimization algorithms are proposed that improve the triangularization, which in turn improves the shape of cells.

1.2.2 Validation of Journey in a Non-Simulated Setting

- Supervision of the design and implementation of Mammoth, a massively multiplayer research framework.

A key item in the validation of Journey was the development of Mammoth, a massively multiplayer research framework. As one of the founding members of Mammoth and the lead architect for 5 years, I shaped the development of the framework. Over 50 students have contributed to the framework, leading to several publications, graduate thesis and honor's projects.

- Integration of Journey in Mammoth.

Although Mammoth was developed as a modular framework, significant effort was required to deploy Journey inside Mammoth. Most of the components mentioned above needed to written from scratch, as they did not exist in Mammoth prior to this research. In addition, extensive testing was required to ensure the proper integration and that the experimental results were valid.

- Implementation of a way-point non-player character (NPC) to generate player-like load.

Before Journey, the only reliable non-player character in Mammoth was the random wanderer, which wandered randomly across the game map. Even though this NPC was ideal to generate movement across a game map, it generated very little load on the system. Other experimental NPCs were not suitable for running experiments with Journey. As such, a way point system was added to Mammoth, where points of interest can be tagged and linked together. A waypoint NPC was designed to travel from waypoint to waypoint, following the established links.

- Creation of an experimental environment with hundreds of NPCs.

A key point in experimenting with Journey is demonstrating its ability to handle a large number of nodes. As such, several shell scripts were designed and implemented to remotely manage (start, stop, etc.) NPC characters running on hundreds of computers. The scripts are currently being used by several other students of the Mammoth project to run experiments.

- Evaluation of Journey by measuring CPU usage, memory usage and network throughput in over a dozen of experiments.

Several tools were required to gather metrics on the computers running Journey experiments. In addition, given the distributed nature of these experiments, a system had to be devised to gather and analyze this data.

1.3 Section Breakdown

This thesis is divided into four parts, in addition to the introduction chapters. Chapters 2 and 3 provide background information in the domains of Network Virtual Environment and Massively Multiplayer Online Games respectively.

The first part provides information on the building blocks of Journey, namely Replicated Objects in chapter 4, Obstacle Aware Partitioning in chapter 5 and Extending Remote Procedure Calls with Proxies in chapter 6.

The second part introduces Journey, with a description of the motivation and the trust-based approach in chapter 7. Solutions provided to deal with load issues are presented in chapter 8, while issues relating to fault-tolerance and cheating are covered in chapter 9.

The third part is dedicated to Mammoth, the implementation of Journey, and the experiments done to validate the work done in Journey. Chapter 10 introduces Mammoth, discussing its initial goals and development history. A detailed description of Mammoth's architecture and implementation can be found in chapter 11. The integration of Journey is described in chapter 12. Finally, several series of experiments are presented in chapter 13, illustrating the performance impact of the proposed system.

The final part concludes this work, providing a summary of the work in chapter 14 and some possible future work in chapter 15.

Chapter 2
Related work on Network Virtual Environment

The idea of creating a distributed game infrastructure is decades old, with works such as [DG99] formally introducing the concept. This particular work introduces Mimaze, a distributed game that leverages multicasting for communication. The paper introduces many key issues, such as state inconsistencies and the possibility of cheating. However, Mimaze did not deal with scalability issues, restricting itself to 25 players.

Since then, there has been no shortage of work proposing distributed architectures for games on a massive scale [CXTL02,KLXH04,IHK04,BECM05,AT06,BPS06, FTT07,CYB+07,ASdO09] . Several of these ideas are based either on Pastry [RD01], which describes a communication system for a large scale peer-to-peer system, or Mercury [BRS02], which proposes a publish/subscribe system that could be used for large-scale games.

2.1 Frameworks

A key concern in distributed game frameworks is scalability, dealing with an increasing number of players. This concern is well introduced in [CXTL02], where scaling to multiple servers in a client/server architecture is achieved by partitioning the game

world. However, developing a multi-server application introduces some important synchronization and consistency concerns. The Matrix middleware [BECM05] allows for dynamic partitioning of the world, while hiding most of the complexities from the programmers. The world is partitioned into non-overlapping zones, and each game server is responsible for a zone, plus its overlapping interest surroundings. The key to the proposed solution is the matrix servers whose main duties include routing messages between clients and servers and monitoring load on servers. Messages targeting a specific zone are routed to the appropriate game server, and forwarded to all servers hosting neighbouring zones. If a game server is overloaded, a matrix server will detect it, find an empty server and assign it half of the surface of the overloaded server.

A true peer-to-peer distribution of the game workload is proposed by many as an efficient alternative to traditional client/server architectures [KLXH04, IHK04, FTT07, CYB+07]. However, to be used in a game environment, a peer-to-peer system must provide performance, availability and security [KLXH04]. An interesting approach is presented by [IHK04] where the world is divided into zones and stored in a distributed hash table (DHT). Information on the content of zones is only distributed to specific clients on request, and responsibility for a zone is given to the first client requesting it. At that time, the zone is removed from the DHT and only returned to it once the client no longer needs access to that zone. This removes any unnecessary load on the DHT. An alternative approach is proposed by the Hydra architecture [CYB+07], which also divides the world into disjoint regions. Each of these regions is considered a slice, and each of these slices is assigned to a server and a backup server. Game logic must be deterministic, as events are executed both on the primary and backup server. Thus, if the primary fails, the backup server can continue managing the slice. Synchronization is done through logical clock ticks, and the whole thing is managed by a rendez-vous server, although that part can also be distributed with a DHT. Alternatively, Mediator [FTT07] innovates on the mentioned frameworks by introducing the notion of a reputation system, rewarding strong peers with more work and responsibility. Work is coordinated by a set of mediators, each with different responsibilities. All game actions are translated as jobs and are assigned to different peers.

2.1.1 Colyseus

The Colyseus [BPS06] framework is worth particular attention because of the numerous similarities between it and the proposed framework. Although the framework in designed for first-person shooting games (FPS), it has some important similarities with our proposed architecture. The framework uses a single-copy consistency model, where objects can be either mutable or immutable. Immutable objects are globally replicated (i.e. they are loaded locally on each participants). Mutable objects are store in the global object store and have a single primary (authoritative) copy that resides on exactly one node. Changes to a mutable object are serialized to the primary copy. The replicas are then updated in an application dependent fashion. To reduce the bandwidth requirement, Colyseus uses area-of-interest filtering (sending only what is interesting) and delta encoding (sending only the changes in an object).

Interest management is done through a pub/sub system (which is implemented in Mercury), using a system of range-query subscriptions. Management of the replicas (sync, replication creation, replication deletion) is the responsibility of the replication management component. Nodes must periodically register their interest with the node hosting the primary copy of an object to keep receiving replica updates.

A key difference in Colyseus is the introduction of the *think* function of an object, a function to be invoked at every frame/clock tick to determine the actions of that object. This think function is actually the main source of load in the system. In addition, the execution of the think action might require access to objects currently stored on other nodes. Thus, the primary objects are also responsible for predicting which objects might be needed and pre-fetching them.

2.2 Consistency and Latency

A fundamental challenge faced by all distributed systems is consistency [LLL04] and latency [Hen01]. This is especially true in a video game, where inconsistencies due to latency can cause serious impacts to game play [Hen01, HSPA09].

A classic solution to this problem is to synchronize everything using a global clock

Warcraft 3	several seconds [CC06]
NHL Madden Football	500 ms [NC04]
Virtual RC Racing	150 ms [PW02]
Unreal Tournament 2003	60ms [QML+04], 150 ms [BCL+04]

Table 2.1: Maximum acceptable latency depending on type of game.

and time slots [YMYI05]. However, one interesting result obtained from [DG99] suggest that only 65% of movement data is needed for a realistic game experience. This is because games exist in a continuous space, as opposed to a discrete space commonly used in databases and network protocols. Thus, strict traditional consistency models do not apply to games. [LLL04] proposes a more relaxed consistency model, where updates are applied as long as they are delivered within a fixed time threshold. A similar idea is explored by [FR05] where events can be correlated and obsolete messages are discarded. [Haw08] explores the idea of consistency in multiple distributed server environments while [FGW06] proposes a model where consistency concerns can be seperated from the application logic.

It should be noted that the maximum latencies that can be tolerated by a player without impacting too negatively on his gaming experience varies depending on the type of game. Table 2.1 describes some of the commonly described maximum latency for different types of games. The IEEE standard on distributed environment (1278) [ANS93] set a much stricter standard, setting the maximum latency at 100ms.

2.3 Partitioning and Load Balancing

A key element presented in all the approaches is the partitioning of the game world. This can be particularly difficult given that the game world has dynamic memberships: players constantly join and leave the game world. A simple approach is to initially partition the game world into rectangles, further sub-dividing it as the load in a particular rectangle increases [BECM05]. An alternate approach is to group players

depending on their proximity to each other [LC02]. However, this problem is shown to be NP-complete as it maps to the subset sum problem. In addition, given that players are constantly in movement, the grouping must be constantly recalculated, creating an incredible amount of overhead. Alternatively, [CXTL02] suggests that partitioning need not be restricted to space, but can also be temporal, where elements are grouped by their update rate, or functional, where elements are grouped by role or type.

Given its property to easily deal with obstacles, triangulation can be used as an interesting alternative to grid partitioning. This property is well illustrated in [DB06] where triangles are used to significantly reduce the pathfinding search effort. In addition, [BKV06] shows how a specific constrained Delauny triangulation [She96] can be used for efficient interest management. Delauny triangulations can also be used as an efficient solution for grouping, as presented by [BA08]. This is similar to the grouping approach using Voronoi diagrams, as proposed by [HL04].

Ultimately, a static distribution of load cannot deal with players that are constantly moving across the game world. Ideally, load should be dynamically and equally distributed across all nodes. However, achieving this is a non-trivial task. Load balancing can be done at two levels [LL03], globally (one node manages all load distribution) or locally (each node manages its load with its neighbors). [CWD+05] presents a global load balancing algorithm which takes into account the position of players and tries to keep neighbouring regions together, while trying to minimize inter-server traffic. A more hybrid approach is proposed by [LS06], where a node is responsible for monitoring its own load, but the request for help is sent to a single global server. Furthermore, a key concept introduced by [DZ03] is load sharing, the ability to share load before a node is overloaded. Their data shows monitoring load and sharing it before an overload is much more efficient than dealing directly with an overload.

An alternative to dealing with load is to minimize the amount of information shared across nodes. Ideally, a node should only receive information about elements relevant to it. This concept, known as interest management, is presented by [MLS05a] as the ability to identify when objects should be interacting with each other and enable message passing between them. This work introduces the concept of interest as

a circular aura around a player. Understanding how this aura moves around with the player allows the system to predict which object might be interesting to the player. The application of interest management into a commercial game in an effort to conserve bandwidth can be found in [Cad08]. Furthermore, a comparative study of different interest management schemes [BKV06] shows that grid and triangulation based IM have a non-trivial performance advantage over traditional aura-based interest.

2.4 Fault Tolerance

The origin of dependability can be traced back to the first generation of electronic computers, which were built from unreliable components [ALR04]. Numerous practical techniques were used to improve the reliability of computation, such as triplication with voting. Over the last 50 years, these concepts were refined and formalized, giving birth to the more modern concept of fault tolerance used today.

We can define the service delivered by a system as the behaviour of the system, as perceived by the user. A service failure occurs when the delivered service deviates from the correct service. Failures can be broken down into two categories: consistent or inconsistent. Consistent failures are perceived identically by all users, and are easily reproducible. Inconsistent failures, also known as byzantine failures, are perceived differently, if at all, by different users in the system. They are also very difficult to reproduce.

The deviation in a failure is called an error, and the cause of this error is called a fault. Faults can be broken down into two categories, either intentional faults (malicious logic, intrusions) or accidental faults (physical, design, interaction). Faults can be found both at the development phase and the use phase of the software development life cycle.

Dependability is the ability to deliver a service in a trusted fashion, without failure. This encompasses several attributes, such as availability, reliability, safety, confidentiality, integrity and maintainability. Many different techniques can be used

to achieve dependability. One common technique is fault prevention, where focus is put on preventing the occurrence of faults. Another technique is fault tolerance, where the system is designed to continue providing a trusted service, even in the presence of faults.

An example of fault tolerance is N-version programming [CA78], where an algorithm that requires high dependability is implemented in N different ways. When that algorithm is required, all N versions are executed. The results are then compared and voted on, the predominant value being the correct answer. To survive m faulty executions, n implementations are needed where $n >= 2m + 1$. If the number of faulty executions is greater, the fault might not even be detected. The implementation of N version programming can be difficult, as the N versions must have identical behavior, while having radically different implementations. Implementing the voting scheme might also be particularly difficult, given that inexact voting might be required to deal with numerical deviations.

2.4.1 Election and Consensus Problem

In a peer-to-peer NVE, participating nodes might be required to make a common decision. In a reliable distributed environment, several simple consensus algorithm can be used to achieved this common decision [PSL80]. However, achieving a consensus in an unreliable environment is a much greater challenge. The complexity of this problems depends heavily on the assumptions made about the model of computation and the possible faults [Fis83]. Dealing with byzantine failure is particularly challenging, as no solution to the consensus problem exists if the communication model is asynchronous. Solutions do exist when dealing with a synchronous communication model, but they require a very high number of message rounds. Election algorithms are not helpful, as they are considered a more difficult problem then consensus [SM95]. However, [HKU] proposes an optimistic approach to voting, which considerably reduces the number of message rounds.

2.5 Trust

The increased popularity of P2P systems is threaten by the presence of malicious users. Threats can range from traitors (players using acquired trust to break the system) to moles (players helping malicious players to acquire trust) or the simple whitewashers (players connecting under new ids to shed bad reputation) [MGM06]. Trust systems reduce the risks for peers, as they can make better informed decisions about who they communicate with. These systems are broken down into three components: information gathering, scoring/ranking and incentive/punishment. First, one must determine what and how much data should be gathered on the interactions between peers. This is a quality vs. quantity issue, where the bias should be put on the quantity. Then, the data must be processed and transformed into a trust score [GV08]. This score could be binary (trusted/untrusted), discrete (0 to 100) or even continuous (between 0.0 and 1.0). The third step is to take action based on this score. In file sharing networks, an incentive is to grant increased speed to a node, a punishment is to blacklist it.

2.6 Games and Experimentation

Unfortunately, little academic work exists on commercial games themselves. Cooperation between academicia and the game industry is a fairly new concept, and very little data exists on how people play games. This information is considered confidential, as it is a very valuable commodity in the industry. One exception is the May 2003 to March 2006 Eve Online data set, which was released to a group of researchers. Their work [FBS07] demonstrates that given enough data, the workloads generated by players follow a pattern and can, up to a certain point, be predicted.

Independent studies on games can also be found. For example, [Hen01] provides some insight into what can be considered acceptable latency on a HalfLife-2 [Ent09] multiplayer game server. More importantly, [PG07] provides some very valuable insight into the behavior of players on a World of Warcraft [Ent09] game server. For example, they measured that server population peaked at around 4000 players, and

that 40% of play sessions last less than 10 minutes. This conclusion has a significant impact on P2P game research, as most P2P frameworks suffer from a high connection cost.

Lack of available data on existing game architectures only reinforces the importance of experimentation. However, the key problem with experiments on scalability issues is the need for hundreds, if not thousands of clients. Although simulations can be used to evaluate certain characteristics of these framework, it does not replace testing and benchmarking using a large number of clients. [RK07] proposes a solution to this by using synthetic (AI) players. This has the advantage of closely reproducing game conditions without the unpredictability of real human players. Possible behaviors for such AI players can be found in [Mat03], where actions are decided using either a greedy approach, Markov chains or hierarchical plans. The realism of such experiments could further be improved by introducing artificial network delays (lag), as described by [ZWZ+06].

Chapter 3
A look inside Massively Multiplayer Online Games

A massively multiplayer online game (MMOG) is a video game that allows a **large number** of players to **interact** in a **virtual persistent** world. Breaking down this definition,

- **large number** : Some of the more popular MMOGs have millions of subscribers, with several thousand players connected to each game node.
- **interact** : The nature of the interactions differ depending on the game, but players either are asked to compete against each other, cooperate with each other, or a combination of both. MMOGs geared towards cooperation usually have constructs such as groups, parties, guilds, or alliances to support the cooperation. Competition takes the form of races or tournaments or other forms of rankings, and winners are well rewarded.
- **virtual** : The world in which players interact is not physical: it is digital, existing only within the computers of the players and of the game provider. As such, any objects or other form of property within the game is also virtual. This concept of virtual ownership has sparked various legal battles [Mee06].
- **persistent** : When a player disconnects, the world continues to evolve without him. Once the players return, they resume at the point they left off.

MMOGs are the most commonly found NVEs on the market today. As such, a closer look at their enabling technology can provide us with a greater insight into solving key problems in NVEs.

3.1 Revenue Model

Massively Multiplayer Online Games are developed and operated by for-profit corporations. They are considered a high-risk, given the large amount of up-front investment required to develop, launch and maintain such a service [Dig09]. However, they can also be very profitable. For example, World of Warcraft [Ent09], one of the MMOGs with the largest subscription base, earned an estimated 500 million USD in 2008 [New09].

The success of a MMOG highly depends on the revenue model it adopts. These are five of the most common ones:

- **Subscription** : The first and for a long time, most profitable model, requires players to pay a monthly subscription fee. Many MMOG providers offer discounts for players purchasing more than one month at a time. A few providers have even gone to the extreme of offering a life-time membership fee, although those are usually extremely expensive.

- **Premium Subscription** : A variation of subscription, players are first invited to play the MMOG for free. However, to enhance their play experience, they are offered the option of paying a monthly subscription. The effect of this enhancement varies from one game to the other. For some games, premium subscription only gives access to new items and areas to explore. Other providers cripple the game-play experience by setting level caps until players pay for the premium content.

- **Micro-Transaction** : In this revenue model, players are also invited to play the game for free. However, to obtain certain items, players are invited to buy them from the game store using real currency. Given the small cost of items,

players are often asked to transform a certain amount of real currency into game currency (tokens, credits, etc.) which can then be used in the game store. For some games, micro-transactions are only used to purchase cosmetic and non-gameplay items. Other games require players to spend real currency for real items, such as healing potions or spell-casting material components.

- **Prepaid Card** : Not a revenue model in itself, but more of a payment method, prepaid cards address the reality that not all gamers have access to a credit card. The value of the prepaid card depends on the revenue model used by the MMOG. For example, a subscription-based MMOGs might have time cards (1-month, 120 hours, etc.) while micro-transaction-based MMOGs often load the prepaid card with a certain amount of their in-game currency (tokens, credits, etc.). Prepaid cards are more popular in Asia, where the likelihood of owning a credit card is much lower.

- **Ad-based** : A relatively new revenue model, ad-based MMOGs pay for themselves solely through in-game advertising. This advertising might have a permanent spot in the players hud, or might be only shown at specific moments (login screen, loading screen, etc). This model is often combined with premium subscription, where the premium subscription allows the removal of the adds.

It's important to note that many other less-used revenue models also exist. For example, some games ask players to buy real-world items in toy stores. These items have a key or code, which can be used to obtain powers or items in the game.

It is also not uncommon for a MMOG to exploit more than one revenue model. For example, Ragnarok Online [Inc10] was a monthly subscription-based MMOG for several years. However, in the last year, they have opened a microtransaction-based free-to-play game server where critical items can only be bought from the game store. In addition, they have given experience boosts to their paying subscribers, to reward them for their loyalty.

3.1.1 Content

Independent of the revenue model used by an MMOG, a successful MMOG needs a large number of subscribers. As such, attracting new players and keeping veteran players happy are two of the key challenges MMOG providers face. The most common solution to this problem is to continuously generate new content and add it to the game. Content is defined as any environment or component of the environment that the player can interact with. Examples of new content could be a town, a dungeon, new character classes, new opponents, new skills, etc.

The unfortunate reality is that content is one of the main expenses of developing a video game today. Statistics have shown that the number of writers, musicians and artists working on a video game far outnumber the the programmers on the team [CP09]. In addition, any new content to be added to a game must be extensively tested. The initial expense of generation of content of an MMOG is also significant, since the game world must accommodate thousands of players.

3.2 MMO Archetypes

A key difference between a massively multiplayer online game and a typical multiplayer video game is that most of the in-game logic is processed in data-centers managed by the game's developer, as opposed to a player's computer. Massive multiplayer online games can be further divided into different archetypes, depending on how the game data and players are partitioned across servers. One such classification proposes three types of archetypes: single universe, shared universe and hybrid universe.

3.2.1 Sharded Universe

The most common of these archetypes is the sharded universe, where players are distributed across identical copies of a common world. The term "shard" originates from one of the first popular MMOGs, Ultima Online [Art10], which describes each instance of a game's world (the shards) as broken pieces of a mystic gem. This story element was originally introduced in 1980 in Ultima I, and was repeated many times

in the Ultima mythos before it was actually used in Ultima Online, the MMO version of the game. Each shard is nearly identical to the others because they all originate from the same single crystal. Other MMOs use different terminology to express the same concept, e.g. World of Warcraft describes shards as "realms".

Sharded MMOGs are fairly common for four reasons:

- They require a smaller amount of content: a sharded environment can be designed to support a few thousand players.
- They are easier to scale: as more players join the game, new content is not required. The provider only needs to instantiate new shards.
- They are less expensive to create (smaller amount of content needed) and maintain (ability to start out with only a few shards).
- They are easier to deploy. Different shards can be used for players in different geographic regions. Thus, deploying the game to a new region requires the creation of new shards and does not affect existing players.

However, most sharded MMOGs have the severe limitation that players that play on different shards are unable to interact with each other. Migrating players from one shard to another is possible, but often requires administrator intervention and is offered as a billable value-added service. Typical examples of sharded MMOGs include World of Warcraft, Ragnarok Online and Age of Conan.

3.2.2 Single Universe

Some MMOGs pride themselves on their ability to maintain a single large virtual environment for all their subscribers. However, the single universe archetype is particularly difficult to implement.

- A single universe requires an immense amount of content to support the large number of players (millions in some cases).

- Single universe MMOGs are more difficult to scale, as adding more capacity can require an overhaul of the whole system.
- Single universe MMOGs are difficult to deploy in different geographical regions. For example, a player from America should be able to easily interact with a player from Asia, with as little lag as possible for both players. This most likely requires servers both in America and Asia, and a fast communication link between the two servers.

It should be noted that the single universe archetype is reserved to special categories of games, simply because of the incredibly large amount of content required. Space exploration games, like Eve Online [CCP10a], tend to fit in well with this archetype given the vastness of space. Games where players generate most of the content, such as Second Life [Lab10] also fit well into this model.

3.2.3 Hybrid Universe

MMOGs featuring a hybrid universe are very rare, because, although they share the advantages of both single and sharded universes, they also share their many disadvantages. The idea is simple: allow players to communicate as in a single universe, but shard the actions and network-intensive components of the game. This makes the design of the game significantly more complicated.

The content requirement is higher than in a sharded environment, but significantly lower than in a single environment. Scalability is possible, as load is mostly found in the "shared" part of the game. Deployment is still very difficult, as players from different geographical regions can still meet in the single universe and wish to cooperate in the "sharded" component of the game. To date, one of the only games to have successfully exploited a hybrid universe is Guild Wars [Net10].

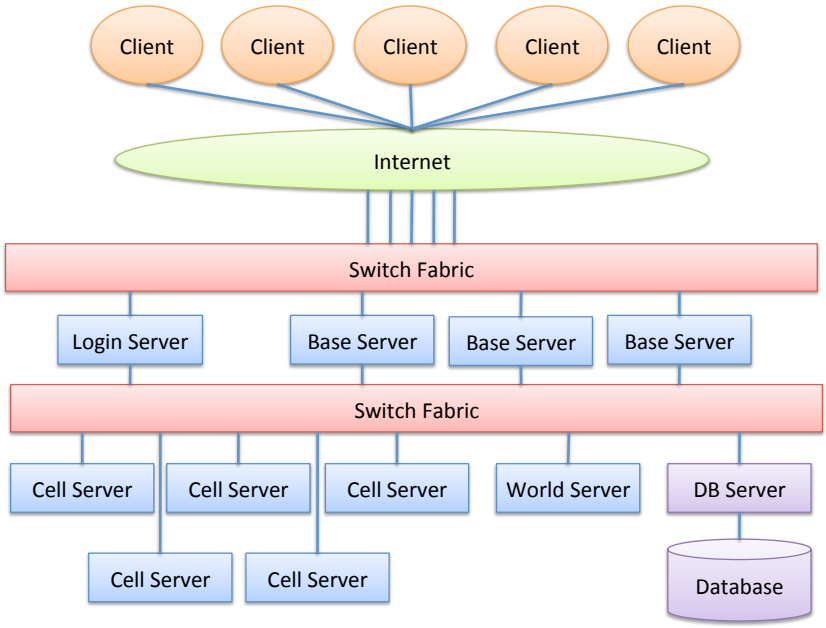


Figure 3.1: Bigworld Architecture

3.3 Technologies

Given the complexity of developing MMOG technology, it can be much more cost effective to purchase the technology from a 3rd party provider. However, not all MMOG technologies are equal. The following section introduces some of the MMOG frameworks currently available on the market.

3.3.1 Bigworld Technologies

Bigworld Technologies is one of the most mature MMOG technologies available on the market. Not only does it provide the backend infrastructure for an MMOG, but most of the tools needed to build the game itself, including the graphical 3D client and the content editor. Note that since Bigworld is a commercial product, very little information is available on its inner workings. Repeated attempts to contact them have failed. Most of the information presented in this section was gathered from promotional material, which might not properly reflect the features of the product.

Bigworld3.1 supports both the single and sharded universe archetype. It divides the world into cells, which are then assigned to cell servers. If a cell server is overloaded, cells are redistributed across other cell servers. Cell servers can even be shared with other shards or game worlds. If a cell server fails, cells are automatically redistributed to other cell servers. All of this remains transparent to the player, as communication is managed thought the base servers. Bigworld also features tight bandwidth control, through the use of level of details and data prioritization on every game object. Game objects are programmed in Python. (see figure 3.1)

3.3.2 Multiverse

Multiverse is another 3rd party MMOG toolkit, but with a very different business model. With Multiverse, the tools to develop the game are given freely to the developer. However, Multiverse handles all transactions between the customer and the developer, retaining 10% of the revenues plus transaction fees. The set of provided tools to build the game world is very complete: a terrain generator, a world editor and a model viewer. Little programming effort is required to build a game itself, mostly scripting in Python to define the game rules and behaviour of items.

Multiverse breaks down computation load into a set of servers. Events such as moving in the world, picking up items, fighting foes, saving the state of an item, detecting collisions between objects or trading between two players are all handled by different server processes (server plug-ins). These processes can be run on a single or different machines. A messaging server, using a publish/subscribe sub-system, ensures communication between the different server processes. All the server software and the various plug-ins are written in Java, and their behaviour can be customized using Java or Python code. The world state is recorded by the object manager server, which records the state in turn to any JDBC compatible database.

A multiverse world is broken down into zones, which are defined as a cube of maximum 4,000 km per side. Moving from one zone to the other requires zoning (teleporting from one zone to the other). Zones are managed by world manager servers, each of which can support a maximum of 2000 players. They are broken

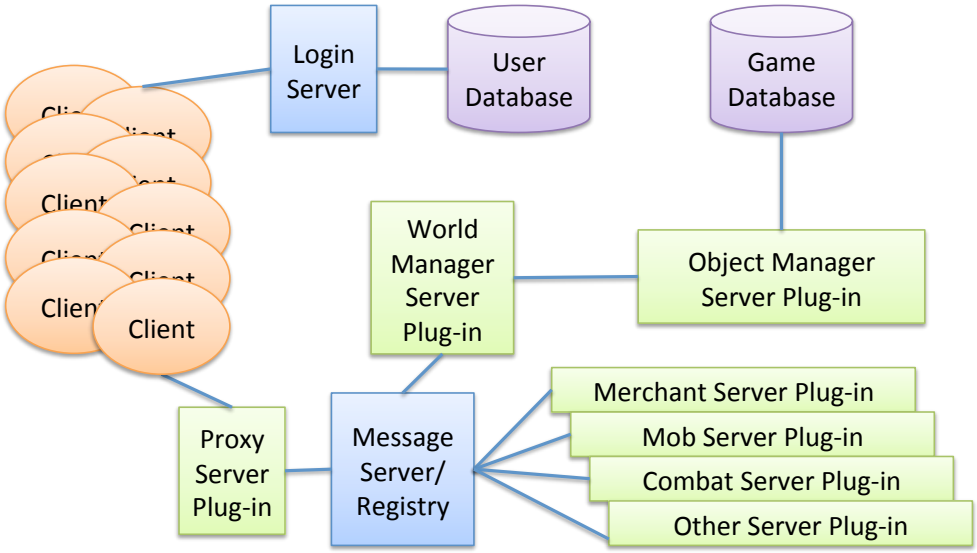


Figure 3.2: Multiverse Architecture

down using a quad-tree [FB74], the position of the break down can be manually set or determined dynamically by the load balancer. As more objects are added to a quad, the quad itself is further broken down. When a server can no longer manage the load of all its quads, it can transfer some quads to another world manager server. This is transparent to the user, as all network traffic is routed by the proxy servers (see figure 3.2).

3.3.3 Project Darkstar

Project Darkstar is an interesting variation on the 3rd party MMOG toolkits currently available on the market. Contrary to the two previously presented frameworks, Darkstar is only a communication framework. It does not provide a graphical client or a world editor. However, the model that Darkstar proposes is radically different, as it is completely zoneless and shardless. Its goal is to provide a reliable, scalable, persistent, and fault-tolerant experience while providing a transparent single-threaded event-driven programming model to the developer.

The idea with Darkstar is simple: every action/event is broken down into small

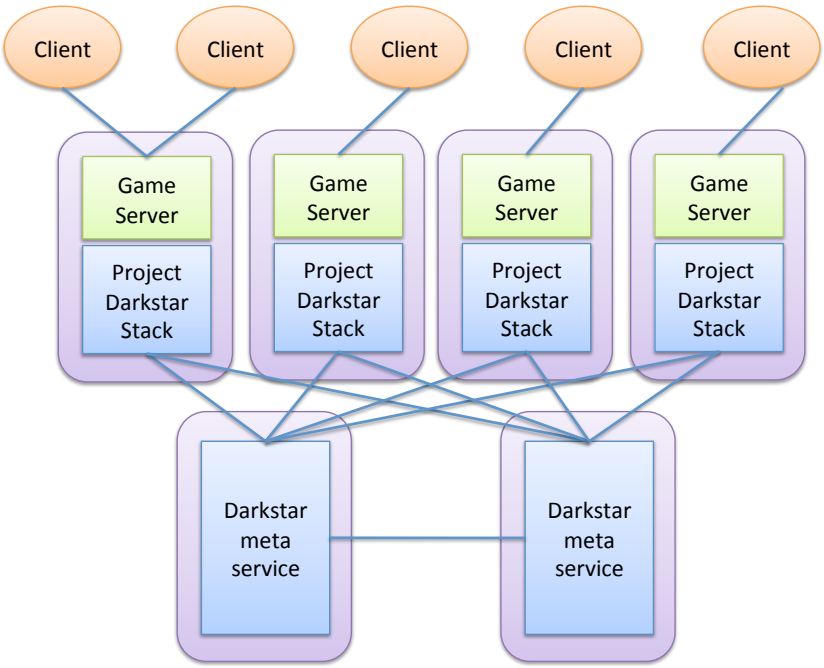


Figure 3.3: Darkstar Architecture

units. While these units can be stored and executed on any server-side node, they are grouped using an affinity scheme. This groups actions that can be executed together on the same node, because these actions will modify the same set of data. This freedom to execute a unit on any node breaks the traditional concept of zones and allows for easier load balancing. In addition, each time an object is modified by a unit, it is saved back in the object store in a transactional fashion, diminishing the risks of failures and cheating.

Each Darkstar server is composed of three managers: the task managers, the data manager and the channel manager. The task manager ensures the execution of the units. Note that although units created on a client are guaranteed to execute in order, there is no order guarantee for units created on different machines. Units must be short lived to prevent excessive locking of data. If a unit requires more than 100 milliseconds to complete, it will be terminated by the server. The Data Manager stores all the game objects, known as ManageObjects, making sure that their access is sequential and their state is correctly saved. ManageObjects do not refer to each

other directly, a ManageReference must be used to access one ManageObject from another. Finally, the channel manager creates publish/subscribe data channels, which are used to control communication between clients and servers. (see figure 3.3)

3.4 A Closer Look at Eve Online

> Eve Online is a massive multiplayer online game (MMOG) set in a science-fiction based, persistent world. Players take the role of spaceship pilots seeking fame, fortune, and adventure in a huge, complex, exciting, and sometimes hostile galaxy [CCP10b].

Originally created in 2003, Eve Online has evolved significantly over the years, both on the front end and the backend of the game. This game is academically interesting for two reasons:

- Eve Online is one of the few games to use the single universe archetype, which can be easily considered the most challenging to implement.
- CPP, the company developing Eve Online, is very open about its implementation of the game.

The following section describes some interesting architecture elements of the game, the problems faced with this architecture and how there were solved. As previously mentioned, Eve Online uses a single universe archetype. Although the maximum capacity of concurrent users is unknown, on January 4th, 2010, Eve Online reached a peak of 54,446 simultaneous users [Bre10].

3.4.1 Architecture

Eve uses a three layers system: the proxy blades, the sol blades and the database cluster [Min08].

- The proxy blades are the public interface to the game. They manage player connections and forward requests to the sol layer.

- The sol blades manage the interactions that occur in different solar systems.
- The database cluster handles the persistence layer of Eve.

All three of these layers have faced performance problems, limiting the number of players than can be found simultaneously in a given solar system. Discussions on problems with the database cluster can be found in subsection 3.4.2, while network performance problems between the proxy blades, the sol blades and the clients can be found in subsection 3.4.3.

3.4.2 Backend Bottleneck

Persistent storage in Eve Online is handled by a single monolithic SQL Server database. Although this database is distributed over several servers, it represents a significant bottleneck, especially during peak hours. In 2006, CCP was experiencing some serious delay problems: the database was unable to handle the load of 1,250 transactions per second at peak hour. At that time, it was not unusual for players to experience 20 seconds delay, a delay that is considered unacceptable in the gaming world. Traditional solutions of adding new servers, or upgrading existing servers would not provide the performance increase needed to solve this crisis. Careful analysis of the situation indicated that the problem was disk IO based.

CCP solved this problem by moving its database to a solid state disk array [CTR06]. Up until January 2009, the architecture was as follows:

> The Eve database runs on a Microsoft SQL Server 2005 Active/Standby cluster. It has two identical database servers, with the database stored over two RamSan 400s and a DS4800 fiber channel array [Min09].

The performance increase was immediate and developers were able to redirect their effort to developing new game content. The setup remained relatively unchanged, until the load on the fiber disk channel increased to an alarming level. By then, the game generated up to 2000 transactions per second. The problem was solved again with a hardware solution, upgrading the disk array to a 2TB RamSan 500.

3.4.3 Node Bottleneck

As previously mentioned, interactions between players in a given solar system are managed by a sol blade. Typically, blades are set up with two nodes, each node hosting a solar system. However, some blades hosting a high traffic solar system only have one node active in order to dedicate all their resources to that solar system.

The maximum number of players that can be found in a given system is limited by the load that can be handled by the machine hosting the solar system. This can be properly illustrated by the Jita solar system, which, before February 2009, could only handle 700 concurrent players. After some important software upgrade, Jita was able to accommodate more than 1400 players.

One of these software changes was the implementation of a stackless IO network layer [Exp08b]. Early analysis had shown that when executing a remote call to a busy server, resolution of the call was almost instantaneous, but the remote call itself would take over 1 minute to return. This pointed to a problem within the network layer. Switching over to the stackless IO network layer resulted in a considerable performance increase.

Once the IO concerns were removed, memory limitations quickly became the main concern. Given that the network layer no longer limited the number of players in a given solar system, the system would scale until it ran out of memory. This issue was solved by upgrading the Eve server software to 64bit [Exp08a], thus allowing it to address more memory.

Part I

Building Blocks

Journey is a complex piece of architecture, built on solid and proven existing concepts. These concepts, and their use in Journey are discussed in this part. First, chapter 4 describes the core technology behind Journey, replicated object. The following chapter presents how obstacle aware partitioning can increase Journey's performance. Finally, chapter 6 introduces Journey's communication model, which heavily relies on remote procedure calls using proxies.

Chapter 4
Replicated Objects

A game programmer should ideally be able to work with abstractions from the game domain. A virtual world is usually comprised of many objects, e.g. game items and players. Game developers therefore naturally apply object-oriented decomposition techniques to partition the game state.

Replicated objects are a distributed implementation of game objects: they encapsulate that part of the state of game objects that has to be distributed to players. Every node that needs access to the game state encapsulated by a replicated object possesses a local instance of that object, a duplica. One key challenge is to ensure that the state of the duplicas are kept up to date and consistent.

The first two sections of this chapter presents Quazal's Net-Z and Eterna, two commercial multiplayer game middleware implementations that served as inspiration for our own replicated objects technology, which is presented in the third section.

Note that the following chapter assumes that the reader is familiar with the notion of the observer design pattern as defined in [GHJV95]. The reader is highly encouraged to look up this information if he is unfamiliar with this design pattern or the publish/subscribe communication paradigm.

4.1 Quazal's Duplicated Objects / Net-Z

The core goal of Quazal's duplicated object technology, as found in the Net-Z product, is to create a high level abstraction that relieves developers from dealing with many of the complex issues of distributed computing. As previously mentioned, object-oriented programming is a natural fit to game development, as game objects can easily be mapped to classes. With duplicated objects, game objects are duplicated and distributed to all the workstation participating in the game. A duplicated object can have one of two states, either duplication master or duplica. When the state of a duplication master is altered, the state changes are disseminated to all other duplicas.

4.1.1 Data Definition Language

A key element to this technology is DDL, the data definition language. It allows programmers to describe game objects and how their states should be transmitted over the network. This has the advantage of introducing network concerns early into the development cycle, a common oversight in many game development projects [Gro03]. Once game objects are defined using DDL, network-enabled code can be quickly generated and is easily tweaked. DDL defines two main components: datasets and class declarations.

Datasets describe the data stored inside a duplicated object (state) and its update policy, which specifies how state updates to the duplication master are propagated to the duplicas. For example, one could decide that updates are only propagated if the dataset is significantly changed. Another update policy might specify whether updates should be sent over a reliable or unreliable communication channel.

Class declarations define the attributes and methods of game objects. Attributes are specified using datasets, while methods are directly defined in the class declaration. Methods can either be defined as remote methods calls (RMC) or actions, the difference being that only RMCs can have return values. The methods can be executed either locally (only on the local duplica), or remotely on either the duplication

master or both the duplication master and all the duplicas. This differs from traditional duplication object models, where methods are only executed on the duplication master.

The two components defined in listing 4.1 are inspired from the examples found in [Qua01]. Once the DDL for a game object is completed, a compiler generates a series of stub files. Developers only need to insert the custom game code inside the method stubs generated by the compiler.

Listing 4.1: Example of a DDL definition for a simple game

```
// DDL file
// Declaration of the datasets to be associated to duplicated
// objects
dataset Inventory {
  int item1;
  int item2;
  int item2;
} constant;

// An extrapolation filter is used to update the Position
// dataset to minimise bandwidth usage
dataset Position {
  double x;
  double y;
  double z;
} extrapolation_filter;

// Declaration of the duplicated objects. The appropriate
// datasets are associated to the objects and the action
// declarations made
doclass GameObject {
  Position m_dsPos;
  action PlayASound(int iSoundID);
```

```
};

doclass Avatar :: GameObject {
  Inventory m_dsInv;
};
```

4.1.2 Other Features

As mentioned previously, duplication objects (and Net-Z) offer a high level of abstraction, allowing complex networking features to be provided to the developer with little or no development cost. One such feature is fault tolerance: the ability to detect and deal with a failed node. When a node fails in a multiplayer game that uses the duplicated object paradigm, it is likely that the node contained one or several duplication master objects. However, for each failed duplication master, one of the orphaned duplica can be promoted to assume the role of the master object. Thus, the system can survive the failure transparently, as long as a duplica exists for each failed duplication master.

Given the high cost of bandwidth and detrimental effect of lag, dead reckoning [SC94], a feature in which the position updates of an object are predicted using its current speed and acceleration, is highly desirable. When properly implemented, dead reckoning can provide a significant bandwidth decrease. However, correctly implementing dead reckoning is a very difficult task. Given that the state of object is conveniently stored in datasets, the duplicated objects architecture can provide dead reckoning as update policy on datasets, removing much of the implementation complexity.

Another important feature worth mentioning is the ability to easily migrate a duplication master from one node to another. Hosting duplication masters increases load on a node, as it is responsible for processing actions/RMC and disseminating state updates. A node hosting all the duplication masters in a game could see it's performance significantly decreased. However, duplication masters can easily and transparently be migrated to other nodes. This effectively allows balancing of the

load generated by hosting duplication masters.

4.2 Eterna

Unfortunately, the duplication object technology, as found in Net-Z, cannot be scaled to more than 128 players. Although this is more than sufficient for most multiplayer games, the technology cannot be directly used in massively multiplayer games, where several thousands of players can interact with each other. Eterna, another product from Quazal, attempts to leverage the flexibility of duplicated objects, while scaling it to a massive scale. This scalability is achieved through the use of duplication spaces. However, given the high demand for the Net-Z product, very little development has been done on Eterna in the last few years.

4.2.1 Duplication Space

As mentioned previously, the duplicated objects technology cannot scale to environments with thousands of players. This is partly due to the high cost of bandwidth (as charged by Internet providers) and the limited processing/networking capacity of computers. When dealing with a small number of players, it is feasible to duplicate objects over every single participating node. However, this becomes very inefficient as the number of players increases, especially since most player nodes do not need all this information.

Ideally, nodes should only store duplicas which are relevant to their current view of the game. Duplication Spaces provides this mechanism, by allowing duplicated objects to "discover" other duplicated objects [Qua02]. A match function is used to control how the "discovery" occurs.

Duplicated objects within a duplication space are either subscribers, publishers or both. A subscriber is an object that can discover other publisher objects. When a subscriber discovers the duplica of a publisher, it creates a duplica on the node where the duplication master of the publisher resides. This matching is illustrated in figure 4.1 where different publishers can be matched with different sets of subscribers.

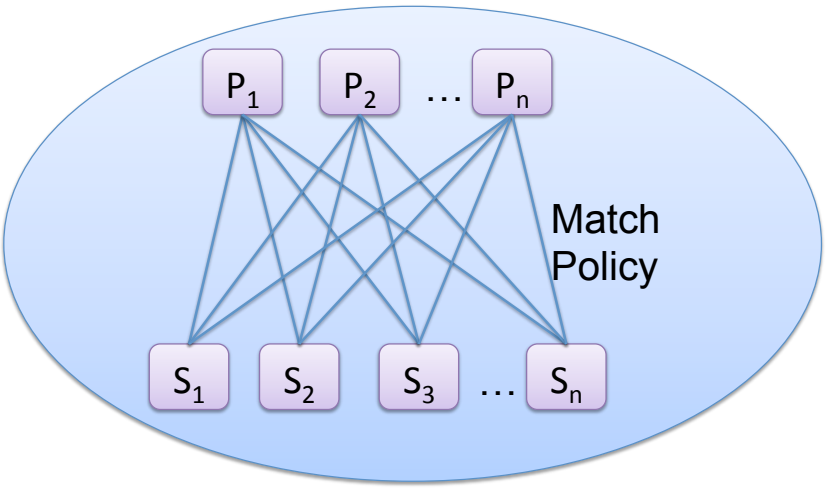


Figure 4.1: Duplication Space, as defined in Eterna.

As previously mentioned, a match function is used to pair subscriber and publisher. A match can be determined based on several criteria, such as the closeness of the two objects, their types, or any other properties involving the state of the two objects. For example, one match policy might match all objects within a 5 meter radius of each other. Another match policy might match all objects found in a particular inventory. An other example is to match all radios operating on the same radio frequency.

4.2.2 Cells

In a simple duplication space, the matching function previously described is executed on a single node that has copies of all the objects in the world. As the number of objects increases, it becomes unfeasible for a single node to handle all this load. Thus, duplication spaces can be further broken down into distinct cells.

When cells are used, publishers and subscribers are broken down into groups, most often based on type or location and assigned to a cell. One node is responsible for determining the matches between all objects that belong to the cell. The matching duties of each cell are then assigned to different nodes. Cells can also be dynamically created to deal with the additional load of new players joining the game. Similarly,

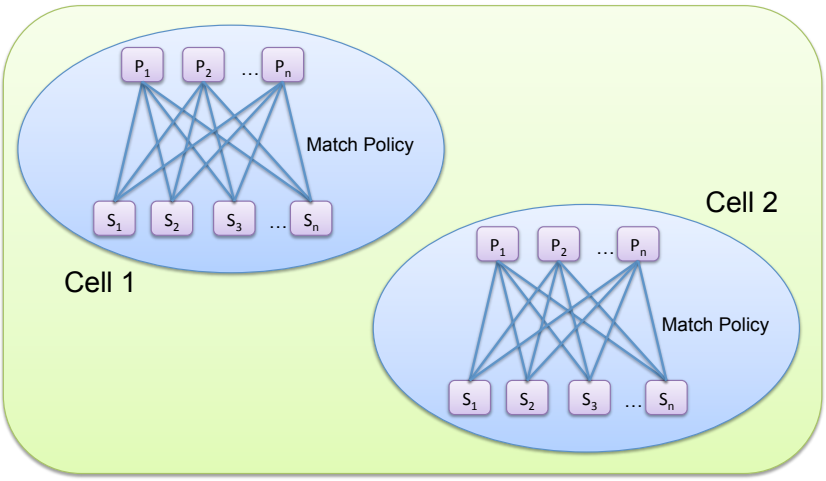


Figure 4.2: Duplication Space with Cells, as defined in Eterna.

in period of low load, cells can be merged together. A CellMatch function is used to map which duplicated object should be managed by which cell.

4.3 Replicated Objects

Replicated objects is the high-level distributed communication layer developed at McGill and used to power the Mammoth Research Framework [KVK+09]. It shares the same core design philosophy as Quazal's Duplicated Spaces, to create a high level abstraction that hides complex issues from developers. However, the goals of both frameworks differ greatly. Given that Quazal is developing a product that must suit a large number of clients, every feature is customized for maximum flexibility. For example, remote method calls can be executed on the duplication master, all the duplicas or individual targeted duplicas. Replicated Objects is more of a research tool, and simplifies the execution model to streamline experiments. For example, remote method calls in the replication engine are always executed on the "master" copy of an object. The logic behind this architecture is explained in section 4.3.1.

Another important fundamental difference between Duplicated Objects and Replicated Objects is how distributed objects are defined. Developers using Duplicated

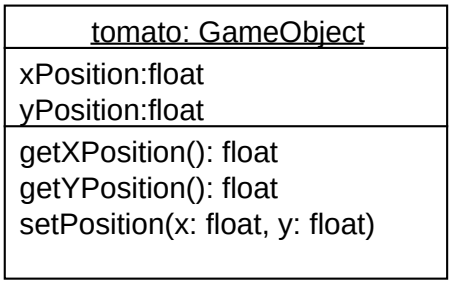


Figure 4.3: A simple tomato GameObject.

Objects define their objects using the DDL language and a compiler generates a series of stub files. These stubs are then completed by the developer. However, Replicated Objects uses a proxy system, where distributed objects are written in Java, and a generator creates the appropriate proxy classes to encapsulate the network functionalities. More information on the proxy system can found in chapter 6.

4.3.1 Masters and Duplicas

When using replicated objects, the state of a game object is replicated across player nodes. Let us take the example of a *tomato*, an instance of a GameObject class (see figure 4.3). The state of the tomato is defined by two attributes describing the position of the tomato. Two methods are used to describe the state of the tomato (`getXPosition()` and `getYPosition()`) while the third method (`setPosition(x,y)`) allows the modification of the state of the tomato.

With replicated objects, a game object can either be a master or a duplica. Only one master can exist at a time for any given object. It contains the authoritative state. Any number of duplicas can exist, each of them having their local state. In the case of figure 4.4, the α instance of tomato exists as a master on node A, and as a duplica on node B and C.

Whenever the game executes a read operation on a game object, it uses the local state. For example, the `getXPosition()` and `getYPosition()` methods are executed

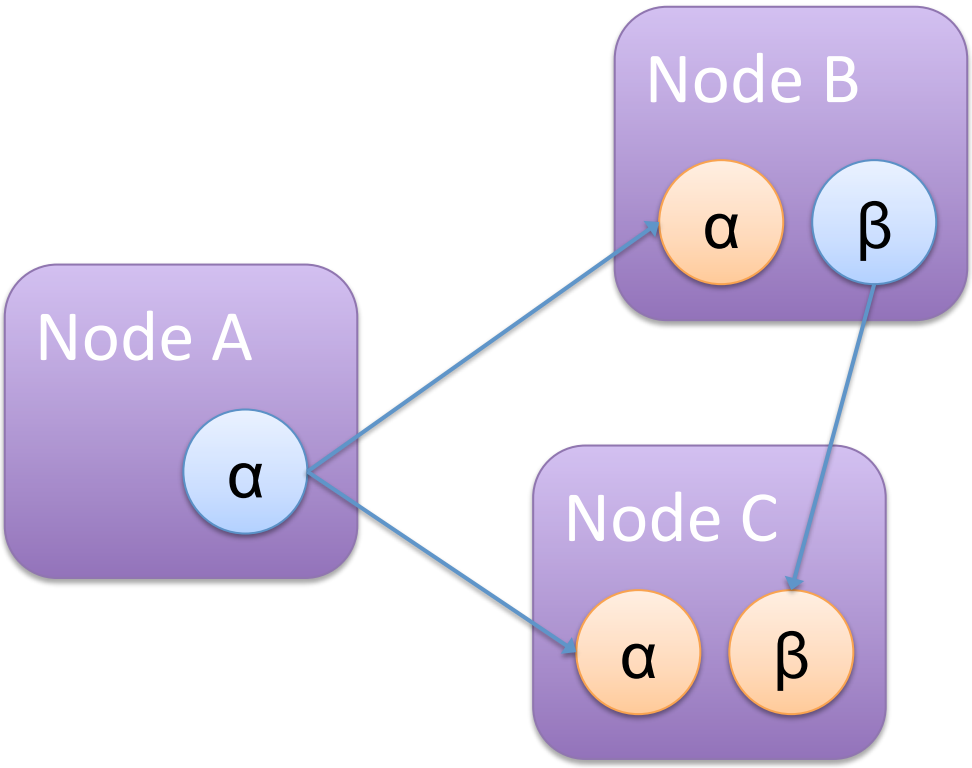


Figure 4.4: Two instances of Tomatoe, α and β, replicated over three nodes, A, B, and C.

locally, even on the duplicas. Modifying operations, however, cannot be executed locally for consistency reasons. If local execution was allowed, it would be possible for concurrent modifications across nodes to take place, which could result in serious inconsistencies visible to the players. For instance, if two players simultaneously decided to move the tomato, only one player should succeed.

In this case, the `setPosition(x,y)` should only be executable on the master object. To this aim, whenever the method is executed on a duplica, the request is serialized and forwarded to the master object. Once the state of the master is updated, the newer state is forwarded to all the duplicas, which, in turn, update their local state.

The remote execution of modifying operations is completely transparent to the game layer. It has no knowledge whether the game object is a master or a replica. The game simply invokes the operation on the game object: replicated objects will redirect the call to the duplication master node, if necessary. This transparency is not only convenient for the programmer, it also makes it easy to migrate the duplicated master from one node to another for load balancing or fault tolerance reasons, without affecting the game layer.

Dealing with Concurrency

It is possible that multiple nodes might try to simultaneously modify the state of an object. In the case of replicated objects, concurrent access is protected by a simple "fail" lock. In other words, if two nodes execute the `setPosition(x,y)` methods simultaneously, only the call to first reach the master object succeeds. The logic behind this is simple: once the first call has completed, the state of the object has changed. Thus, it would be illogical to execute the second call, which was based on the previous state.

This scheme highly favours the node hosting the master object, given that locally routed requests are much faster than those originating from the network. In a game context, this might seem very unfair to players not hosting objects. However, to prevent a player from cheating, it is also highly desirable not to host the master

object on a node that belongs to a player that might gain an unfair advantage by modifying the object in a game-rule infringing way.

It should be noted that fairness and causality are not addressed in this concurrency policy, as they are often ignored in most game environments. A real-time game environment requires actions to be resolved almost instantaneously. The synchronization mechanisms required to ensure perfect fairness and causality are time consuming. However, for low-frequency actions they could be implemented on top of the guarantees provided in Journey, if needed. This is, however, out of the scope of this thesis.

4.3.2 Replication Spaces

The simplest approach to distributing objects is for each player to maintain a full copy of the game state, i.e. create duplicas of all game objects on each player's node. The problem is that this approach does not scale: as the number of players increases, the number of messages to be sent over the network and to be processed by each player's machine increases exponentially.

Since virtual worlds in MMOGs are vast, the most effective strategies to address this problem is to store on a player's node only the game state that is relevant to its avatar. This represents only a small subset of all duplicated objects found in the virtual world. This small subset is determined and maintained through the use of interest management.

4.3.3 Interest Management

As mentioned previously, an effective solution to the scalability problem in replicated objects is to create duplicas only where they are needed. Interest management (IM) is the process of determining which information is relevant to each player [MLS05b]. In replicated objects, IM is a function that determines what a given object is interested in. That function is then used to determine which duplicas are needed for a given interest. For example, a node declares itself related to object A. To function properly, that node requires a duplica of A and any object A is interested in.

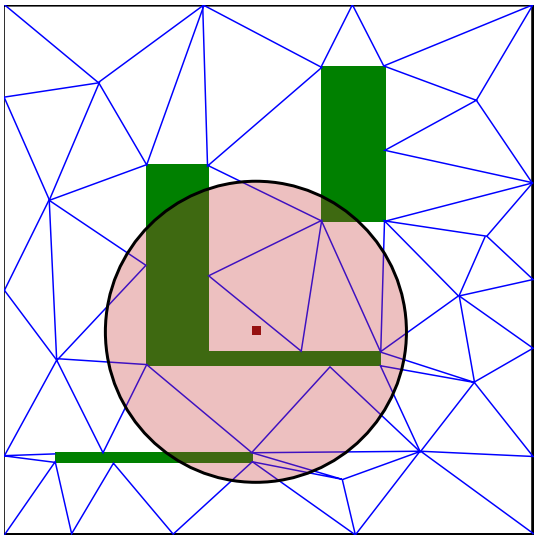


Figure 4.5: Example of circular aura interest.

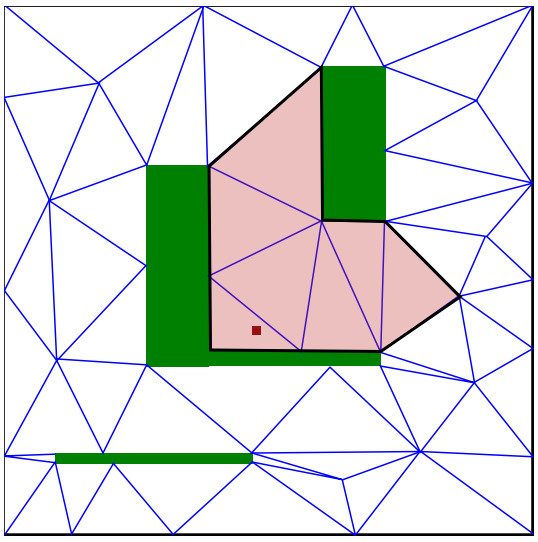


Figure 4.6: Example of interest based on triangulization.

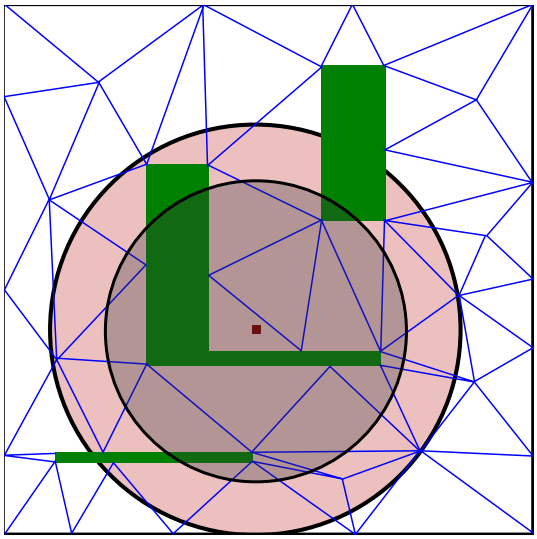


Figure 4.7: Using circular interest, difference between view (grey) and interest (red).

The interest function can be implemented in numerous ways, several of them described and compared in [BKV06]. The most common type of interest function is the circular aura function, as shown in figure 4.5. It is widely used because of its simple implementation and efficiency. However, a much more realistic notion of interest can be built using the partitioning techniques proposed in chapter 5. By using an obstacle aware partitioning, we can easily determine what area of the map is in the object's field of vision (see figure 4.6).

Interest Management vs. Field of View

One common type of interest is field of view. For example, a node controlling a player is interested in all other objects that the player can see. As the player moves around the map, the interest of that player changes. To ensure a proper game experience, new interests must be quickly detected so that the appropriate duplicas can be propagated to the player's node.

To ensure that all necessary duplicas can be found on the player's node before they are seen, a larger interest area is used. This is illustrated in figures 4.7 and 4.8), where the interest area of that node is painted pink, and the actual view of the player

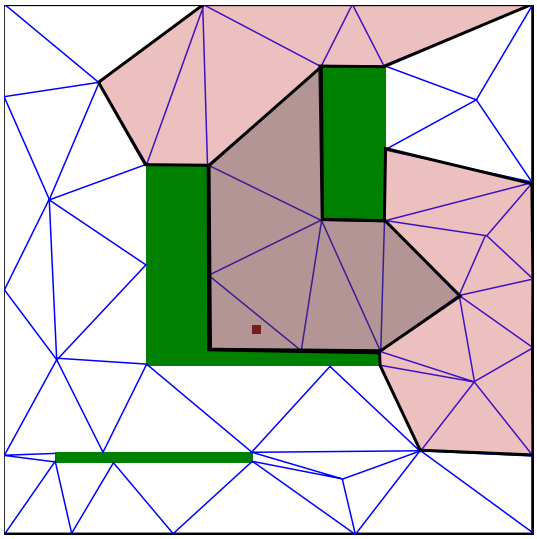


Figure 4.8: Using triangle interest, difference between view (grey) and interest (red).

is painted a darker shade of pink. Since the interest area is larger than the field of view, duplicas will be propagated before they enter the field of view of the player.

How large the interest area must be depends on the speed the player is traveling in the virtual world. This is because the field of view of a fast moving object changes faster. If the replication system updates itself every 2 seconds, the interest area should be increased by at least the distance the player can travel in 2 seconds. In such a case, regardless of the direction the player travels, all the necessary duplicas will already be hosted on his node. However, increasing the size of the interest area even more will only create useless duplicas and increase the total load on the system.

Chapter 5
Obstacle Aware Partitioning

The game map represents the continuous space where players are located and the game unfolds. It is subject to continuous queries, such as "what is near the player?", or "can the player go from point A to point B?". Dealing with these problems in a continuous space is a non-trivial problem, which is often addressed by discretizing the map. This conversion of a continuous map into several small discrete areas is often referred to as *meshing*. Common meshing techniques will split a map into a series of identical squares or hexagons, while more specialized techniques discretize the map into a series of variable-size rectangles.

This chapter presents a technique to mesh a map using triangles that reflectes the barriers and obstacles found in that map. This technique requires the construction of an obstacle map, which is then passed to a triangulation algorithm as a set of constraints. The quality of the meshing is directly related to the simplicity of the obstacle map: a high number of constraints will adversely affect the triangulation algorithm, resulting in triangles with small area or extreme angles. We propose various techniques that can be used to simplify an obstacle map, allowing for an improved triangulation.

5.1 Why do we Partition a Map?

Partitioning of the game map is a core element of Journey. It is used to divide up the responsibilities of the main components (load-balancing, fault-tolerance, cheat detection) and assign the work to difference nodes. However, partitioning game maps has been an essential programming task since the beginning of game development.

One such example is the manipulation of location of game objects on the map. Although determining the location of a target object on the map is usually trivial, determining which objects are near our target object is a non-trivial task. If the map is one continuous space, then we need to compare each object on the map to our target object to determine if it is near. This can be very resource consuming if the map contains a large number of objects. By discretizing the map into several smaller areas, we can reduce the number of checks required to determine the neighbourhood of our target object. Using a quad tree discretization [FB74], we only need to check the quad of our object and the neighbouring quads. When using triangular tiles, we only need to check the tile of the target object and the neighbouring tiles. This results in a very important gain in performance for many game components relying on object location, like collision detection or A.I. behaviour (see figure 5.1).

This discretization can also be of great benefit to pathfinding algorithms. The advantages of pathfinding at different levels of granularity are already well known. By using different partitioning algorithms, paths can be found using divide-and-conquer techniques, greatly increasing response time. And these are just a few examples of how map partitioning is used in game development.

5.1.1 Polygon Triangulation and the Obstacle Map

The most common partition algorithm used in modern video games today is the quad tree. The game map is divided into 4 rectangles (called quads), which are then sub-divided themselves into smaller quads depending on the number of objects found in that quad [FB74]. Thus, all quads have approximately the same number of objects. Locating objects near a specific point is O(log(n)), where n is the number of objects

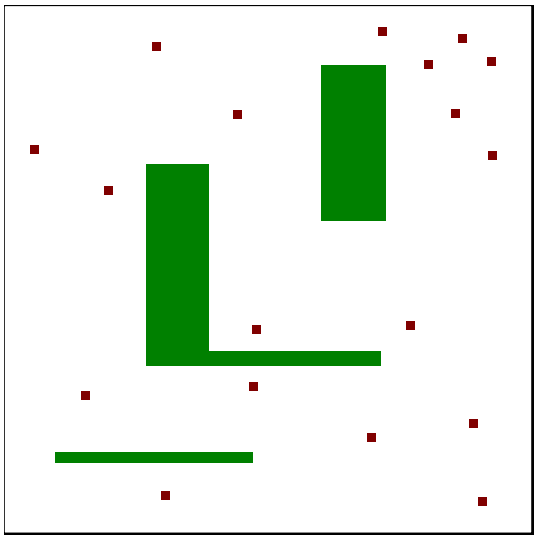


Figure 5.1: Example of a trivial game map. Players are drawn in red, obstacle in green.

on the map. Unfortunately, quad trees are not designed to reflect the geometry of a game map: they do not take into account the obstacles found on the game map (see figure 5.2). Hexagonal partitioning techniques, which are used for instance in telecommunications, have the same flaw.

An obstacle map is a geometric representation of the game map containing only the obstacles found on the map (see figure 5.3). For the purpose of this work, obstacles are polygonal in shape, and block player movement.

The proposed partitioning technique is to build an obstacle map, simplify it, and then generate a triangle mesh using the obstacle map as a set of constraints (see figure 5.4). As mentioned previously, triangle meshes have the distinct advantage of easily reflecting the constraints found on a game map. Problems typically associated with triangle meshes, such as thin triangles, are addressed using various meshing optimizations proposed in this chapter.

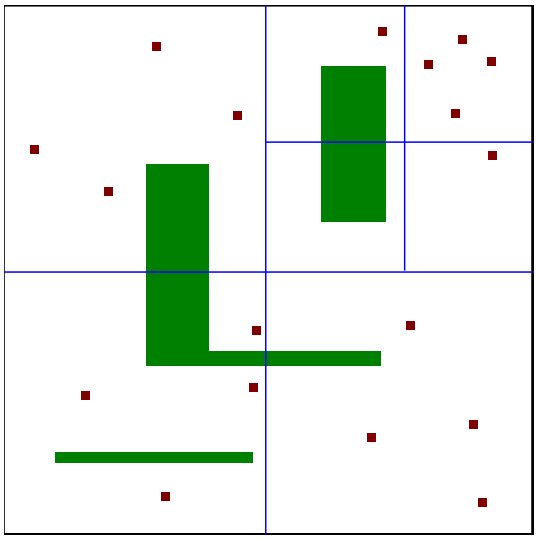


Figure 5.2: Example of a game map divided by a quad tree.

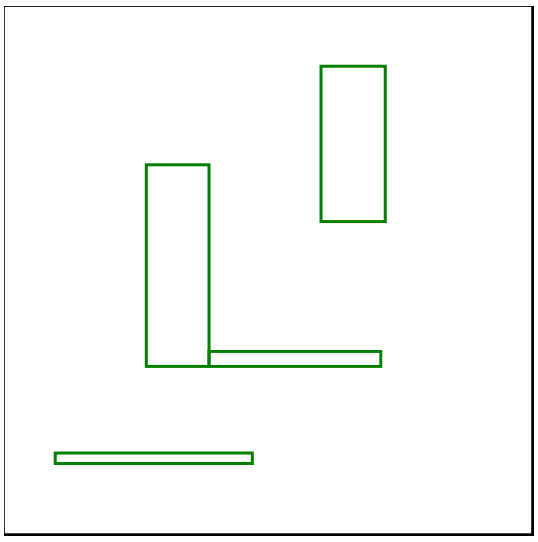


Figure 5.3: Example of an obstacle map.

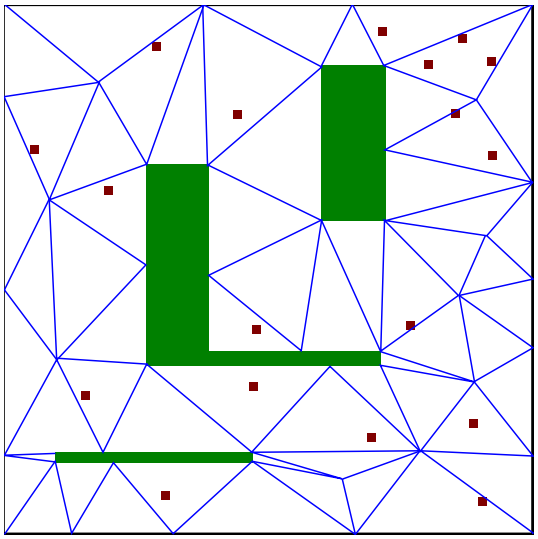


Figure 5.4: Example of a game map partitioned using triangles.

5.1.2 Partitioning in Journey

Journey's relies on partitioning to divide the game map into tiles. These tiles are then used to create cells, which are the main components of the load balancing system. Cells can be grown or shrunk by reassigning tiles from one cell to another. The size and shape of a tile determine how many objects can be found in that tiles. In addition, responsibility in the fault tolerance and cheat detecting system is assigned to different nodes using cells. Thus the load created by these systems is also influenced by the shape of the tiles.

Thin triangles create problematic tiles, as they can contain few objects, and these objects are usually shared among the neighbouring triangles. Large triangles are equality troublesome, as they can contain too many objects, making it impossible for Journey to share the load across different nodes.

5.1.3 Characteristics of a Good Triangulation

The run time of an algorithm that uses a mesh is related to the quality of that mesh. An ideal triangle mesh for our purpose is a collection of equilateral triangles of exactly

the same surface. However, given the obstacle constraint, it is virtually impossible to obtain a perfect map.

A good triangle mesh is mostly composed of non-thin triangles of approximately the same area. For the purpose of this work, we define a thin triangle as a triangle with at least one very small angle (less than 30 degrees). There are two notable disadvantages to the presence of thin triangles:

1. Thin triangles have a very small width, which means moving objects will easily cross in and out of the tile. This increases the tracking costs of moving objects.

2. If the location of an object is determined by its shape, rectangular objects are rarely contained in only one thin triangle. The object will most likely exist in two or three tiles, increasing run time costs of algorithms.

It should be noted that these disadvantages are also found in triangle tiles of any shape with a small surface. As such, small triangles are defined as triangles surface less than half of the target surface.

Although the presence of some thin or small triangles in a mesh is not problematic, their predominance in a specific area of the map can be catastrophic to the run time of algorithms. Thus, it is considered favourable to avoid them.

5.2 Improving the Obstacle Map

Triangulation algorithms have been around for a long time, and the presented work does not seek to improve them. Instead, we present a series of techniques that can be used to simplify the obstacle map, thus decreasing the number of constraints submitted to the triangulation algorithms. These improvements can be seen as filters on the geometry of a map, simplifying that geometry and thus, the obstacle map. However, for a filter to be useful, it must preserve important properties of the map. For example, if two triangles are not connected because an obstacle is present between them, then these two triangles should still be separated after the geometry is simplified using the filter.

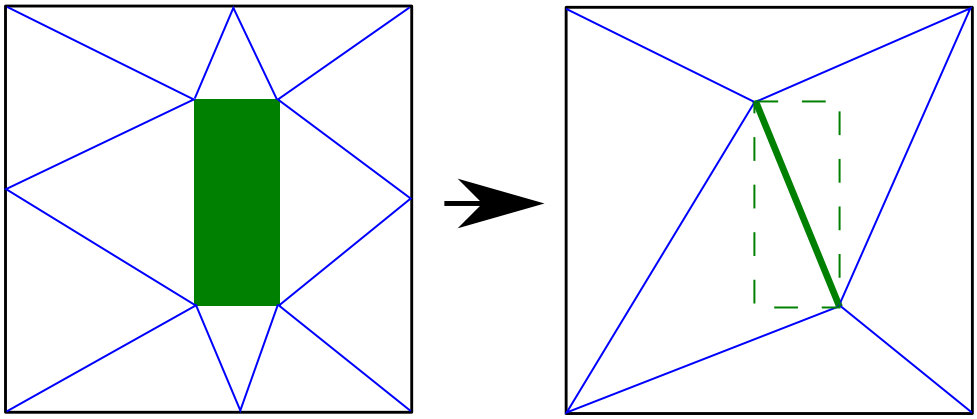


Figure 5.5: Effect on triangulation of transforming small isolated objects into a line.

5.2.1 Dealing with Small Isolated Objects

As mentioned previously, constraints prevent a triangulation algorithm from generating an ideal map. An edge constraint forces the creation of 2 triangles, while rectangle constraints will force the creation of 6 triangles (4 outsides, 2 insides). These "forced" triangles are undesirable, since one of their edges is automatically determined by the constraint, and as a result, the created triangle is too small. Thus, edge constraints are much preferable, as they create less "forced" triangles.

The basic property of the rectangle obstacle is that space on opposite sides of the obstacle should not be connected. By transforming a rectangle into a single edge, diagonally crossing the rectangle, we conserve the connection property, while greatly reducing the number of triangles needed (see figure 5.5). In addition, the less constraints the partitioning algorithm has, the easier it is to generate an ideal triangulation.

Small isolated objects found on a game map are problematic, as their four edges create at least 4 thin triangles, and two small ones (inside). By using the technique described above, we can reduce this side effect to two thin triangles and no small ones. Figure 5.5 demonstrates that even in the simplified topology, space on either sides of the rectangle obstacle remain unconnected.

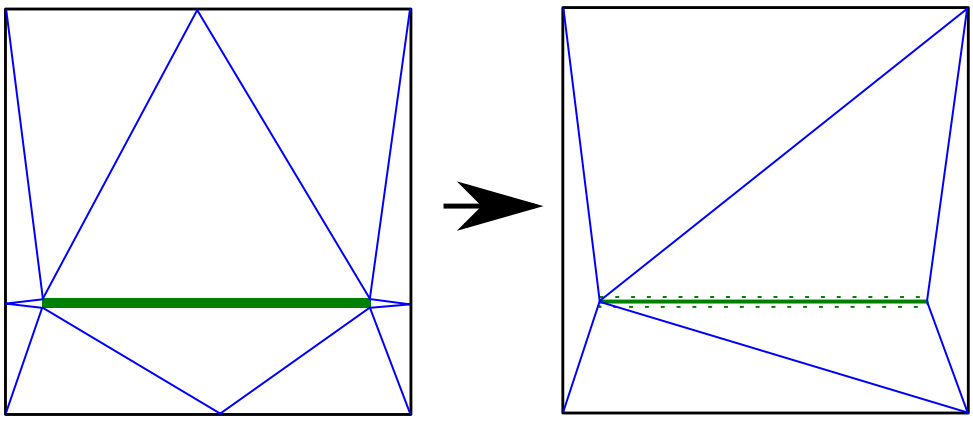


Figure 5.6: Effect on triangulization of transforming rectangular wall objects into a line.

5.2.2 Transforming Walls into Lines

Walls are a special case of rectangle objects, as their goal is to separate two areas of a map. Most walls are composed of two large edges and two thin edges. Long edges cause no problems to current partitioning algorithms. However, the thin edges are problematic, as they force the creation of two very small triangle.

Similar to the technique described previously, a rectangle wall can be simplified to an edge, without violating any of the triangle connection properties. However, instead of using a diagonal line across a rectangle, a straight line is used across the rectangle (see figure 5.6). This simplifies the work of the merging algorithm described in section 5.2.4, which links connecting walls.

5.2.3 Eliminating Very Small Objects

A second subcase of dealing with small obstacles is dealing with very small objects, defined as objects smaller than the players. Experiments have shown that objects this small have very little effect on algorithms using the triangulation. However, their presence greatly increases the complexity of the triangulation, as they introduce many small edges and thus, small triangles.

Two solutions were explored, one reduces them to a point constraint, the other

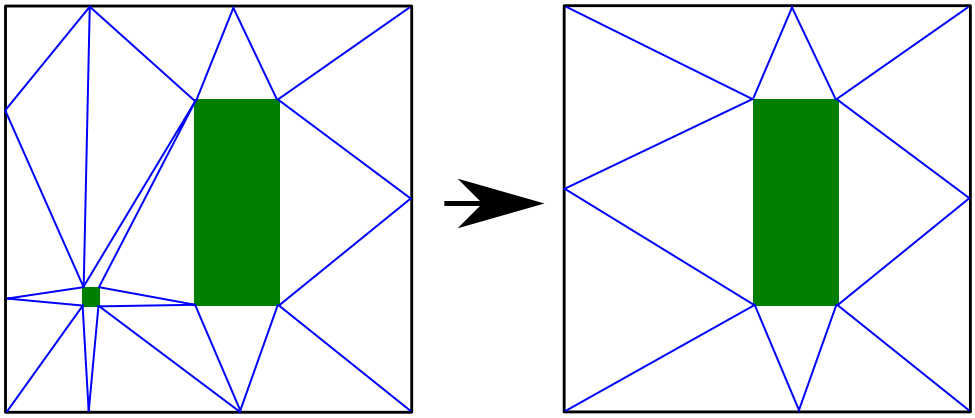


Figure 5.7: Effect of removing small isolated objects on triangulation.

simply removes them from the obstacle map. The latter was the best solution (see figure 5.7), as a small concentration of very small objects would still generate many small triangles. In addition, the removal of small constraints had no negative effect on Journey. However, removing very small objects does have a negative effect on path-finding algorithms using the partitioning. This would suggest the need for different types of partitioning optimizations depending on how the partitioning will be used.

5.2.4 Merging Points Close to Each Other

Two of the optimizations described above transform a rectangle object into a single edge. None of these optimizations take into account that some of these rectangle objects might be adjacent to each other. Once simplified to edges, these obstacles are not longer adjacent. This creates extra spaces between the edges, which in turn results in more small triangles in the mesh. A simple solution to this is to merge points adjacent to each other, closing the number of small openings in the map (see figure 5.8).

The key difficulty in this optimization is choosing a threshold for which points should be merged. If the threshold is too small, no points will be merged. However, if the threshold is too large, the optimization could merge points that were never meant to be merged. This might close small paths that were inserted into the map by design

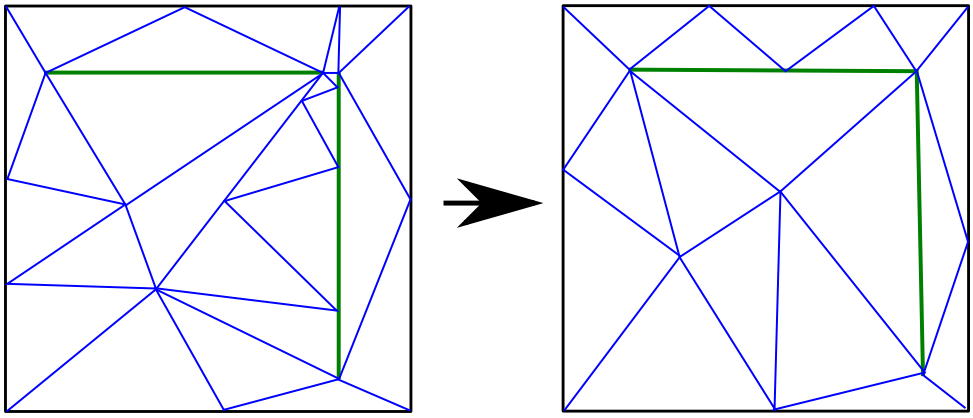


Figure 5.8: Merging nearby points can greatly simplify the obstacle mesh.

so that a player was supposed to be able to use them.

Experimentation has showed that this type of error rarely occurs when the threshold for the merging is smaller than the player. The proposed threshold is thus slightly smaller than a player: small enough to avoid closure of paths meant for players, but large enough to fix some of the inaccuracies introduced by the other approximation techniques.

5.2.5 Merging Small Overlapping Objects

As previously demonstrated, each rectangle in the obstacle mesh creates more constraints, further increasing the number of triangles in the mesh. This is a particularly complex problem when a group of small objects are overlapping. Individually, these objects are negligible and could be ignored. But since they are overlapping, they create a larger obstacle of complex shape.

The first straightforward solution to this situation was to simply merge all the shapes into a simple complex polygon. A convex hull algorithm [Gra72] can then be used to determine the shape of the new polygon. However, this resulting polygon is often very irregular, especially when many small objects are used to create the new polygon.

The convex hull algorithm was simplified by linking the center of the shapes,

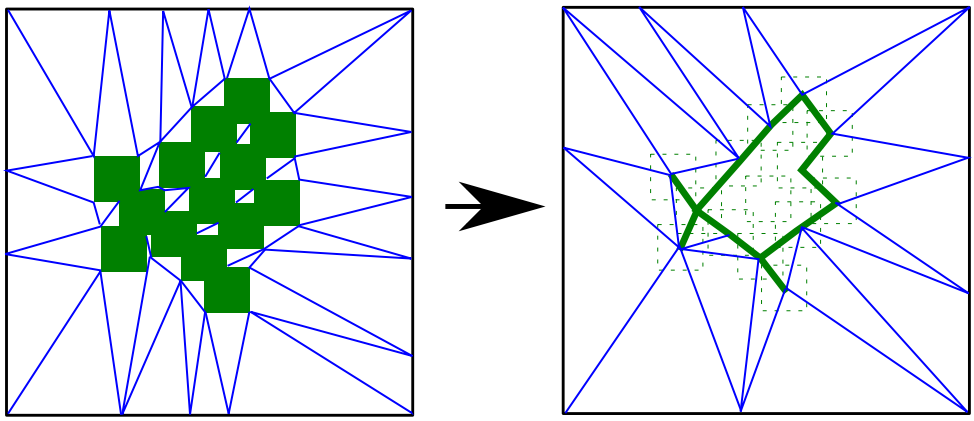


Figure 5.9: Merging overlapping objects greatly simplifies the obstacle mesh.

instead of using the outer edges. The resulting shape can be further simplified by transforming any adjacent edges with an internal angle of over 150 into a single edge. This second simplification does not alter the shape too much, yet reduces the amount of triangles that are created around the shape (see figure 5.9).

5.2.6 Eliminating Flat Edge Triangles

As mentioned previously, constraints limit the ability of a triangulation algorithm to produce a good mesh. One constraint found in all triangle meshes is the four edges delimitating the extremities of the game world. These edges will typically create flat triangles, especially if an object is be found near the edge of the world.

One solution to this is to simply remove these constraints, allowing the algorithm to produce a mesh bigger than the size of the game world. Depending on the algorithm, this can be as simple as specifying the game world to be twice the size, and then removing all excess triangles.

The result, as illustrated in figure 5.10, is a mesh with a smaller number of flat triangles.

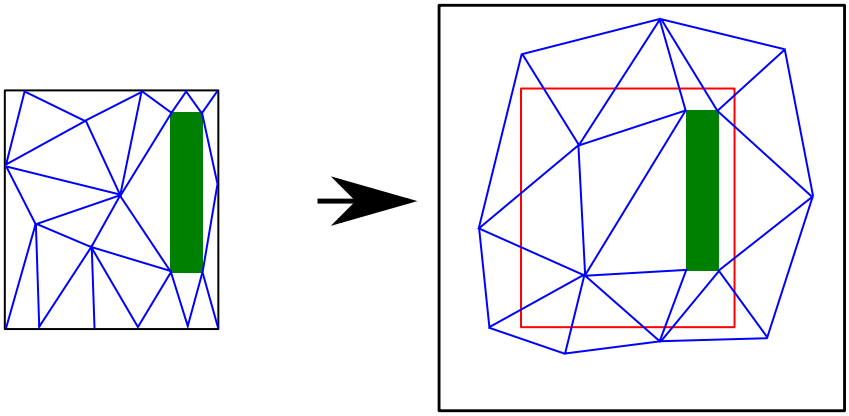


Figure 5.10: Not using the edges of the world as constraints allows the creation of a higher quality mesh.

5.2.7 Order is Important

Early in the experimentation phase, it was discovered that the order of the simplifications was very important. For example, the optimization removing the very small obstacles (see section 5.2.3) must be executed before the obstacle-to-line simplification (see section 5.2.1) is applied. Otherwise, very small objects will be converted to lines before they can be removed. A similar conflict occurs between the merging of small objects (see section 5.2.5) and removing very small objects, as objects that should be merged should not be removed.

On the other hand, the early execution of a particular simplification can also enhance the execution of subsequent algorithms. For example, the merging point simplification (see section 5.2.4) is much more efficient if the walls have already been transformed into lines (see section 5.2.2).

Unfortunately, there are no clear rules specifying the order in which simplification must be applied. However, experiments on ordering revealed that if a simplification targets a subset of elements from another simplification, it should be executed first. For example, a simplification targeting very small objects should be executed before a simplification targeting small objects, as the very small objects might be removed or transformed by the earlier simplification.

5.3 Experiments

Two types of experimentations were done on the simplification techniques. The first set was done using a quality metric, creating triangulations using various sets of simplifications and measuring the quality of the mesh. The second set of experiments compares the network load of 4 interest management techniques, two of which use triangulation.

5.3.1 Quality Metrics

As previously mentioned, a good quality mesh contains a minimum of small or thin triangles, as they can create additional load in the system. However, triangles that are too big should also be avoided, as they can contain an abnormally large number of game objects. This reduces the effectiveness of the various algorithms that could make use of the mesh.

When a triangulation is requested, a target surface area is provided to the partitioning algorithm, giving some indication to how large the triangles should be. If we know the size of the map, the number of triangles in a non-constraint meshed (ideal mesh) can be determined.

$$\text{ideal number of triangles} = \frac{\text{surface of game map}}{\text{target surface area of triangles}}$$

The quality metric can then be defined as the number of bad triangles over the ideal number of triangles.

$$\text{metric} = \frac{\text{number of bad triangles}}{\text{ideal number of triangles}}$$

A lower value of the metric indicates a higher quality mesh. A bad triangle can be defined as:

- small triangle: the triangle's surface is smaller than 50% of the target surface area.

	Town20-2	Town19-4
Surface	400 u^2	900 u^2
Number of objects	572	4244
Number of trees	108	2134
Objects per u^2	1.43	4.71
Trees per u^2	0.27	2.37

Table 5.1: Characteristics of Town20-2 and Town19-4

- large triangle: the triangle's surface is larger than 150% of the target surface area.
- thin triangle: the triangle has at least one angle that is less than 30 °.

5.3.2 Quality Metric Experiments

Metrics were gathered using two maps, Town19-4 (see figure 5.11) and Town20-2 (see figure 5.12), the standard testing maps in Mammoth. The characteristics of both maps can found in table 5.1. It should be noted that Town19-4 is significantly more complex, with buildings that are much more elaborate than what is found in Town20-2. Note that Mammoth does not use a specific measuring unit. Distance is measured in units and surface is simply measured in u^2. A player occupies a surface of 0.03 u^2.

The experiments were done using two different implementations of Delaunay triangulation, a simple one implemented by P. Chew [Pau10] and a more complicated implementation by J. Shewchuk [She96]. The simplifications were tested using two different target surface areas, 1.0 and 0.5 square units, allowing for an approximate maximum of 33 and 17 players per triangle, respectively. The simplifications were tested by themselves, and in combination with other simplifications.

The results for the experimentations can be found in table 5.2. Note that a lower value indicates a higher quality mesh, a value of 0 representing an ideal mesh.

Several interesting conclusions can be drawn from the experiments. The first is

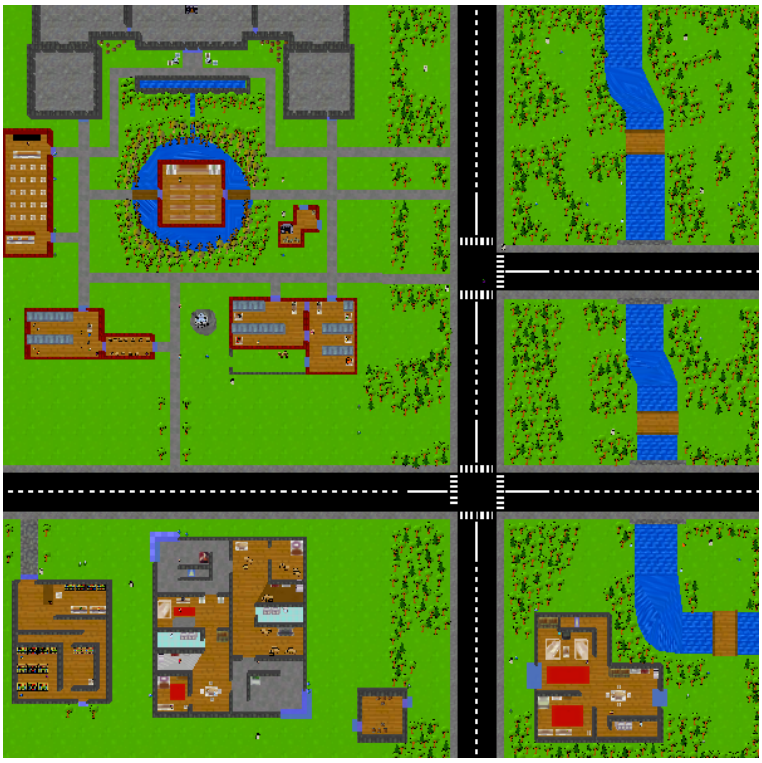


Figure 5.11: Town19-4 map

Figure 5.12: Town20-2 map

Map	Town20-2				Town19-4
Partition Algorithm	Chew		Shewchuk		Shewchuk
Target Surface Area	1.0	0.5	1.0	0.5	1.0
*(a) No optimization	1.707	1.185	1.908	1.080	err
(b) Remove Very Small Polygon	0.890	0.792	0.616	0.583	1.429
(c) Convert Small Polygon	1.217	0.941	0.925	0.457	4.956
*(d) Combine Nearby Points	1.640	1.184	1.844	1.055	err
(e): (c) and (b)	1.151	0.899	0.772	0.584	4.576
(f): (b) and (c)	0.825	0.830	0.504	0.425	1.257
(g): (d), (b) and (c)	0.788	0.777	0.480	0.446	1.288
(i) Combine Overlaping Shapes	0.945	0.868	0.716	0.514	3.915
(j) Spline Overlaping Shapes	1.062	0.862	0.841	0.560	2.104
*(k) Wall to Line	1.532	1.087	1.747	1.039	err
*(m): (k) and (c)	1.512	1.063	1.699	1.004	err
(n): (k), (c), (a) and (b)	0.763	0.699	0.362	0.340	0.702
(p): (i), (k), (c), (a) and (b)	0.766	0.732	0.451	0.359	1.163
(q): (j), (k), (c), (a) and (b)	0.774	0.704	0.461	0.347	0.963

Table 5.2: Results of Triangulation Experiments.

that the ordering of simplification does matter, as can be seen in experiment (e) and (f), where the same two simplifications (b) and (c) are executed in different order. Regardless of the algorithm or the target surface area, it is always better to execute the (b) optimization before the (c).

It should be noted that experiments (a), (d), (k) and (m) are tagged with an asterisks. The experiments, when done using the Shewchuk algorithm on maps Town19-4, produce either particularly bad results, or no results at all. This can be easily explained by the high number of small edges in these experiments, a situation that is problematic with the Shewchuk algorithm. On the other hand, no results are available with the Chew algorithm on the Town19-4 map, as the algorithm is unable to handle the high number of constraints found on that map.

In every scenario, the quality of the mesh is improved using a single simplification. In addition, the combination of multiple simplification is usually very effective. One notable exception is experiment (n) versus experiments (p) and (q): the addition of the simplifications dealing with overlapping objects reduces the quality of the mesh. This is because the simplification that removes small objects removes all the trees from the map. Thus, although experiments (p) and (q) obtain a lower score, their mesh less accurately reflects the geometry of the world. Thus, it is highly recommended that simplifications dealing with groups of objects be applied before any small individual object is simplified or removed. Although these simplifications have little effect on the quality of the mesh, they prevent valuable obstacle data from being removed by other simplifications. This also suggests that the definition of the metrics used to compare the approaches could be updated in the future to take into account how accurate the obstacle-aware triangulation is.

One last conclusion is the disappointing performance of the wall to line optimization. Although these optimization produces meshes that are more aesthetically pleasing, they have very little effect on the results. This is probably due to the fact that the two maps we used for testing do not contain many walls. In the future, it would be interesting to look at game maps of interiors of buildings to investigate if in that case the wall optimization is more useful.

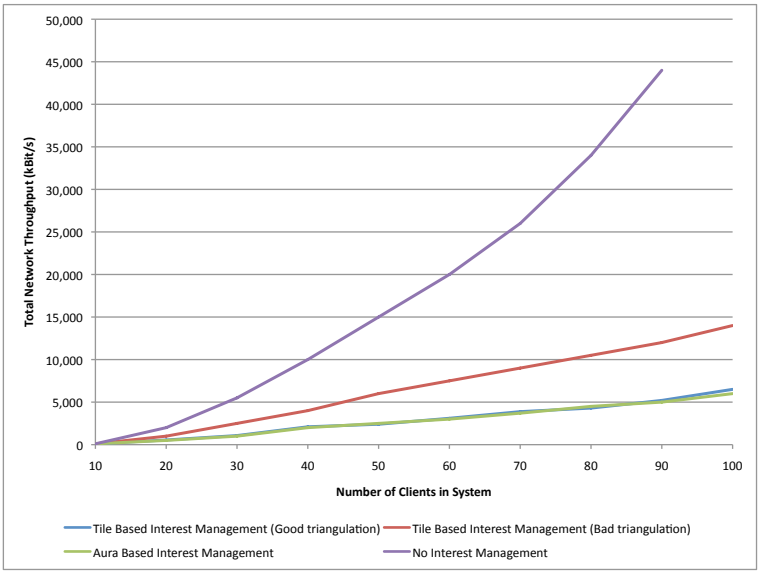


Figure 5.13: Network Traffic (in/out) at Hub during IM tests.

5.3.3 Interest Management

To better illustrate the usefulness of good triangulation in interest management, this experiment measures the load generated by different interest management strategies. The metric measured in this case is network traffic at the hub (with the current network engine, all traffic is routed through a central hub), since one of interest management's primary goal is to minimize the amount of data to be transmitted. As such, an effective interest management solution should decrease the network traffic. Four scenarios were evaluated: no interest management, aura-based (circular) interest management and triangle-based, both using a good and bad triangulation. These triangulations have a tile quality value of 0.5 and 1.7 respectively, as described by the metric in section 5.3.1.

Figure 5.13 illustrates the increasing amount of network traffic as more clients connect to the system, regardless of the interest management technique used. However, the no-interest management experiment shows a polynomial curve, which is clearly undesirable. Both the aura-based and the triangle-based interest management technique show considerable improvement. However, in the case of the triangle-based

interest, the improvement depends on the quality of the triangulation.

The good triangulation offers performance similar to aura based interest management, which is highly desirable. This indicates that using a good triangulation allows for more realistic IM (when compared to aura, which does not take into consideration walls and other obstacles), without increasing network bandwidth. However, using a lower quality mesh creates a huge bandwidth penalty.

Chapter 6
Extending Remote Procedure Calls (RPC) with Proxies

Executing a subroutine or procedure on another computer is not a new concept. Origins can be traced to the mid-seventies, in RFC 707 [Whi75], where the idea for a "procedure call protocol (PCP)" is presented to allow for remote procedures. To this date, many different implementations are available.

However, remote call systems have a reputation for being slow and inefficient. This can be easily explained by the extensive marshalling[1] required by these systems. Remote call systems are designed to be generic, to function on any method with any number or type of parameters. This imposes a great burden on the marshalling system, as it must be able to encode and decode these method parameters. This marshalling becomes even more costly for cross-platform/cross-language remote call systems. In addition, most remote call systems are overly synchronized. To ensure that data shared by all nodes is update in a coherent and ordered fashion, all method calls much be routed to a centralized location and extensive locking must be used. Because of this, most RPC systems do not allow asynchronous method calls, preferring a simplified call flow where one remote call must be completed before a second is executed.

[1]Marshalling is the process of transforming the memory representation of information to a data format suitable for storage or transmission.

These inefficiencies make remote call systems ill-suited for performance-driven applications, such as games. However, in a game specific context, the definition of method calls can be greatly simplified, reducing the burden on the marshalling system. In addition, games often do not require perfect consistency, thus reducing the number of locks and safeguards needed in the system. Ideally, remote calls can even be asynchronous, as games don't always expect a return value on a remote call. Thus, game-specific optimizations could provide great performance improvements to remote call systems.

In addition, most object-based remote call systems use some kind of proxy/stub system to create the remote object on the participating nodes. These proxies usually come in two flavours, the master stub and the replica stub. Externally, these stubs are identical, only their internal implementation differ. This represents one of the great advantage of such a proxy system: it provides location transparency. In other words, users of the proxy need not know if they are using the master or replica version of the stub.

In this thesis, in order to provide an efficient and easy to use, unified framework for fault tolerance, load balancing and cheat detection, we further extend this architecture by creating different types of replicas. Various features, such a fault tolerance, logging and persistent storage can be added to the application with limited effort, by creating new types of replica stubs. For example, in addition to providing local access, a fault-tolerant replica object could take over the role of a master stub if a fault occurred on the node currently hosting the master.

6.1 Existing RPC Infrastructure

Studying existing remote procedure call infrastructures was a critical step in developing this RPC system. The basic concepts in most RPC systems are quite similar: they allow remote execution of methods, and provide consistency using various forms of locking. As such, the following section focuses on the data structures and messaging systems used by the different RPC infrastructures currently used on the market.

6.1.1 Open Network Computing (ONC) RPC

Originally developed by Sun Microsystems as part of their Network File System project (NFS), Open Network Computing (ONC) RPC has been implemented and deployed on various Unix-like and Windows operating systems. It is considered as one of the first popular RPC systems to be widely deployed. Procedure calls are serialized into eXternal Data Representation (XDR) [Sri95], an IETF standard for encoding data in an architecture independent manner.

ONC RPC itself provides no reliability guaranties, unless it is used with a reliable transport layer, such as TCP. The RPC call message has three unsigned integer fields:

- program number, managed by a central authority (Sun), each number uniquely identifies an application.
- remote program version number, allows different versions of an RPC language to run concurrently.
- remote procedure, uniquely identifies the procedure to be called.

Authentication data is sent with each RPC, and is validated at each call. When multiple messages must be sent simultaneously, they are often clumped together in a message. One interesting feature is that an RPC can be broadcast or multicast to several recipients. Extensive error handling for failed calls is also provided. It should be noted that ONC does not provide any object replication features.

6.1.2 Common Object Request Broker Architecture (CORBA)

Defined by the Object Management Group (OMG) [OMG09], CORBA is most widely known because of the large number of language implementations, including Ada, C, C++, Erlang, Lisp, Ruby, Smalltalk, Java, COBOL, Perl, PL/I, Python, TLC, and Visual Basic. However, the usability of CORBA is severely hampered by it's lack of interoperability among vendors [Cha98, Hen06].

It should be noted that CORBA implements object transparency [V+97]: a remote party can execute a procedure on a target object with no knowledge of the location or

the implementation of the object itself. This is achieved through the ORB component, which ensures communication between the various components of the framework. Objects are identified through an object reference which is created when a CORBA object is created. A directory service is provided to allow a client to lookup objects.

CORBA objects are defined using the OMG Interface Definition Language (OMG IDL), similar in syntax to interfaces in Java. This allows objects to be language independent. IDL defines a set of basic types, and allows references to other types of objects. A set of language compilers can then be used to generate stubs for the client platform and skeletons for the server platform. Generic stubs and skeletons also exist, to allow remote execution of procedures on objects not defined in IDL.

Commonly, three communication protocols are supported. These protocols mostly differ with respect to the blocking/non-blocking behaviour of a client when a call is issued.

- Synchronous: The client blocks once the call is invoked, waiting for a response.
- Deferred Synchronous: The client does not block, but must check periodically for a response.
- Oneway: The client does not block, since no response is expected.

In addition, asynchronous message-based communication is also possible [SV98]. Method calls are transmitted from one party to another using the General Inter-ORB Protocol (GIOP). A key issue in making sure that two different CORBA implementations are compatible is making sure their ORB protocols are compatible.

6.1.3 DCOM: Distributed Component Object Model

The Component Object Model (COM) is a software architecture that allows the components made by different software vendors to be combined into a variety of applications [WK94]. This architecture can be considered as a standardized inter-process object communication system. COM was designed by Microsoft, and is heavily used in the design of their operating system.

The Distributed Component Object Model (DCOM) extends the Component Object Model (COM) to support communication among objects on different computers on a local area network (LAN), a wide area network (WAN), or even the Internet [HK97]. As an extension to COM, DCOM is transparent, adding network communication to the existing COM inter-process system.

DCOM also provides location transparency, as clients do not need to know if a component is running locally or remotely. A location service allows an object to be located, using a Class ID (CLSID), which in turn, is simply a 128-bit integer. The marshalling and unmarshalling of method calls is part of the distributed computing environment (DCE) standard. DCOM Objects are defined using an Interface Definition Language (IDL), similar to what is found in CORBA. A Microsoft IDL (MIDL) compiler then creates the proxy and stub code needed by the DCE to transmit the call across the network. The marshalling and unmarshalling process can also be customized, giving the programmer full control on how the object is transmitted.

DCOM also provides some interesting security features, mainly in regard to data protection: integrity and privacy. If the integrity functionalities are activated, each method call carries additional data that will guarantee data integrity. As for privacy, it can be controlled at many levels, allowing programmers to define who can access particular DCOM objects and which methods can be accessed.

It should be noted that COM and DCOM have fallen out of use, now deprecated in favour of the .NET framework.

6.1.4 Java RMI: Remote Method Invocation

Java Remote Method Invocation (Java RMI) enables a programmer to invoke methods on objects located on other Java virtual machines, possibly on different hosts [Mic10]. A key goal of RMI is to support seamless remote invocation on objects in a simple (easy to use) and natural (fits well in the language) way [Mic06].

As with other RPC systems, Java RMI uses stub and skeleton classes to hide the networking and invocation code of the RMI system. However, these stubs and skeletons are generated dynamically by the virtual machine when the object is registered

with the Java RMI registry. In addition, remote objects are defined natively as Java interfaces.

This has several implications:

- No IDL language is needed.
- All types in Java are supported.
- Interfaces can use inheritance.

However, RMI uses object serialization to marshal and unmarshal parameters. Although this provides great flexibility, it hinders performance given that object serialization in Java is very slow. Natively, Java RMI only provides one execution model: all method calls are blocking and overly synchronized.

Although RMI was a strong candidate as the RPC system for Journey, its slow marshalling/unmarshalling and it overly blocking model proved problematic performance-wise. A custom solution inspired by RMI was decided to be preferable.

6.1.5 Quazal NetZ

Although NetZ is primarily an object replication middleware, its communication layer is powered by a very interesting RPC system. The concept of duplication space and DDL, which would be the Quazal equivalent to DCOM/CORBA IDL is heavily discussed in details in section 4.1.

A key difference between NetZ and traditional RPC systems is that data is replicated on client nodes. Most RPC systems only distribute a stub to client nodes. However, NetZ distributes a duplicated object (interface and state), allowing for both local and remote calls to be executed on the duplicated object. This can result in a major speed increase in some method calls, as they can be executed locally. This is especially true for methods that do not modify the state of the object. However, duplicating the state introduces some serious data replication challenges, since changes to the master copy of the object must always be distributed to duplicas.

While most RPC system favour transparency, NetZ sacrifices transparency to allow for greater customizability. For example, all remote method calls are associated with a context object. This context object allows a developer to customize each method call, including whether the call is blocking and how the state update is propagated. In other words, the programmer is encouraged to interact and customize the RPC system.

6.1.6 Other RPC systems

This section is, by all means, not a complete list of all RPC systems currently available on the market. Web RPC services, such as SOAP, or database RPC systems, such as ODBC are not mentioned, as they are considered outside the scope of this research.

6.2 Architecture of the Journey RPC System

When designing the RPC system for Journey, three types of RPC architectures were considered. The first uses a configuration file to define the RPC method call and to generate the appropriate stubs. A developer simply needs to fill in the method stubs with the game logic. However, this architecture was discarded because of the complexity of creating a stub generator that would merge the implementation code if the stubs were to be regenerated. In addition, the use of stubs did not provide the desired level of transparency. The second architecture uses aspect-oriented [EFB01] programming to weave the RPC code into existing classes. Although this solution has the highest level of transparency, it required using an aspect-oriented compiler such as AspectJ [Asp10]. Unfortunately, AO compilers are nowadays still far less mature as standard Java compilers. In addition, it is not clear how the use of AOP would affect performance. Thus, the third option was chosen, using the proxy design pattern to add RPC functionality to existing classes. This provides a good balance of transparency and efficiency. This architecture is presented in more detail in the following subsections.

Note that the RPC system of Journey is integrated with the replication space

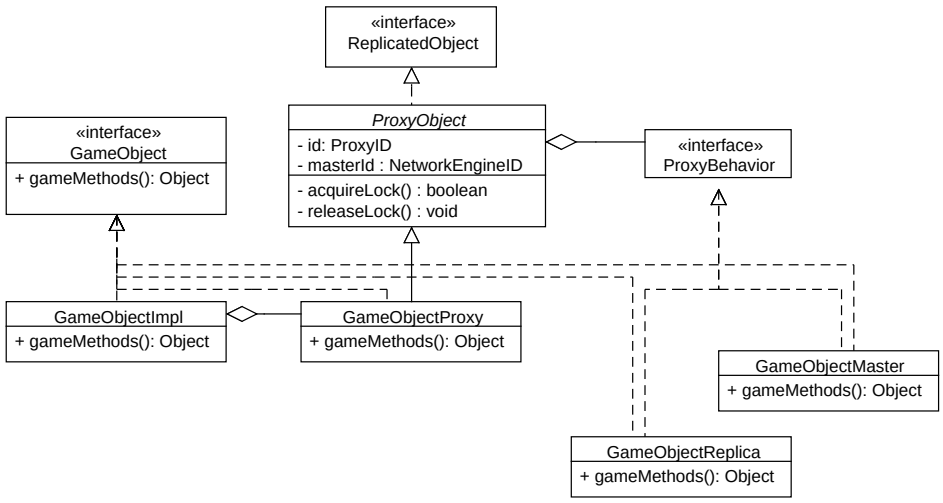


Figure 6.1: The Architecture of the RPC system

technology. As such, after an RPC call, any changes to the state of a master object are propagated to all the replicas of that object.

6.2.1 Using the Journey RPC System

When using the RPC system, three design restrictions are imposed on the developer:

- Classes with remote calls must be split into two components, an interface defining all public calls and a class containing the actual implementation. This allows the proxy to have the same interface as the implementing class, and thus be used transparently by the developer. This architecture is illustrated in figure 6.1.

- All remote calls must have a void return type and their first argument must be a `RemoteCallContext` object. This design restriction is to allow remote calls to be asynchronous. The `RemoteCallContext` object is used to track the remote call and store any return value or exception that might have been thrown.

- To allow the RPC subsystem to distribute updates on shared objects, variables

updated by remote calls must either be public or protected. Although this restriction could be avoided using reflection or aspect-orientation, doing so would be very inefficient.

To use the RPC system, developers must use annotations to tag the remote calls in the interface. For example, the `setDestination` method call in listing 6.1 is a remote call that modifies variables in the "destination" data set. However, the `getDestination` method is not remote, and is executed locally on the object.

Listing 6.1: Example of an annotated method call

```
@RemoteCall(dataSet="destination")
public void setDestination(CallContext callContext, double x,
    double y);

public Position getDestination();
```

Variables to be updated by the RPC system must also be tagged in the implementation class (see listing 6.2). This ensures that annotated variables are properly updated after remote calls. In addition, variables are tagged with the data set they are members of. For example, if the `setDestination` method call from listing 6.1 is executed, then the `destination` variable is updated on all remote duplicas. However, the `speed` variable is not updated, as it is not in the same data set as the `setDestination` method call. The `path` variable is never updated after a remote call, as it is not tagged as a remote variable.

Listing 6.2: Example of an annotated variables

```
@ReplicatedAttribute(dataSet="destination")
protected Position destination;

@ReplicatedAttribute(dataSet="speed")
protected double speed;

protected Path path;
```

Once the interface and the implementation class are annotated, the developer must declare them in the `replicationengine.properties` file (see listing 6.3).

Listing 6.3: Sample of replicationengine.properties

```
player.interface = Mammoth.WorldManager.Player
player.implementation = Mammoth.WorldManager.World.PlayerImpl
```

The final step is to run the proxy generator, which generates all the required files. The developer can then use the ReplicaFactory class to register the objects to be replicated and the corresponding proxies will be automatically created.

6.3 Implementation

As mentioned previously, the presented RPC system is tied into the replication space technology. As such, remote calls are executed only on a master object and state changes are propagated to all the replicas. A proxy generator class uses the annotation presented in the previous section to generate the proxies.

6.3.1 Asynchronous

One key limitation in many RPC systems is that executing a call is always synchronous. This is often a side-effect of the RPC system trying to ensure perfect consistency through locking. However, in game programming, speed of execution is often more desirable than perfect consistency. Thus, asynchronous method calling is highly desirable.

In an asynchronous setting, return values from method execution is not immediately available to the client. Instead, it is made available once the remote method execution is completed. A mechanism is needed to store the answer so that the developer can latter poll for it. In the presented RPC system, this is the job of the `RemoteCallContext` object. When a developer executes a remote call, he must create a new `RemoteCallContext` object. This object has a unique execution id, which

identifies the remote call throughout its lifetime. Once a remote call is completed, the `RemoteCallContext` object is updated with the return value.

6.3.2 Proxy Generator

Written using Apache Velocity [Apa09] templates, the generator analyses the annotations described in the previous section and creates four files for each object with remote calls.

- The proxy class
- A class describing the behaviour of a master object
- A class describing the behaviour of a duplica object
- A message class used to update duplicas

The first iteration of the proxy generator only generated two separate proxy classes, one for masters and one for duplicas. However, when work on object migration started, a significant design flaw was discovered. Migrating an object from one node to another is the equivalent of asking a master object and a duplica object to exchange roles. However, since masters and duplicas were defined as different objects, exchanging the roles required creating new proxy objects. This complicated matters, since the old proxies had to be de-registered from the game engine and replaced by the new proxies. To avoid this problem, the current proxy generator encapsulates the master/duplica behaviour in separate classes, making it easy to switch behaviour without having to recreate the proxy class.

Although the notion of datasets were introduced early in the design, they were not implemented immediately. Given how custom serialization works in Java, it was very difficult to pass the dataset name to the de-serialization algorithm as it traversed the object's hierarchy. In addition, injecting custom serialization code into classes would have violated the transparency requirement. The message class was introduced as a work-around to this problem, by making the de-serialization algorithm

external to the object itself. This introduced the small restriction that replicated variables must be declared as protected or public. As previously mentioned, reflection or Aspect-Oriented Programming could have been used to removed this restriction. However, given the high-frequency of the operation, reflection would have been too inefficient. Similarly, solutions using Aspect-Orientation were previously discarded for performance reasons.

Part II

Journey

This part introduces the main components of Journey. Firstly, the notion of trust and its relation to the other components of Journey is discussed in chapter 7. Load Balancing, which allows Journey to scale to impressive workload, is introduced in chapter 8. Finally, chapter 9 is dedicated to both fault tolerance and cheat detection, two components which provide an enhanced user experience through reliability and fairness.

Chapter 7
A Unified Approach to Load Balancing, Fault Tolerance and Cheat Detection using Trust and Game Replicas

Researching duplication space has demonstrated the existence of an important commonality between load balancing, fault tolerance and cheat detection (see figure 7.1). For example, the mechanisms required for load balancing, such as node monitoring and object migration are key components required to implement fault tolerance. Likewise, monitoring a node for software faults or cheating requires similar functionalities. The notion of trust is key to all three of these concerns. In case of an overload, fault or cheat, responsibility should only be bestowed to a trusted node.

Firstly, the unified approach, further describing the commonality between components is discussed. Secondly, the notion of trust is explored, as it determines the level of responsibility each nodes in the system has.

7.1 Unified Approach

Duplication spaces, by definition, have redundant data. Each time a new participant joins the system, it receives a copy of objects interesting to it. In addition, each time the ownership of an object is exchanged, a duplicate copy is created. This redundant

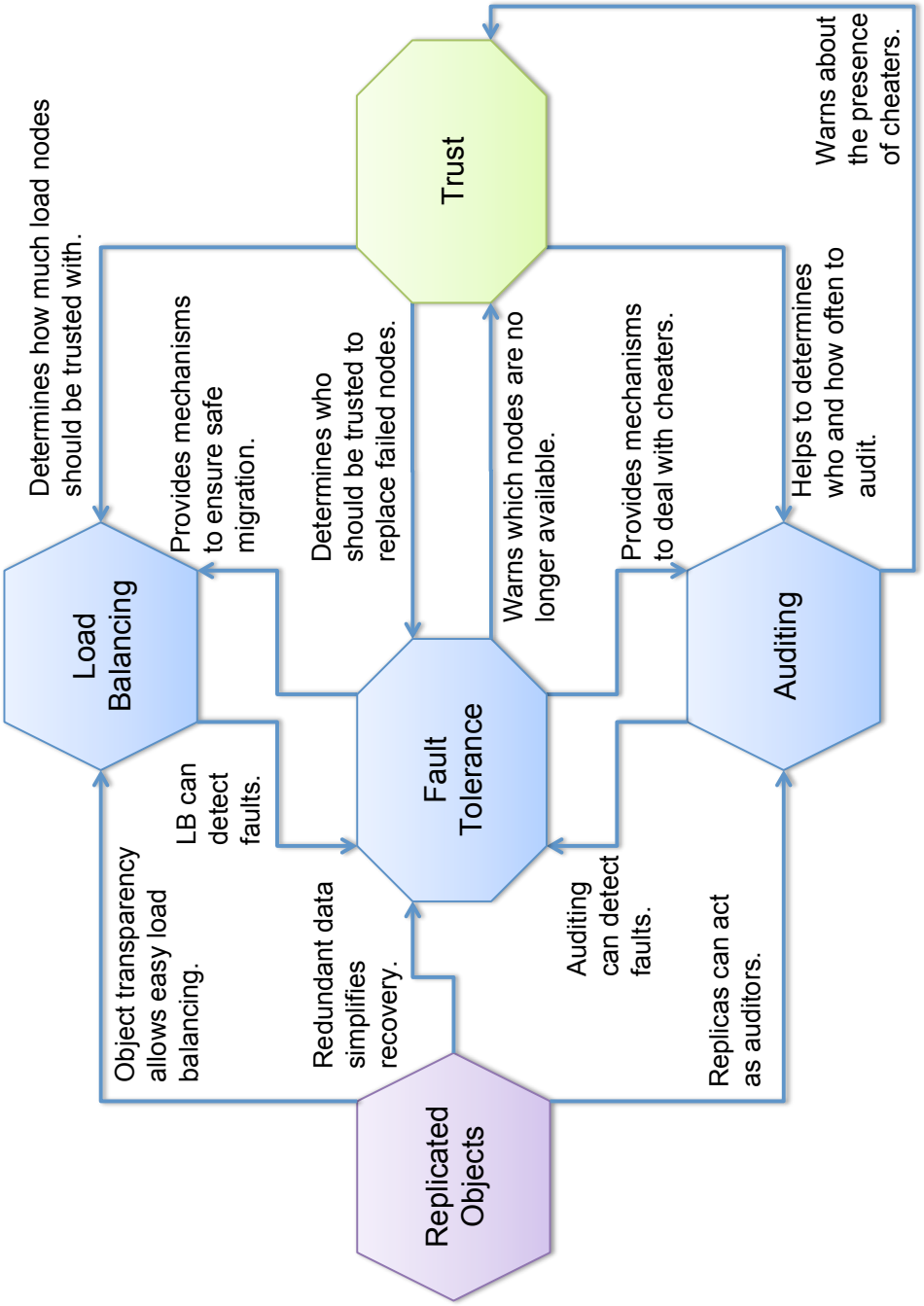


Figure 7.1: Interactions of Components in Unified Architecture.

data can easily be exploited to provide fault protection and cheat detection. In addition, developers use objects without any knowledge of the location of its master copy. Not only does this simplify load balancing by allowing objects to be moved around freely, it accelerates the recovery process in case of faults, as the location of the master copy can easily be updated.

Replicas can also be used as auditors. When a request is sent to a master object, it is forwarded to an auditing replica. Since state updates are propagated to all replicas, the auditing replica can easily detect how an incorrect update was propagated. A single occurrence of error might indicate the presence of a fault, and the corrected state can be easily propagated. However, repeated errors might indicate the presence of a cheater, who should not be trusted with the master copy of objects.

Unfortunately, auditing each action for faults or cheating is too resource intensive. However, faults and consistent cheaters can still be detected by auditing a sample of remote calls. The amount of calls that can be safely audited can be determined by the current load of a node, as measured by the load balancing system, or its reliability in the system.

In addition, a common fault in P2P networks is the voluntary or involuntary loss of a participant, most likely due to disconnection. Load balancing infrastructures consistently monitor the state of the participants, detecting overloads and balancing the load to other participants. This makes the load balancing infrastructure ideal for detecting nodes that have failed or disconnected. In addition, the ability to migrate objects from one node to another, as done in load balancing, is a key aspect of recovering from a fault.

To make decisions about which nodes can be used for load-balancing, fault tolerance and cheat detection purpose, we propose to add the notion of *trustworthiness* of nodes. A node can acquire trust by behaving properly in the system. The more trustworthy a node it, the more responsibility it is given.

7.2 Trust

Theoretically, all nodes found in a Journey system are equal. This means that they have the equal potential to host any type of components, whether a cell or an object. However, this concept of equality is not practical, given that not all nodes are equal in terms of resources or reliability.

Reliability can be considered at many level. For example, a node with low computing power would not be considered reliable under heavy load. The network connection of the node also plays an important part in its reliability. In addition, it is impossible to determine if a node joining the system has malicious intent.

In Journey, trust is defined as a metric to measure the amount of confidence the system should have in a given node. The more confidence a node has, the more responsibility it can receive. Trust can not only control which types of components a node can host, but the amount of components a host can handle.

A node gains trust the longer it stays in the system and by handling objects without faults. Trustworthy nodes can then receive more master components, especially when an overload, fault or cheat is detected.

Nodes hosted by the service provider (the company running the game) are often considered perfectly trustworthy.

7.2.1 Levels of Trust

The current design of Journey defines three levels of trust.

- **Level 0 : Untrusted** : At this level, the node is not trusted and cannot host any master object or cell. It can, however, register its interest to receive replicas. A node not hosted by the service provider will typically start at this level, unless trust is recorded in a persistent fashion.

- **Level 1 : Trusted for Master Object** : A node granted this level of trust can host a limited number of master objects. This level of trust is usually given to an untrusted participant after of certain period of faultless operation. However,

the amount of responsibility is limited, as it is assumed that this node can leave the system at any time.

- **Level 2 : Trusted for Master Cell** : The highest level of trust granted in the system, a node with this trust level can host any number of objects or cells, and can be used for any form of auditing. Nodes at this level are considered to have extreme capacity, high reliability and security beyond reproach.

In a typical Journey setup, trust level 0 is assigned to clients at startup, and server-like nodes hosted by the service provider are assigned a trust level of 2. Clients can earn level 1 by operating flawlessly for a certain period of time.

7.2.2 Acquiring Trust through Time

Dealing with the loss of a level 1 or 2 node is more resource consuming than dealing with the loss of a level 0 node, because lost master objects or cells must be recovered. Thus, trust should be acquired only after reaching a certain period of time of flawless execution. This point is well explained in [PG07], where 40% of World of Warcraft play sessions they observed lasted less than 10 minutes. Those short game session represented players who logged in only to check for messages and transactions (trades, auctions, etc). It was also shown that once a player breaks the 15 minutes barrier, he is much more likely to stay in the game. However, statistics from [FBS07] show that 80% of sessions last 60 minutes, but less than 10% of play sessions will exceed 120 minutes. If these statistics were used to determine trust over time (see figure 7.2), trust should increase only after 15 minutes (point X), and should start decreasing after 60 minutes (point Y).

7.2.3 Acquiring Trust through Capacity

In this case, the word trust is a bit misleading. Trustworthiness does not only represent a node's *ability* to provide a service reliably, but its *capacity* to do so. As such, a node that can successfully host several hundred objects without any capacity problem is considered more reliable than a node capable of hosting less than a hundred

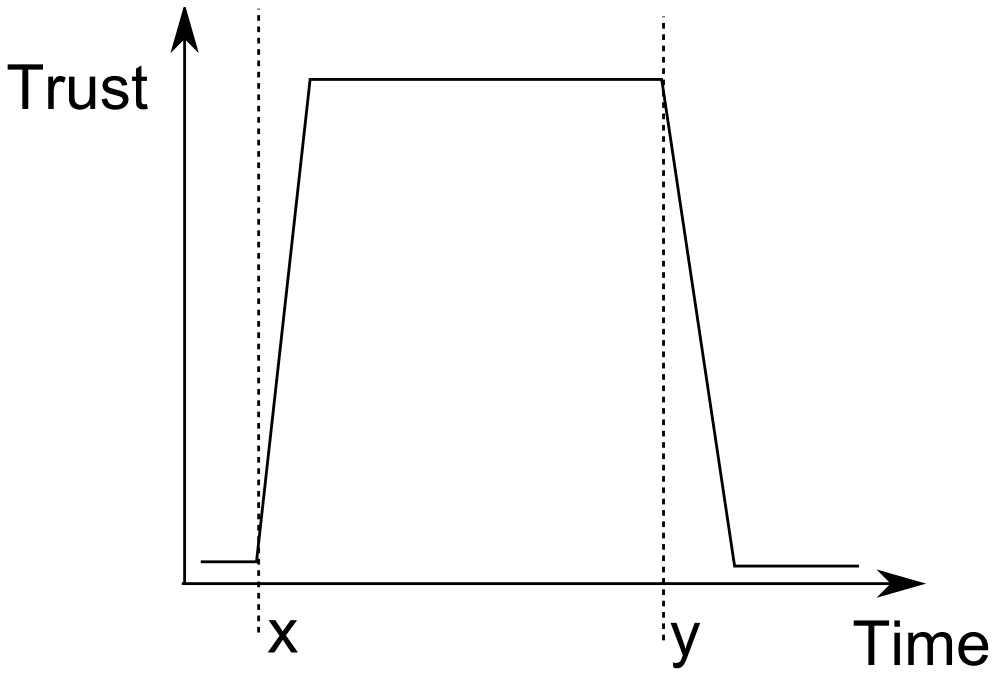


Figure 7.2: Trust level increasing and decreasing over time.

objects. When dealing with scenarios where objects must be migrated from a failing node to a new node, high capacity nodes are much more likely to be considered as receivers because of their ability to deal with higher load.

7.2.4 Acquiring Trust through Honesty

Ideally, the goal of the system is to run without faults, either caused by cheating, failures or load problems. A low capacity node that has successfully functioned for a certain period of time without fault is more trustworthy than a high capacity node known for returning inaccurate results. In addition, this type of reputation is influenced by the amount of processing done by a node: it is impossible to determine if a node is honest if it never processes any requests. Trust by honesty is especially important when recovering from faults caused by cheaters, as the system is most vulnerable to cheating during the recovery process.

7.2.5 Trust Across Sessions

When player activity is recorded and associated with an account, trust information can be gathered from several play sessions. Thus, it is possible to recognize players with a history of cheating and assign them a low level of trust. However, cheating players often bypass these security precautions by using different accounts, often stolen from other players. In addition, the usefulness of historical data for trust is limited, as the quality of the network connection of the player can greatly vary over time. In addition, a player may use his account on different computers, depending on his geographical location. This means that network and processing recources can also vary from one play session to another.

Chapter 8
Load Balancing by Dynamic Adjustment of Cells based over Obstacle-Aware Partitioning

In MMOGs, players can flock to a given location for many reasons. In some games, financial institutions, such as banks or auction houses are considered rallying points. In other games, special events are organized in predetermined cities, making these cities hubs of activity. Other times, players might discover easy access to a resources in a certain area, flocking by the hundreds once the word spreads. Unfortunately, many flocking behaviours cannot be predicted. These behaviours are problematic, since the load generated by a flock increases exponentially. To provide a responsive game experience, MMOGs must be able to dynamically adjust to deal with such situations.

The notion of load balancing has been explored for years in various fields. When dealing with load that is distributed over a 2D (or 3D) space, which is the case for most games, it is common practice to implement distribution schemes that split the world into a pre-determined number of square regions, or dynamically split the game world using a quad-tree-like division scheme. We propose a solution where we break down the world into a set of cells composed of *triangle* tiles. Using triangles allows the creation of cells that flow around obstacles in the world. These obstacles naturally

partition the interest regions of players in the game world. For instance, a long wall blocks the sight of a player. He is therefore not interested in objects located behind the wall.

Interest management for the content of a cell (or multiple cells) is assigned to a trusted node. That node will overload if there are too many players located in its cell. To deal with overload, cells can be shrunk by moving triangle tiles from the overloaded cell to an adjacent cell, reducing the number of players in that cell. In extreme cases, a cell whose hosting node is overloaded can be migrated entirely to another, more powerful node.

The effectiveness of the proposed load balancing solutions can be measured using Mammoth. By creating several hundred artificial players and having them flock to a given location, we can easily generate flocking scenarios that simulate what is happening in real game. These flocks trigger the load balancing algorithms which then allows us to measure their ability to cope with load.

The following chapter describes how Journey addresses load problems, and allows for large number of players to interact in a game world. Firstly, the notion of load is discussed. The chapter then focuses on how load can be dealt with and describes the proposed load balancing algorithms.

8.1 Load

Before addressing the problem of load distribution, the definition of load must be properly discussed. What is load exactly, and how should it be measured? One definition of load is any task that consumes hardware resources on that node. With this definition, load can be measured using a traditional approach, like measuring hardware constraints (CPU load, memory usage, number of page faults). However, load on a game can also be defined by the amount of game activity found on that node. In the case of Journey, game activity is generated by masters objects and the number of interested parties in those master objects.

8.1.1 Physical Load

Given the high number of processes running on a traditional game machine or server, it is impractical to monitor game load using game logic. Other processes, such as anti-viruses, consume a non-trivial amount of resource. Measuring hardware constraints is a much more efficient way of determining if a machine is currently overloaded, because it measures the resource consumption of all the processes currently running on the machine. It also happens that hardware monitoring is fairly easy to implement, as most operating systems already contain the necessary tools. For the remainder of this chapter, this type of load is referred to as physical load.

8.1.2 Logical Load

Unfortunately, physical load is not enough information for most partitioning algorithms. Although physical load reflects the workload found on a particular node, it contains very little information on what is generating load and how it can be distributed. Partitioning algorithms require information such as "Where is the load is concentrated?", or "How is the load distributed?" to better distribute load among different nodes. This information can be gathered by analyzing the game state, or the current game activities. However, implementing this type of load monitoring is difficult, as it requires the construction of an effective load model to represent and calculate the load. For the remainder of this chapter, this type of load is referred to as logical load.

8.1.3 Logical Load Model

The first step in establishing a load model is to determine which game elements generate load. In the case of duplication spaces, experiments have shown that the following elements generate load:

- master cells: because the node must do interest management for each master cell it hosts, as well as keep track of the objects moving in those cell. In addition,

a higher load value is assigned to active objects (objects controlled by a node and moving) as opposed to a passive object.

- replicated cells, because the node that hosts them receives updates about those cells.
- master objects, because the node that hosts them must process all actions done on these objects and make sure that every time the state of the master object is updated, the changes are broadcasted to the replicas.
- replicated objects, because the node that hosts them receives updates on those objects.

Based on experimentation, weights were assigned to each of these elements, and therefore the logical load of a node can be calculated as the weighted sum of these four numbers. This simple definition of logical load is used throughout the rest of the thesis. In the future, it could be interesting to experiment with more sophisticated logical load definitions, or to additionally use physical load sensors to obtain more realistic load estimates for nodes.

8.2 Dealing with Load

Several commercial MMOGs handle load in the game world by allocating a dedicated game server to take care of each part of the world. This is a very straightforward solution, but it does not deal well with flocking scenarios (i.e. when players converge to specific points in the world).

Flocking scenarios are fairly common in MMOGs. Some of these scenarios are permanent in nature, resulting from a design decision in the game. For example, banks and auction houses are fairly common meeting points in a game. Other flocking scenarios are sporadic. For example, a game master[1] can organize a scavenger hunt in a particular city. In this situation, the load increase for this area is easy to

[1]Employees of the company hosting the game, they are tasked with policing the game world and organizing special activities.

predict. However, some load increases cannot be predicted, as they result from player interactions in the game. For example, large fleet battles in Eve Online have been known to cause quite a slowdown in particular solar systems. CCP, the creators of Eve Online, took a creative approach to solving this problem by asking players to warn game masters in advance of where a large fleet will occur with high probability. This would allow CCP to host the particular solar system on a dedicated high-powered node.

As previously mentioned, Journey uses a cell-based system to distribute work across several nodes. Given the difficulty in predicting load spikes, Journey features functionalities to dynamically resize these cells and offload load to a different node. However, successfully dealing with dynamic load is a very complex task and requires two key elements: dynamic partitioning and load balancing.

8.2.1 Dynamic Partitioning

Static partitioning of a workload distributed over a 2D surface is straight forward. The world is divided into fix (pre-definted) partitions, which are in turn hosted on fixed nodes. Dynamic partitioning is more reactive, partitioning only occurs when needed, most often to deal with a load spikes. The question is, how and when to partition?

How to Partition?

In a virtual world, game objects and players are distributed in a 2D or 3D space[2]. Ideally, the game entities are equally distributed over the space. However, situations like flocking create non uniform distributions where players and game objects are highly concentrated in certain areas. Thus, any partitioning technique used must be able to deal with one or more concentrated areas (hot spots). This works explores two partitioning strategies, line-based and tile-based. Tile-based partitioning is favoured in Journey for reasons explained in the following sections.

[2]For the purpose of this work, only 2D space is considered. 3D games, where players walk on surfaces, can be handled just like a 2D space.

Line-Based Partitioning In a line-based partitioning scheme, the world is divided into non overlapping rectangles of arbitrary shapes. Initially, a single rectangle partition is used to cover the whole world area. That partition can then be split into two partitions using a line that cuts across its longest side (see figure 8.1). The exact position of the cut depends on the load in the cell, or else is drawn in such a way that it cuts the world in half. Although fairly simple to implement, there are some important limitations of line-based partitioning.

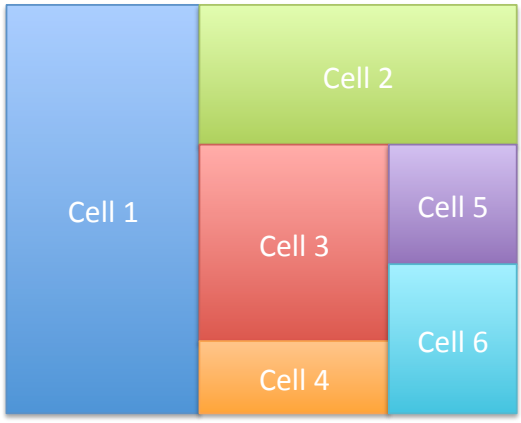


Figure 8.1: Space partitioned using rectangles.

Firstly, an ideal cut in a partition would distribute the load equally among the two new partitions. However, if that partition is suffering from a hotspot, the cut is most likely located in the middle of that hotspot (see figure 8.2). In that case, both partitions would still need to interact very closely with each other, given that adjacent objects still need to interact with each other. In this situation, migrating a partition to another node most likely results in very meager performance gains.

Secondly, although rectangle partitions are easy to divide, they are difficult to merge. Ideally, cells must be merged in the inverse order in which they were divided initially. This limitation can be dealt with by allowing cells of arbitrary polygonal shape. However, this can seriously reduce performance and complicate interest management, given that arbitrary polygons are much more difficult to work with.

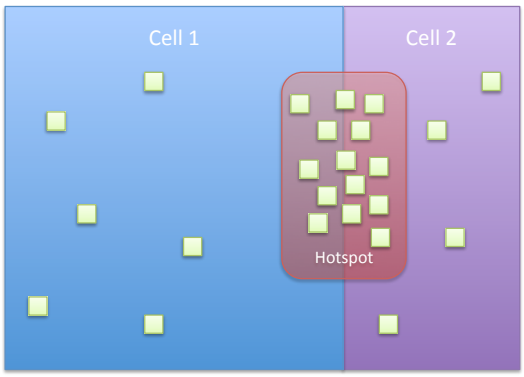


Figure 8.2: Partition split in the middle of a hotspot.

Tile-Based Partitioning In a tile-based partitioning scheme, the world is divided into a set of non overlapping tiles. These tiles can be of arbitrary polygonal shapes: squares, triangles, hexagons, etc. However, optimizations are easier to achieve if all the tiles have the same number of sides (for example, if they are all triangles). The initial partition is composed of all the tiles found on the map. Segmenting is achieved by assigning the tiles to different partitions. Ideally, all the tiles in a partition are connected, as split cells can degrade the performance of the interest management algorithm. When splitting a partition, simple algorithms just look at the current number of tiles in the partition. More elaborate algorithms determine the load on each tile and insure that the load is fairly equally distributed across the two new partitions.

Two types of tiles were evaluated with this framework, square tiles and triangle tiles. Square tiles were interesting because of their ease of implementation and their popularity in the research literature. On the other hand, triangle tiles had the distinct advantage that they can be generated to follow the topology of the world. In the end, triangle tiles were chosen for their efficient performance when dealing with interest management [BKV06].

When to Partition?

A key element in efficient partitioning is to decide when partitioning should take place. Should partitioning occur only when load must be distributed? Or would it be more efficient to pre-partition a space in anticipation of load problems?

Pre-partitioning a space has the definitive advantage of making it easier to migrate load when in situation of heavy load. The partitioning process can be quite complicated and require additional resources, which is exactly what needs to be to be avoided in a system under heavy strain. However, when using a pre-partitioning scheme, there is a performance penalty for managing multiple cells. This performance penalty is unnecessary for systems that have a low load. In the end, given the low cost of resizing partition by transferring tiles between partitions, pre-partitioning was not used. As such, partitioning only occurs when needed.

8.3 Load Balancing

Load is generated by both cells and objects, masters and replicas alike. However, master objects do generate higher load, making it undesirable to host a large number of them on the same node. Master objects can only be transferred to trusted nodes. Given that trust for cells and objects is different, two separate load balancing schemes are proposed.

Note that the proposed schemes should be considered load sharing algorithms, as they focus mostly on distributing the load before a node is overwhelmed [DZ03]. However, if a node is overwhelmed, the algorithms will still migrate load from the overloaded node to underloaded nodes. The proposed schemes assume that load and trust information is available globally. This can be offered by a central server, or stored in a distributed hashtable (DHT).

8.3.1 Burst Migration

Load balancing in Journey is feasible because of object location transparency: developers do not know whether a game object on their node is a master or a replica.

Objects can be freely moved from one node to another. The process of moving objects across nodes is called migration.

Migrations are often considered distributed transactions in the academic literature. The most common model is two-phase commit, where communication is done in two steps to make sure that both parties agree on the migration. Unfortunately, two steps migration is too slow to use in Journey.

Journey currently supports burst migration. This migration scheme simply transfers the object to the destination host, with no approval or confirmation. This provides incredible speed, although it is very unreliable. Discussion on how to make this algorithm more fault-tolerant is found in section 9.3.4. A description of the algorithm is found in annex A.

8.3.2 Load Sharing Master Cells

The proposed load sharing algorithm assumes that cells are composed of tiles, either triangular or rectangular. The system is initialized with a single cell, containing all the tiles. When a new node allowed to host master cells joins the system, it is initialized with an empty cell. Given that load(L) information is available globally to all nodes, we define the total load (TL_2) and the target load (AL_2) for nodes with level 2 trust in the system (S_2) as:

$$\forall x \in S_2, TL_2 = \sum L(x)$$

$$AL_2 = TL_2 / |S_2|$$

We define a buffer value, which is used to define which node is underloaded and overloaded. The buffer value is necessary to avoid nodes oscillating between the underloaded and overloaded status. Although the buffer value could be defined dynamically based on the numbers of node in the system and the total load, it is currently defined as a constant C. As such, we define the underload (UL_2) and overload (OL_2) values as:

$$UL_2 = AL_2 - C$$

$$OL_2 = AL_2 + C$$

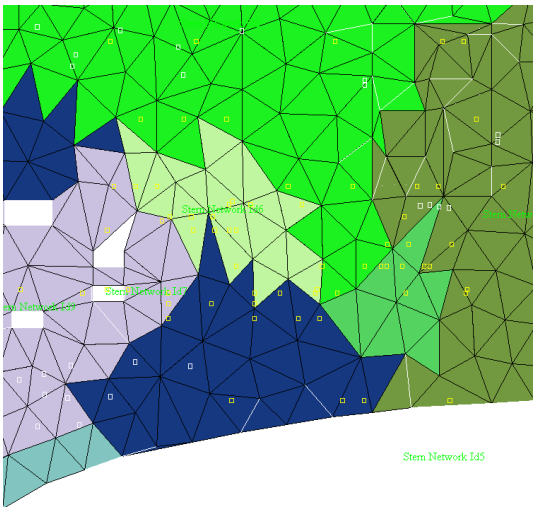
A node whose load is higher than the OL_2 value is overloaded, and will try to shed load to underloaded nodes. This is achieved by shedding tiles from the overloaded cell to a neighbouring cell.

The AL_2 value is determined by the number of nodes in the system, which means its value changes considerably every time a node allowed to host cells joins or leaves the system. The proposed load sharing algorithm does not attempt to fix load problems in a single round of transfer. Instead, it is designed to distribute load over a number of short transfers, distributed over time as to minimize the performance impact. As such, the system should stabilize once the number of nodes in the system remains stable.

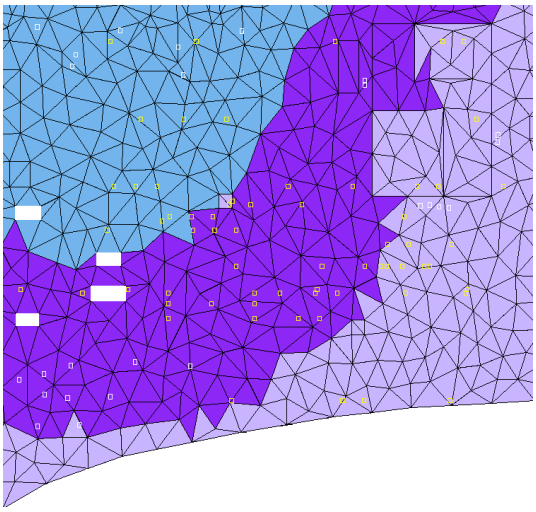
8.3.3 Which Tiles to Migrate?

The biggest challenge to load balancing cells is to choose which triangles should be shed to a neighbouring cell. Choosing the wrong triangles can create oddly shaped cells. Given that a node must know about objects in neighbouring cells, oddly shaped cells can greatly increase the load on the system.

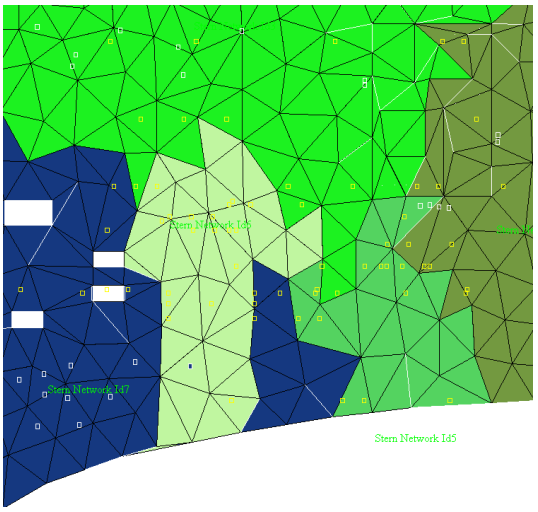
We started by implementing a simple tile selection algorithm, ran experiments to evaluate its performance, and then incrementally changed it to optimize the resulting cell shapes. The first iteration of the algorithm would randomly pick triangles on the border of the overloaded cell and transfer them to neighbouring cells (see figure 8.3(a)). Although very fast, the algorithm produced oddly shaped cells and was not effective. A more successful attempt was to grow cells outward, by transferring all triangles on the border of the overloaded cell to the adjacent cell (see figure 8.3(b)). This produced cells that were very aesthetically pleasing. However, the algorithm would always create cells that were shaped like ribbons, as new cells would always start from the same corner of the map. Attempting to grow cells from different outer



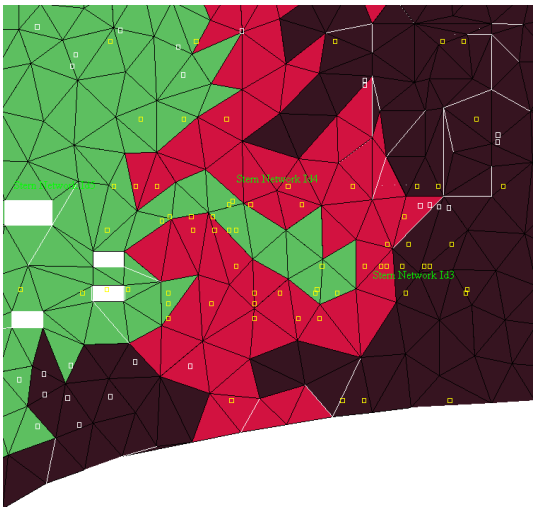
(a) 1st attempt, random distribution of tiles



(b) 2nd attempt, growing cells outward from side of map



(c) 3rd attempt, growing cells outward from different directions



(d) 4th attempt, adding constraint not to split cells

Figure 8.3: Experiments on load distribution algorithm.

edges (see figure 8.3(c)) had the nasty side-effect of splitting cells down the middle when dealing with more than two servers.

A fourth attempt was to add a constraint where a node would not give up a triangle if it split its cell into two (see figure 8.3(d)). However, this often created cells with long strips. Both strips and split shapes were terribly inefficient on the interest management system.

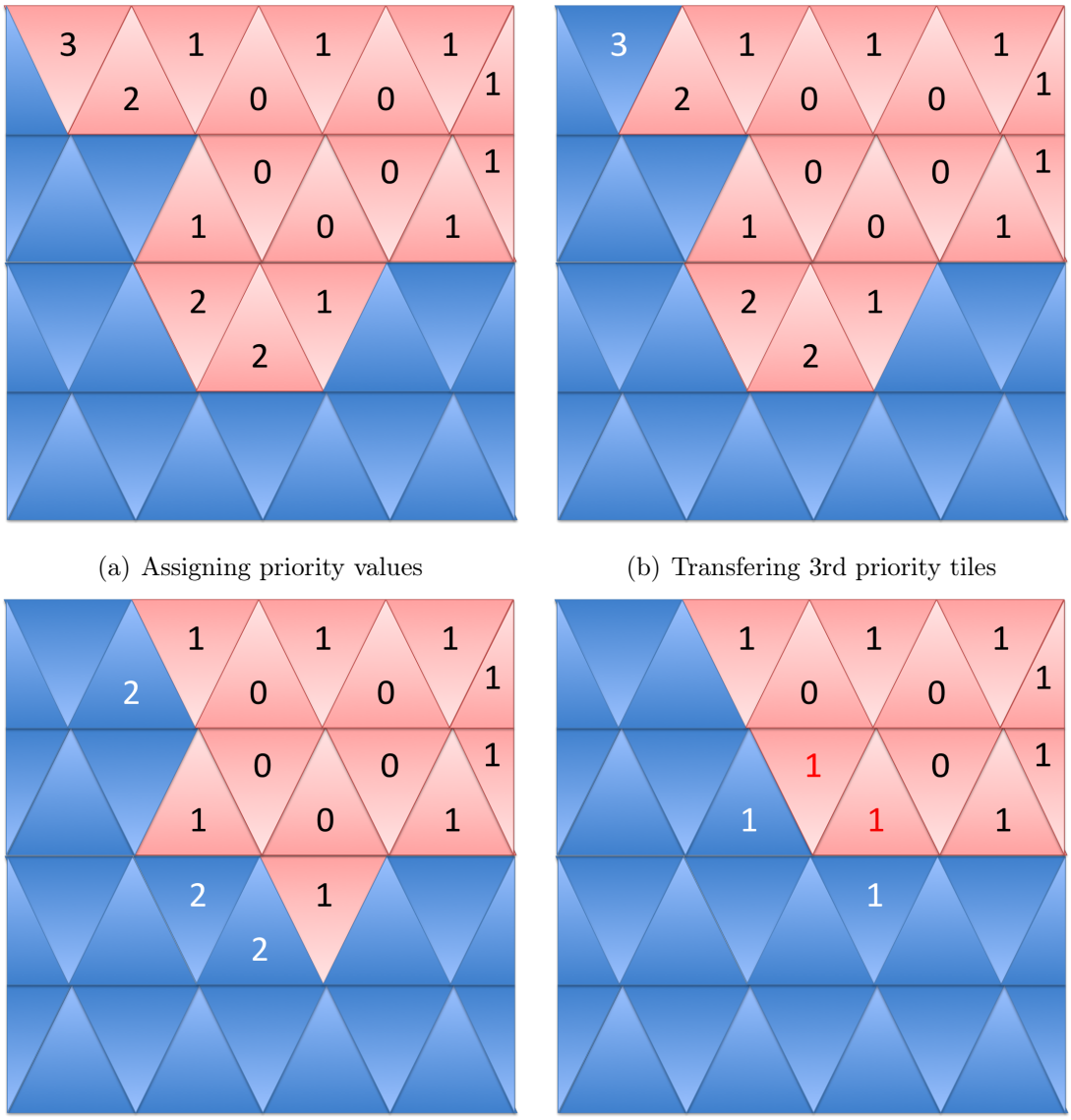
The algorithm that was finally chosen for Journey is slightly more complex, but still does not require elaborate calculations to be performed at run-time. For each tile, a priority value is calculated as follows: a triangle whose neighbours are all members of the same cell has priority 0. Otherwise, the priority of a triangle is the distance between this triangle and the closest triangle of priority 0.

This sorting is illustrated in figure 8.4(a), where triangles are sorted using priorities ranging from 0 to 3. In this example, the cell with red tiles is hosted on an overloaded node, which is trying to transfer some load to the node hosting the cell with the tiles in blue. First, the tile with priority 3 is transferred (see figure 8.4(b)). Then, the tiles with priority 2 are transferred (see figure 8.4(c)). The order in which these three tiles are transferred does not matter, as long as they are transferred before the priority 1 tiles. If the cell still needs to shed some load, then level 1 tiles are transferred. Tiles with only one connected neighbour will be given away first (see figure 8.4(d)). This helps make the cell shape smoother. It should be noted that transferring tiles of priority 1 will often increase the priority of priority 0 tiles, as they are losing a neighbour.

8.3.4 Load Sharing Master Objects

As with master cells, hosting master objects generates load. However, master objects can easily be migrated to other nodes. Let us define the notion of total load (TL_1) and target load (AL_1), this time for nodes with at least level 1 trust (S_1).

$$\forall x \in S_1, TL_1 = \sum L(x)$$



(a) Assigning priority values

(b) Transfering 3rd priority tiles

(c) Transfering 2nd priority tiles

(d) Transfering some 1st priority tiles

Figure 8.4: Demonstration of shrinking a cell.

$$AL_1 = TL_1 / |S_1|$$

We define a minimum load ML constant, where nodes with load higher than ML will try to migrate some master objects to other nodes. However, only nodes with load lower than AL_1 should be considered valid candidates for migration. Considering that nodes with level 1 trust should greatly outnumber nodes with level 2 trust, $AL_1 \ll AL_2$. Given that level 1 nodes are not trusted enough to host master cells, they are good candidates for master object migration.

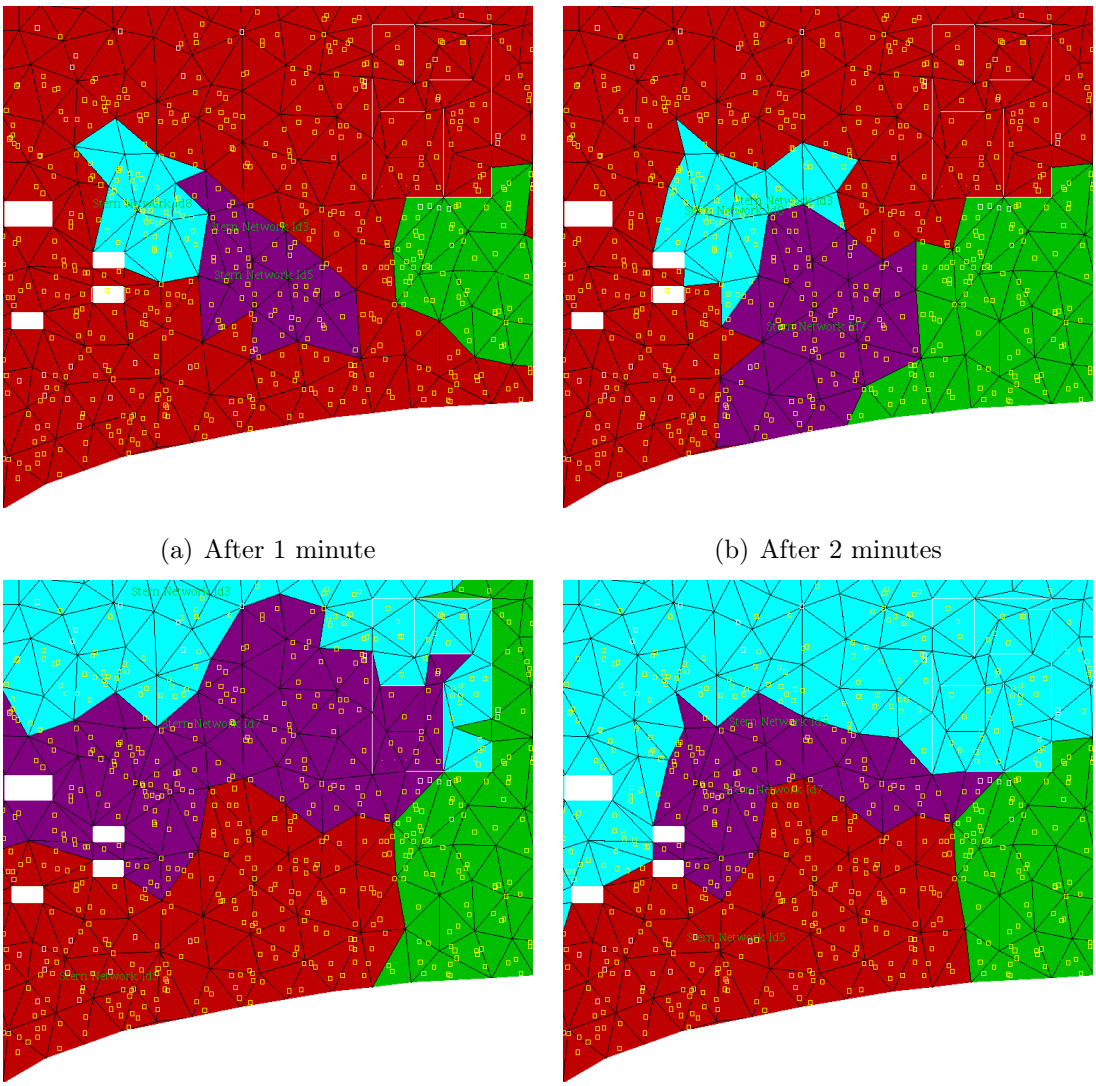
It can easily be argued that some objects are likely to interact with each other. One straightforward example is a player object : a player interacts frequently with the item objects contained in its inventory. It would be logical to keep these objects on the same node, for efficiency purpose. That aspect, however, is outside the scope of this work, and left for future work as discussed in chapter 15.

8.3.5 When a Trusted Node Joins the System

The proposed load sharing scheme has the additional benefit of fixing the bootstrap problem. When the system is initially started, the initial node, A, hosts all the master objects and one master cell containing all the tiles. When a new node B joins the system, it has no master objects and, if it is a level 2 node, a master cell with no tiles. However, the addition of a new node lowers the AL_2 value, causing A to consider itself to be overloaded.

- If B has the trust level needed to host cells, it will receive tiles from A, until the load of A is under the OL_2 value.
- If B has the trust level needed to host master objects, it will receive master objects from A, until its load is higher than AL_1.

After a short amount of time, the system will stabilize and both nodes will have roughly the same load. If a new node C joins the system, both A and B will shed load (either tiles or objects) until all three nodes stabilize.



(a) After 1 minute

(b) After 2 minutes

(c) After 3 minutes

(d) After 4 minutes

Figure 8.5: Demonstration of load sharing tiles with four level 2 nodes.

Figure 8.5 demonstrates the load sharing algorithm in action, as three level 2 nodes are successively added to the system. About 4 minutes are required for the system to stabilize, during which triangles are transferred from one cell to another. After the 4 minutes, the load on each server is approximately equal.

8.3.6 Leaving the System

When a node leaves the system in a controlled fashion (failure cases are only considered in chapter 9), it must transfer the tiles and master objects it is hosting to the remaining nodes. However, because the node is shutting down, it does not have the time for the typical multi-round load sharing algorithm.

Let us assume that node A is shutting down. Using the global load and trust information, node A determines that node B is the node with the lowest load and is able to receive both tiles and master objects. In this case, node B is called the successor node. As a first step, node A transfers its master cell to node B. The tiles in this additional master cell will be given away to the next level 2 node joining the system. As a second step, node A will migrate all its master objects to the successor node. This successor node will then be able to transfer any excess load using the regular load sharing algorithm.

Note that a node shutting down will immediately remove itself from the global load and trust information service. This is to prevent the node from receiving new load while trying to get rid of its cell and its master objects.

If the system must deal with multiple simultaneous shutdowns, this approach can be altered to select multiple successors, reducing the possibility of overloading any single node.

Chapter 9
Fault Tolerance and Cheat Detection within a Replicated Environment

---

There is a close link between the concepts of fault tolerance and cheat detection. In the field of fault tolerance, a component failure is defined as "an observable deviation from the component's specification". At its core, a cheating player can be considered an internal fault in the system, since the player's computer violates the game rules (i.e, the specification). Thus, the concepts of detecting faults found in the field of fault tolerance can potentially also be applied to cheat detection. Dealing with cheaters means isolating them from the game very quickly. Again, fault tolerance techniques can help us deal with this situation, as it is similar to error containment and isolation.

Before trying to detect and deal with faults, it is important to understand what can go wrong in the system. The first section of this chapter defines what is the correct behaviour of the system, by defining possible faults and the consistency model provided by Mammoth. The second section describes techniques used to detect faults, while the third section discusses some solutions to deal with faults. The last section presents how auditing can be used to deal with cheaters.

9.1 Consistency Model

The first step in dealing with faults is to properly define the correct behaviour of the system. In Journey, a change of state on a master object is atomic and defined as an operation. Note that an operation has a local effect, it only affects one node. When dealing with a distributed system, a key concern is keeping a consistent view among all participants. However, given that data cannot be instantaneously shared because of the physical limitation of communication technology, views will be inconsistent at one time or another. Thus, some inconsistencies must be tolerated. This is discussed in section 1.1.2.

A consistency model defines what are the minimum consistency guaranties provided by the system. In the case of Journey, the consistency model is a follows:

- When the state of a master is changed during an operation, the updated state of the master is eventually propagated to all non-faulty replicas.
- At the end of the operation, the master and all non-faulty replicas will have the same state.
- Updates to the state of a replica are executed in the same order as updates on the master object.

Note that it is assumed that the network communication layer is reliable and that messages are delivered within a reasonable amount of time. If messages cannot be delivered to a node, that node is considered faulty and is removed from the system. The network layer also provides basic security features, preventing messages from being altered during transit, or spoofing the source of a transmission.

However, the network communication layer does not provide broadcasting as an atomic operation. This means that if a node crashes while issuing a broadcast, it is not guaranteed that the broadcast will reach all the targeted nodes.

9.1.1 Possible Faults

A failure in the system is any deviation from its correct behaviour. This deviation is caused by a fault in a node. A better understanding of the possible faults in the system is key to designing proper fault detectors, and taking the necessary steps to recover the system.

The most common fault in a node would be the loss of communication with a node, either due to communication failure or because that node in question is overloaded. In the first case, the node is lost to the system and its assets should be recovered. The second case causes more of a problem, as it is impossible to detect that messages are simply delayed. The node can eventually rejoin the system, which is also a problem if another node has already recovered the lost masters. This leaves the system in an inconsistent state, as some masters will exist in two locations at a given time.

Other types of faults in the system are intentional, caused by a user in order to gain a game advantage, or to impede the play experience of other people. These faults are commonly referred to as cheating. For example, a client might send requests that violate the rules of the game, or might send requests related to objects not under his control. The cheats on the system can also be more subtle, such as falsely claiming ownership of an object, or requesting a migration that would move an important object to a cheating node.

It is important to note that some cheats are non-technical in nature. Social cheating has shown itself to be much easier to execute and more difficult to detect. However, such practices are outside the scope of this work.

9.1.2 Trust

Using the notions of trust and roles discussed in section 7.2, three types of participants are defined for fault tolerance purposes: nodes hosting cells, nodes hosting objects and nodes hosting nothing. For the purpose of fault tolerance, we restate these three types of participants as follows:

- Level 2: Nodes hosting cells : These are dedicated machines featuring redundant

hardware, hosted by the service provider. As such, they are considered highly reliable and very expensive to purchase. Thus, only a few of them are present in the system.

- Level 1: Nodes hosting objects : These are temporary participants (Level 0 nodes) that have gained trust. As such, they are considered reasonably reliable, although their level of reliability changes over time.

- Level 0: Nodes hosting nothing: These nodes are not trusted in the system and are not assigned any responsibility. They are more likely to leave the system after short periods of time. Level 0 nodes represent the majority of the nodes in the system.

Note that a node leaving the system when properly shutdown is not considered at fault. When leaving the system in a controlled fashion, a node transfers all its master objects and cells to another reliable node before disconnecting. However, disconnecting from the system in an uncontrolled fashion is considered a fault, as master cells or objects hosted on that node are considered lost.

9.1.3 Service Guarantee

The service guarantee proposed for Journey is that the system will recover if

`Scenario A: Any number of Level 0 or Level 1 nodes are lost.`

or

`Scenario B: A single Level 2 node is lost and any number of Level 0 are lost.`

It is important to note that both scenarios are not compatible. For example, if several Level 1 nodes are lost, the system cannot recover from the loss of a Level 2 node until the Level 1 nodes are first recovered. Also, if a Level 2 node is lost, the system cannot tolerate the loss of any other Level 1 or 2 nodes until the previous recovery process is complete.

9.2 Fault Detectors

Fault detectors are components which monitor participants for deviation in correct behaviour. However, the topic of fault detection is very broad and could be considered a thesis topic by itself. Thus, the elaboration of effective fault detectors is outside the scope of this work. Instead, this section focuses on some simple fault detectors.

A key concern for fault detection in Journey is to determine if active or passive detectors are more appropriate. An example of an active detector would be to regularly poll a node, to ensure it is properly functioning. This has the advantage that a faulty node can be detected before any errors occur in the system. However, active detectors generate load in the system, even when no faults are present. Passive detectors monitor the system for errors, and investigate the causes of such errors. This method generates little load in the system, because no action is taken if no errors are present. However, fault detectors do not work in a preventative fashion, they only react to erroneous behaviour. Both types of detectors were implemented and compared.

The first experiments with fault detectors were done with a simple active detector. This detector would monitor the incoming network traffic and record which nodes were sending out messages. A node that did not send out a message after a certain amount of time would be suspicious. Once a node was suspected, it would receive a "Keep Alive Request" message, which would be answered with an "I Am Alive" message. This was not often required, as the Interest Management component would send out regular updates to all other nodes. However, this method of monitoring was very load intensive, as it required the system to record the arrival time of every message. Thus, this method was quickly eliminated as a viable solution.

Passive solutions provide an interesting alternative, as they generate no load when no faults are present in the system. The key in developing an effective passive detector is deciding which behaviour in the system should be monitored for errors. The first passive detector that was tested monitored for timeouts in RPC calls. Although the presence of a timeout does not necessarily indicate the failure of a node, it is an indicator. The timeout could be provoked by a moment of unusually high load in the system. However, several consecutive timeouts would indicate that a node is no

longer reachable, and thus, faulty. However, this approach was problematic because a node does not necessarily execute calls on objects hosted on every other node in the system. Thus, failed nodes could easily go undetected for long periods of time. This was an unacceptable solution, as failed nodes should be detected without delay.

A second passive solution is to monitor the network engine for connectivity loss. When the loss of a socket connection is detected, it is propagated to the fault detector. In addition, the loss of connectivity is propagated to the other nodes participating in the system, as the disconnected node should be considered removed from the system. The propagation of this event depends on the topology of the network. This second passive solution was chosen as the primary fault detector in Journey.

9.3 Faults Handlers

Faults in the system are usually the result of the loss of a node. Recovering from that loss requires determining the responsibilities of the lost node and assigning them to different nodes. In this case, it involves determining which object and cell masters were hosted on the lost node. These masters are considered orphaned and new masters must be assigned on different nodes.

A key difficulty in handling these kinds of faults is determining which node should be assigned as the new master for an orphan. Nodes with replicas of the orphan can qualify to host the new master object. However, choosing that host is difficult, as there are no guarantees to when the remaining nodes will be advised of the loss.

Distributed election or consensus algorithms are possible solutions to this problem, as they allow the nodes to work out themselves who can be the new master. However, given that the messaging mode is asynchronous and that the environment is unreliable, there are no election or consensus algorithms that would allow the remaining nodes to agree on a single new node. Even if timers were used to make the messaging mode synchronous, any election or consensus solution would have too many messaging rounds.

Thus, an effective recovery solution would need to pre-assign the recovery responsibilities, thus avoiding the need for distributed consensus. In the case of object masters, they are recovered by the node hosting the corresponding master cell (see section 9.3.1). To deal with the loss of cell masters, cells are linked togeter as a circular chain. When a cell master is lost, its recovery is handled by the node hosting the previous cell in the chain (see section 9.3.2).

Although this strategy is effective in most cases, it is possible for a node to crash while it is broadcasting a state update. In this case, it is possible that the state update was not properly broadcast to all replicas. However, it is important that the system recovers to a state were both master and replicas have the same state (see section 9.3.3).

9.3.1 Default Recovery of Object Masters

As previously mentioned, all objects are physically (x,y) located in a cell. A node will have a replica of every object physically located in the master cell it is currently hosting. These replicas can be used to recover objects in case of a failure.

When a node hosting master objects fails, all nodes in the system are advised. Any replicas with their corresponding master on the failed node are tagged as orphans. Level 2 nodes will then check for any orphans located in their master cells. Those orphan replicas are switched to masters.

Note that this recovery scheme only works if master objects are not hosted on the same node as their cell master. This is not a problem, as this constraint can be easily enforced using the load balancing system.

Once a master object has been successfully recovered, it is immediately migrated to a different node. The system will not be considered in a stable state until all recovered master objects have been moved. Otherwise, the loss of a node with "unmoved" violate the service guarantee put the system in an unrecoverable state.

9.3.2 Cyclic Recovery of Cell Masters

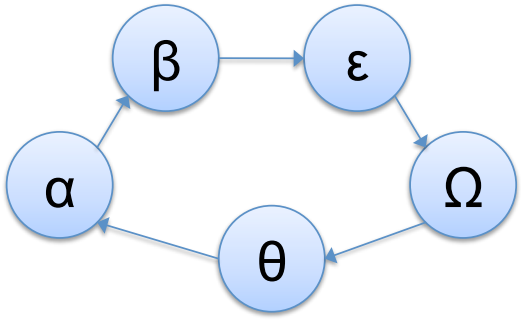
As previously mentioned, each object is physically (x,y) located in a given cell. In a non-replicated distributed system, finding the corresponding cell of an object would require broadcasting a request to all nodes hosting a cell, and waiting for a response from the node hosting the corresponding cell. This is unacceptable in Journey, which requires up-to-date information on all cells in the system in a timely fashion. Instead, every cell in the system is replicated on every level 2 node. Finding an object's corresponding cell only requires iterating through the cells.

The cell replicas simplify the recovery process given that every node has the necessary data to repair the orphan cell. Cells are linked together in a cycle (see figure 9.1). When dealing with orphaned cells, the owner of the precedessor to the orphan cell is responsible for restoring that cell.

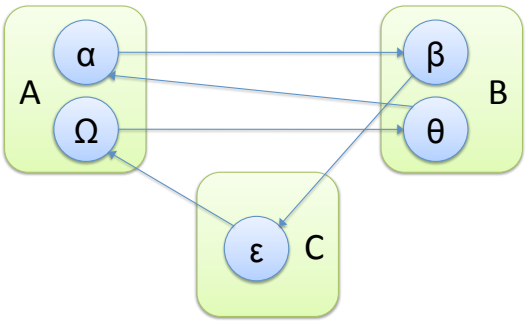
The complete algorithm can be found in annex A.4. If multiple sequential cells were stored on the same node, they will be restored, one at a time, by the node pointing to the first cell in the sequence. By restoring the first cell in the sequence, that node is now responsible for the second node in the sequence. This repeats itself until the complete sequence of node has been restored. As long as one node in the cycle is still functioning, the cell masters can be recovered.

Restoring cells sequentially might appear to be an inefficient solution, given that the loss of several sequential cells would require multiple rounds to restore the system fully. However, the temporary loss of a cell master is not to problematic to the system, as it simply prevents interest management from properly distributing new objects to nodes. Although some nodes will be missing replicas, the system will continue to operate according to specifications until all cells are properly restored. Since players cannot see or interact with objects that are not replicated on their node, the service interruption will most likely be invisible to the players.

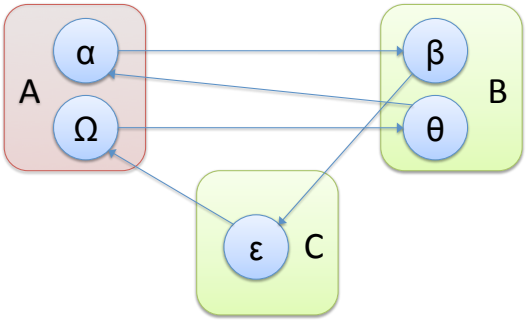
Once a master cell is recovered, a message is sent to all nodes announcing the new owner of the cell. Nodes hosting master objects located in this cell will re-register these objects with the cell master.



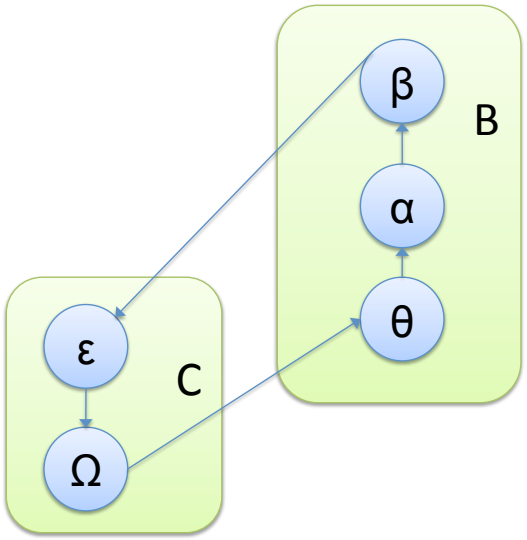
(a) Cell objects organized in a cyclic fashion.



(b) Cells are distributed on 3 nodes: A,B,C.



(c) Node A fails, orphaning 2 cells.



(d) Node B and C recover the orphaned cells, keeping the cycle intact.

Figure 9.1: Cyclic recovery of 2 orphaned cells.

9.3.3 Dealing with Inconsistent Replicas

As mentioned previously, updating the state of the various replicas in the system is not an atomic operation. If a node fails while broadcasting one such state update, the update could possibly not propagate properly to all the replicas. This would leave the system in an inconsistent state, since not all replicas have been properly updated. This is even more problematic for recovery given that there is no guarantee that the node handling the recovery has the latest state.

It is important to note that this situation only arises if a failure occurs during state broadcast. Given that the failure of a trusted node is considered a rare occurrence, the failure of such a node at that exact moment is considered even more exceptional. Thus, any solution that deals with inconsistency should have a very low impact on performance, given the very low likelihood of such a situation occurring.

When updating the state of objects, regardless of the presence of faults, the the update must be either applied to all replicas, or to none of them. Four possible scenarios for dealing with inconsistent replicas are proposed:

- **Scenario 1: Do nothing** This is the easiest solution to implement, as no monitoring is done when an object is restored. If the failure occurs, the object will not be restored to its latest state. However, all replicas will be correctly restored to the state before the update.

- **Scenario 2: Safe Locking** This is the safest solution by far, but very slow. All copies of objects are locked before a state update is sent out. The lock is then only remove once all updates have been applied. In the case of a crash, the state of all replicas can be quickly reverted and the locks are removed.

- **Scenario 3: Best Effort** Updates are sent first to the replica on the cell master, in the hope that at least that update will be applied. However, no additional effort is done to maintain a consistent state. In case of a fault, all replicas are updated to the state of the new master, which might or might not have received the update.

- **Scenario 4: 3rd party monitoring** A third party monitor is used to monitor the state changes of the object. If the state of the object must be recovered, the 3rd party will query all replicas to make sure the latest version is restored. At the end of the recovery, all replicas will have the latest state.

Given the very low likelihood of this failure, scenario 1 seems like the preferable solution. Scenario 2 and 4 create a lot of overhead in the system, while scenario 3 cannot be used with the current network engine of Mammoth.

9.3.4 Fault-Tolerant Burst Migration

The burst migration algorithm presented in the previous chapter is, by definition, not fault tolerant. If the destination node does not receive the object, or refuses the object, the master object is lost and only replicas remain in the system.

Fault-tolerant migration in Mammoth involves three parties: the sender (A), the destination (B) and the node owning the cell master containing the object being migrated (C). If node A or B crash during the migration process, node C will recover the object. However, the migration will not be interrupted if C crashes during the migration.

Given the network model and the reliability assumptions of the previously defined consistency model, we can assume that network messages are never lost and are delivered within X seconds, where X seconds is the maximum time allocated to a message round.

To make burst migration fault-tolerant, we add a messaging round at the start of the migration. First, node A informs node C that it is migrating the object. Node A then burst migrates the object to node B. Node B must then send an updated copy of the object to node C, informing it of the successful migration. If node A is unable to migrate the object, it sends a message to C informing it that the migration is cancelled.

When node C receives the notice of the migration, it waits for 2X seconds. If it does not receive a confirmation from B, or a cancelation from A, the migration is

considered failed and it recreates the master object. A complete description of this algorithm can be found in annex A.2.

9.4 Auditing

As previously mentioned, cheating players can be compared to faults in the system. They disrupt the normal operation of the system by introducing variations in how actions resolve, either to their advantage, or to simply ruin the play experience for other players. The fault detectors presented in the previous sections can not detect these malicious parties, since the faults occur at the execution level of the game, not the node itself.

In order to detect violations of the game rules by malicious player nodes, Journey incorporates node auditing capabilities: since the game state is encapsulated in master objects, a cheating player must alter the way methods of a master object change its state. When auditing is enabled, method calls are executed both on the target node and the auditing node (see figure 9.2). A successful method call should yield identical results, confirming that the method was successfully and honestly executed. The return value can also be audited, ensuring that the target node returns the proper value.

9.4.1 What and When to Audit?

Not every remote method call requires auditing. The need to audit and the frequency of the auditing is determined by the importance of the method call and the effect it has on the system.

For example, a method call used to alter the colour of a player's shirt has little importance and does not need to be audited. However, a method call for the movement of a player might require some auditing, as cheating on the location of the player can have a negative effect on game play. Critical method calls, such as those related to trading items, might require constant auditing, as cheating on them would have major negative impact to the game.

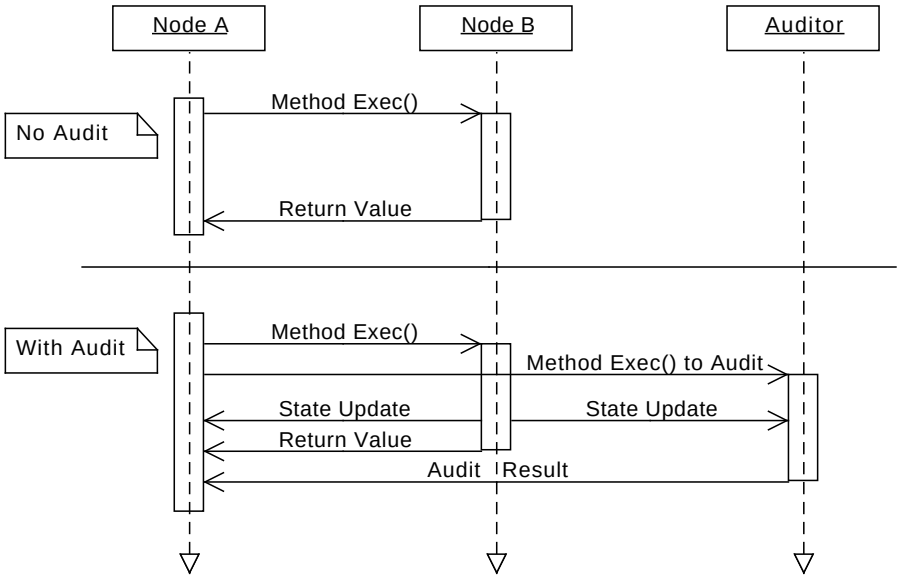


Figure 9.2: Example of Remote Method Executing with and without Auditing.

The frequency of the auditing can also be influenced by the trustfulness of the node hosting the master object. For example, a master object stored on a level 2 node never needs auditing. On the other hand, a newly promoted level 1 node, or a node with a history of faults are likely candidates for an increased amount of auditing.

Note that only deterministic methods can be audited using the proposed strategies. A method executed on two different hosts should have the same return value, and should leave both objects in the same state.

9.4.2 State History

Depending on the auditing strategies, there are no guarantees on when auditing requests will be received. It is quite possible that an auditing object might receive a request for a state change that has already been applied. Thus, the auditing object would not have the past state to execute the audit request.

This problem can be solved if the auditing object records the state changes over

time. By keeping a history of the last states, the auditing object can execute any recent auditing requests. To save memory, state data is serialized before being stored. Historical data that is never needed is never deserialized and discarded.

9.4.3 How to Audit

There exists a tradeoff between how intrusive and how effective the auditing is. Auditing is very inconvenient when it causes a slowdown in the normal operation of the game. However, simple auditing procedures with low overhead are completely useless when they are not protected from tampering. As such, the next sections present three auditing schemes, each with a different level of intrusiveness and effectiveness.

Direct to Master and Auditor

This is by far the less intrusive auditing strategy presented in this section. When a method call needs to be audited, an audit request is sent by the source to the auditor, bypassing the master. Once the master returns, the return value is also sent by source to the auditor, which can use this value, in addition to the normal state update, to determine if the method call was truthfully executed.

This strategy is the fastest of the presented methods, as it does not require any additional messaging rounds. However, it is also the most vulnerable to abuse, as there is no way to insure that a client is truthful. For example, a callee can send a different call to the master object and the auditor, forcing a false auditing result. The callee can also return a false value to the auditor, also creating a false audit result. Fortunately, malicious clients can usually be detected after a certain time, as only their auditing requests will fail.

Auditor Proxy

In this auditing strategies, method calls that require auditing are sent to the auditor, which in turn forwards them to the master object. The process is transparent, the master object believes that the original request came from the auditor. This means the return value is also sent to the auditor, which returns it to the original callee.

This auditing strategy is much more reliable, as the callee cannot pass false information between the auditor and the master object. However, the price of reliability is pretty steep, as the time required for an audited method call is doubled.

Forwarded to Auditor

A hybrid of the two previous strategies, audited method calls are forwarded to the auditor, which in turns forwards it to the master object. However, the method call is forwarded with the callee as the origin, ensuring that the return value is returned directly to the callee. The auditor then only uses the state change to audit the method call.

This strategy adds one message round to the method call, but the callee cannot maliciously falsify audits. It should also be considered less reliable than other strategies, as the auditor does not receive the return value and cannot use that information to validate the call. This scheme is currently used in Journey/Mammoth.

9.4.4 The Auditor

Auditing allows a node to suspect another node of cheating but it does not confirm it. In fact, it is very difficult for a given node to absolutely conclude that another node is cheating. It might even be unfair to convict a node given the evidence of a single auditing node, unless that node is absolutely trustworthy.

A possible solution is to forward auditing calls to different auditors. If a group of distinct auditors each detect a particular node as a cheater, then it is most likely a cheater. However, as previously mentioned, voting algorithms in this kind of network environment are very difficult difficult to implement.

Thus, solutions with a single auditor are preferred. Cell managers are an obvious choice for auditor, as their level 2 trust ensures their trustworthiness. Unfortunately, cell managers are typically very loaded, making them less than ideal candidates.

9.4.5 Dealing with Good and Bad Behaviour

Accusing a node of cheating is a tricky endeavour. Ideally, cheaters should be swiftly removed from the system, as they impact on the play experience of an honest player. However, accusing honest players of cheating can also have a serious negative impact on the game.

A player cannot be accused of cheating after a single violation. In addition, even if the auditor is trustworthy, it can still be faulty. One approach is to assign every node a starting trust score, and to decrease that score every time an audit fails. If the score drops below zero, the node is declared a cheater.

That same score would increase over time, as long as audits were successfully completed. Once the score reaches a certain value, the node would be promoted. Note that this promotion should not be permanent, as previously mentioned in section 7.2.2, the likelihood of the player disconnecting increases with time.

For experimental purposes, two simplified decision algorithms were designed and integrated in Mammoth. The first is very aggressive and declares a player a cheater after 6 failed audits. There is no notion of time and failed audits are never forgiven. The second is very conservative and only declares a player a cheater if more than 5% are failed, and this only after at least 50 audits have been done. This means a player cannot be banned for failing less than 3 audits. Although the second algorithm allows for more faults as times goes by, it also takes more time to fail a player that starts cheating immediately.

9.4.6 Dealing with Cheaters

The most common consequence when detecting a cheater is to banish him from the game. This is particularly easy to implement when combined with an efficient fault recovery solution. Cheating nodes are declared as faulty nodes and are removed from the system. Since the data contained on that node is untrustworthy, it is simply ignored in the recovery process. Instead, the object might be rolled-back to an earlier version.

Most multiplayer games (both massive and not), require a player to sign up with

an account. Privileges for new accounts are usually limited, until a player can prove himself honest. In addition, cheaters can be uniquely identified and easily removed from the system. With this protection scheme, cheaters will prefer to cheat using a stolen account, as to not risk their own account.

Part III

Mammoth

This part focuses on Mammoth, the framework used to implement and test Journey. Creating Mammoth itself represents the effort of numerous students over the last 5 years. Their work is described in chapter 10 . The result is a framework with a modular and flexible architecture, which is presented in chapter 11. The following chapter explains how Journey was implemented and integrated in Mammoth. Finally, chapter 13 discusses the various experiments done to evaluate Journey's performance.

Chapter 10
The Story of Mammoth

Mammoth is a massively multiplayer game research framework. It was created as a collaborative project between a group of McGill professors and students in early 2005, and has evolved considerably during the last 5 years. The author of this thesis, Alexandre Denault, was lead architect from May 2005 to September 2005 and from May 2006 to March 2010.

The story of Mammoth can be broken down into periods delimited by the summer semesters. Summers typically favoured large development groups composed of a mix of undergraduate and graduate students, working on projects and improving the core components. Active developers during the rest of the year were mostly graduate students, working on their thesis or graduate projects. An understanding of Mammoth's history is critical to understanding how its architecture evolved over time.

10.1 Summer 2005, the First summer

Development of Mammoth officially started in May of 2005. The project was headed by Alexandre Denault, a master's student under the supervision of Jörg Kienzle. This student also wrote the graphic library that first powered Mammoth, MinuetoGL, an OpenGL extension of the Minueto graphic framework [DK06]. Pierre Marieu, a French exchange student, developed the core classes of Mammoth, under the architectural supervision of Alexandre. The initial version of Mammoth was client/server, allowing

for a single server and up to 30 clients. All interactions between clients and the server were routed through a single `Game` class.

Client/server communication was provided by Mammoth's first networking engine, developed by Alfred Leung. Although the initial version of Mammoth did not explicitly feature interest management (IM), it did have some basic IM functionalities. The game world was divided into zones, and actions were classified as either major or minor. A client would receive both major and minor messages for events occurring in its zone, and major messages occurring in neighbouring zones. The server would track the position of clients and subscribe them to the appropriate channels, either with a major or full subscription.

One key challenge in game development is to have adequate tools to generate content for the game. Loc Bui developed the first content editor for the game, a particularly difficult challenge as the Mammoth game engine did not exist at that point. In addition, Mammoth development required the exploration of several new Java technologies, such as NIO, Webstart and the Sound APIs. Alexandre Quesnel was in charge of researching these new technologies and integrating them into Mammoth.

The first working prototype, known as version 1, was completed in August 2005. This version of Mammoth was simple; players could walk around the map and pick up objects. However, it served as the basis for all following research. Alexandre Denault resigned as the head of Mammoth, as he had completed his Master's degree. He was replaced by Jean-Sebastien Boulanger, a master's student under the supervision of Jörg Kienzle and Clark Verbrugge.

It should be noted that during this summer, Mammoth was known as project Martlet. This was a place-holder name, until a suitable project name could be chosen. After a few days of fierce voting, the name Mammoth, originally suggested by Denis Lebel, was chosen. Runner up suggestions included Virgo, Medici and Ensemble.

Figure 10.1: Mammoth Client in 2005

10.2 Fall 2005 and Winter 2006

Although the first Mammoth prototype was considered a success, its architecture had some serious limitations. Mammoth used a communication system that allowed clients to have either major or minor subscriptions to zones. This regulated the amount of messages a client would receive for a particular zone. Unfortunately, the system propagated too much data on the network. Clients were receiving useless updates about objects located way outside their area of concerns. Thus, Jean-Sebastien Boulanger added the notion of interest management to Mammoth. In addition, Jean-Sebastien also completed a major refactoring job, separating several of the core component classes into interfaces and implementation classes.

Nadeem Khan, a master's student under the supervision of Bettina Kemme, starting working on a distributed Mammoth prototype that would allow the server load to be distributed across several machines. His strategy was to assign responsibility of zones to different servers, and have clients communicate with those different servers.

Although this distributed prototype was successfully implemented, it diverged too much from the main version of Mammoth for the changes to be integrated.

At the same time, Russell Spence, a physics undergraduate student, developed the first and still current physics engine, as part of a Phys-489 special project. The main contribution of this project was a fast and efficient collision detection engine, which is critical to any game project.

Finally, Jeremy Claude and Marc Ovidiu focused their effort on building a new content editor, as part of an undergraduate computer science project course. Using a modular design left by Alexandre Denault, they used the game engine to create a content editor with an interface very similar to the one found in the actual game.

10.3 Summer of Code 2006

Summers, in the Mammoth team, came to be known as "Summer of Code". The name is inspired by Google's "Summers of Code", where programmers can get funding to work on their favourite open-source application.

The first Summer of Code saw an unprecedented number of coders working on Mammoth. Among the returning coders, both Jean-Sebastien and Nadeem continued to work on their respective projects. Deciding to start a PhD at McGill, Alexandre Denault rejoined the project as the head of Mammoth. In addition to coordinating all the work done on Mammoth, Alexandre did some major improvements on the graphics engine. Jeremy returned to the Mammoth team to finish his work on the content editor, this time joined by Jessica Guo and Yannick Thiel. While Jeremy focused on the user interface of the content editor, Jessica mostly worked on the tools needed to create items and add them to the map. Yannick's work centred on a wall drawing tool (see figure 10.2), which would combine overlapping walls in an aesthetically pleasing way.

That summer, Michael Hawker, Nicolas NgManSun and Marc Lanctot joined the team. Michael was interested in the item creation process in Mammoth. Early on in his work, he noticed that most items in Mammoth were similar. For example, the

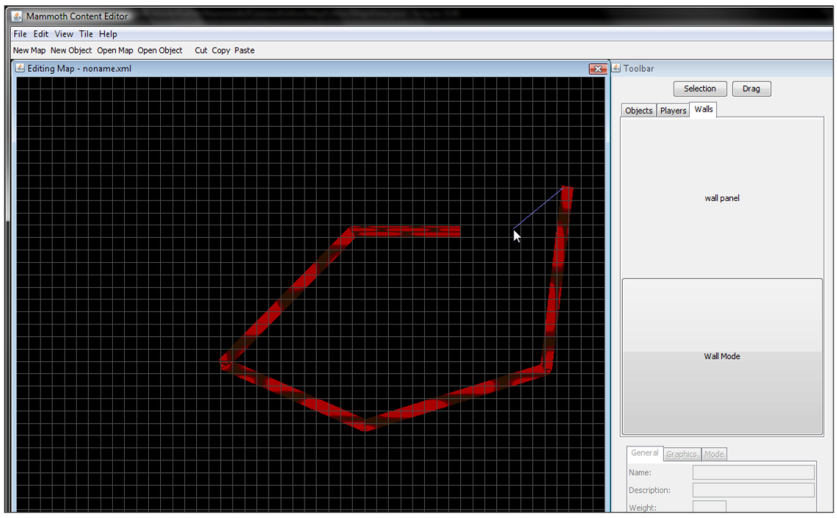


Figure 10.2: Wall drawing tool, developed by Yannick Thiel

Mammoth game map Town19, used in most experiments, contains almost 40 identical flowers. Michael proposed that items could be defined with a type system, where flowers could be defined as a type, and that flowers in-game could be instantiated from that type. Further work on this subject demonstrated that a hierarchy could also be found in this type system. For example, both wooden and metal bookcases shared similar properties. Michael's other research interest was to add purpose to Mammoth. His work introduced the notion of sub games, where players could join a mini-game that would be played on the world map. This quickly introduced concerns on how different games would interact with each other.

Nicolas' and Marc's work both aimed toward adding artificial intelligence to Mammoth. Nicolas' concerns were path-finding on a large scale map, as found in Mammoth. The idea was to use the zone system in Mammoth to determine the path to take at a macro level. That macro path could then be broken down to several smaller paths, and solved using more traditional approaches such as A-star [Les05]. During the summer, Marc's work mostly focused on sub games and Orbius, providing him with a test framework for the path-finding algorithms.

Orbius was a capture-the-flag type of game, designed by Marc, Michael and Alexandre. The goal was for players to collect orbs of different sizes for their team and

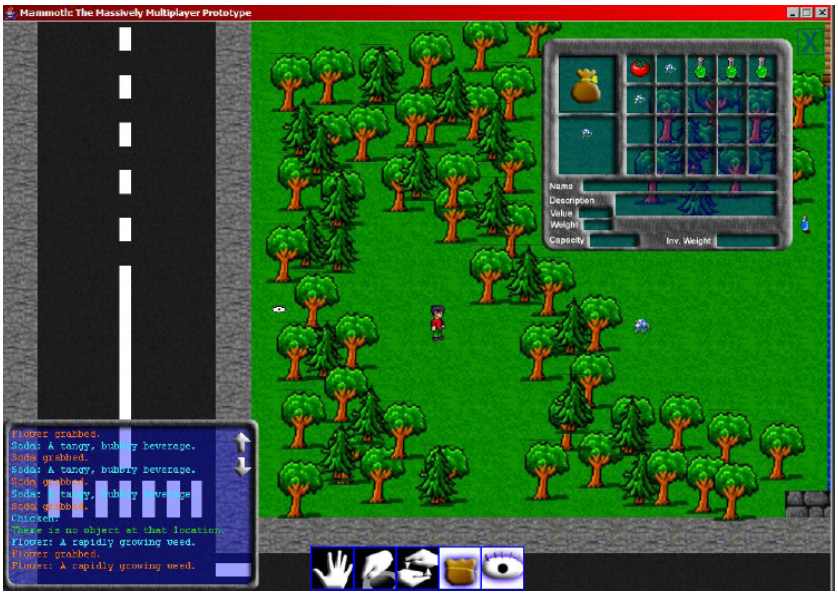


Figure 10.3: Mammoth Client in 2006

bring them back to their home base. To prevent opponents from achieving the same goal, players could tickle their opponents to steal opponent's orbs and hide them. In June 2006, data was gathered from 20 players playing this game for 2 hours. This data was used in research relating to points of interest [Lan05] and path-finding [LSV06].

10.4 Fall 2006 and Winter 2007

This development period was mostly dominated by the AI developers, Marc Lanctot, Jonathan Li On Wing and Adrian Ghizaru. Marc Lanctot designed an infrastructure to easily add NPCs (non-player characters). A key element of this design is that NPCs could either be run locally on a Mammoth server, or as a separate client. When developing an AI, it was very convenient to have that AI run locally. However, large-scale distributed experiments require NPCs to fonction on different nodes as stand-alone clients. Jonathan and Adrian used this infrastructure to work on a learning NPC AI. Their goals were to have the AI record the behaviour of a player, analyze it, and have the NPC reproduce some of that behaviour in the characters once the

player logged off.

During the Fall 2006, Mammoth was revealed to the world, as part of a poster presentation at Cascon 2006. Although publications on Mammoth were already available, it was the first time Mammoth was shown in public in an organized environment. The reception was very positive. In addition, a Mammoth server was setup locally, so that people at the conference could login to Mammoth and try it out themselves.

This period also saw the introduction of Mammoth's most important debugging tool, the Web Monitor. Distributed consistency bugs are one of the toughest issues to handle when programming. However, typical debugging tools are not designed for distributed debugging. In Mammoth, the Web Monitor allows a developer to see the state of all the game objects as the game is running. Each client will spawn an internal web server, making this information available from any computer using a web-browser. A developer can thus compare the live state of the different participants simply by switching through different browser windows. The idea was first developed by Alexandre and further enhanced by Michael who found the tool invaluable in fixing bugs in the items and sub games architectures.

10.5 Summer of Code 2007

Summer of 2007 was an interesting period for Mammoth, as improvements were done on every module. Noteworthy additions to the team were Arianne Perpignani, Valérie Ngo and Édouard Lanctot-Benoit, three arts interns from Collège Jean-de-Brébeuf. Arianne and Valérie worked on revitalizing the 2D artwork in Mammoth, while Édouard designed some 3D models of objects found on the McGill campus.

After working over a year on sub games, Michael had acquired a good understanding of the architectural requirements of sub games. Thus, he designed a more generic and modular interface, allowing for quicker development of sub games. Indeed, using this interface, a simple sub game, such as Flower Recovery, which randomly spawns flowers, could be developed in less than a day. This generic interface was based on message interception, where a pre and/or post processing stage could be added to any

Figure 10.4: Mammoth Client in 2007

message handler. Sadly, this design was not compatible with the idea of distributed servers, as there is no guarantee to where objects related to the game are stored (items, players, etc.).

Some serious improvements were done to the path-finding when Joachim Despland joined the team. His work focused on implementing a more elaborate A-star pathfinding algorithm. Joachim's algorithm would try to find a fast initial solution using very large samples (i.e. big squares) to find a solution. Before the path was found, the algorithm would return a best guess so that the player could start walking. The algorithm would then look for a better solution using smaller search samples (i.e. small squares). Once the better solution is found, the initial solution was discarded and the player continued moving. Experimentations have demonstrated that the initial guess was very rarely wrong in setting an initial walking direction.

During the previous Fall and Winter term, many optimizations were done on Mammoth to increase the player capacity, which was about 50 players at that time. However, the current implementation of the publish/subscribe was simply not fast

enough to expand beyond that limit. Thus Mathieu Couturier set out to study every existing publish/subscribe system that would be compatible with Mammoth. Although he was unable to stay all summer, he did produce a very complete study on existing pub/sub systems compatible with Java applications. His work was continued in the Fall by Dominik Zindel.

At the same time, a new project on persistence in Mammoth was started by Kaiwen Zhang. One key feature of MMOGs is their persistent nature: the state of the world is not loss if the server is restarted. Kaiwen's work raised some interesting issues, such as the different levels of importance for different actions. For example, trading an item requires a high-level of consistency (to avoid cheating). This kind of action needs to be recorded as soon as possible, possibly in a transactional fashion. However, walking around the map has a fairly low level of importance, and position data could be only occasionally recorded.

Mammoth's user interfaces, long in need of a serious overhaul, were beginning to be a limitation for several projects. An open source project, Fenggui [Sch09] was released at that time and allowed for rapid development of a swing-like user interface under any Java OpenGL plate-form. Wisam Al Abed and Ting Sun took on the project to integrate Fenggui into MinuetoGL, and then later into Mammoth.

10.6 Fall 2007 and Winter 2008

By Fall of 2007, the Mammoth code was two years old and in a serious need of refactoring. Fortunately, the Mammoth team had also learned a lot about modular framework development and managing large-scale projects. Although all modules were refactored, the network engine received the most important changes. Firstly, its architecture was completely redesigned. Detailed information on the refactoring Alexandre did on this module can be found in section 11.5. In addition, the old network code, based on Java NIO technology, was proving itself to be a bottleneck. Thus, the network code was migrated to a new Java network library, Apache Mina [Min10]. Alexandre replaced the old network engine by two new ones: Stern, which uses a

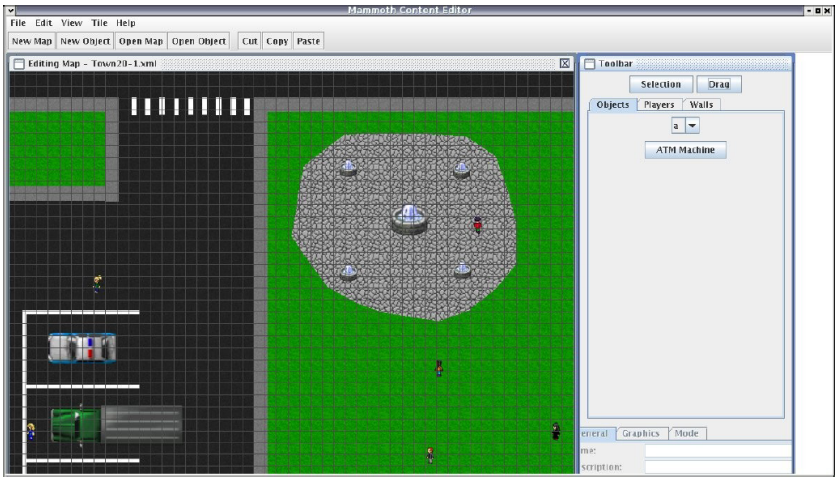


Figure 10.5: Content Editor in June 2007

centralized hub to route all messaging, and Toile, which provides a fully connected mesh topology where all participants are connected to each other.

Another important addition to the network was Postina, a network engine designed and implemented by Dominik Zindel. Using Pastry's peer-to-peer communication layer [RD01] and the Scribe publisher/subscriber interface [CDKR02], Postina was designed to provided a truly scalable network engine. However, the FreePastry library [DEG+04] used for the project had several problems with nodes disabling themselves when too many nodes were in the system. Though Postina did work for a small number of nodes, it was too unstable to provide reliable results in large-scale experiments. However, Postina's development was a great contribution to network engine development, as it helped shape the network interface for future peer-to-peer network engine.

During this period, Mammoth received an important facelift when Alexandre upgraded the graphic engine from the custom-built MinuetoGL to the community designed and supported JMonkey [Pow10]. Given that JMonkey had a large amount of community support. This change greatly reduced the amount of graphic programming needed by the Mammoth team. This upgrade also introduced some interesting pseudo-3D effects for walls and trees.

Wisam Alabed and Yifan Li started a project to allow GPS-enabled portable

devices to transfer location data to Mammoth, as a means to control a player. Using a rudimentary map of McGill, a student would be able to move a Mammoth player by walking around the campus. Most of the implementation was completed, except for the integration with the GPS unit, which was unavailable at that time.

In addition, Nicholas Rudzicz started his work on Arda, a flexible content generation plate-form. The goal was to provide a modular tool to generate content at different levels (terrain, city, street, building, etc.) and to allow these tools to interact with each other. The results could then be exported into Mammoth.

The last change this summer, integrated by Alexandre, was the first generation of the proxy generator, as described in chapter 6. For the first time since its development, Mammoth could run in a true transparent multi-server configuration.

10.7 Summer of Code 2008

Summer 2008 was particular busy, as numerous undergraduate and CEGEP students joined the team. One particular component that received numerous upgrade was the content editor. The first step was to separate the content editor project into several smaller components: the World Object Editor (Wote), the Map Editor (Mape) and the 3D preview/conversion tool (Preview). The simplest of these components was Wote, developed by Julien Dreux. It allowed the creation of object types within the object hierachy. These object types could then be loaded into Mape and added to the game world. Wote was quickly upgraded to Wote2 to include new object reflection features, allowing developers to modify the structure of game objects without having to constantly modify the content generation tool each time.

When Édouard Lanctot-Benoit returned to the project to build more 3D models of the campus, additional work was done on the Preview tool, allowing for easy testing and exporting of 3D objects in Mammoth. Edouard was assisted by another CEGEP student, Vincent Brillant-Marquis, who mostly focused on texturing the models.

Mape, the most elaborate of the content modules required a new complete overhaul. The weakness of the first iteration of the editor was that is was too different

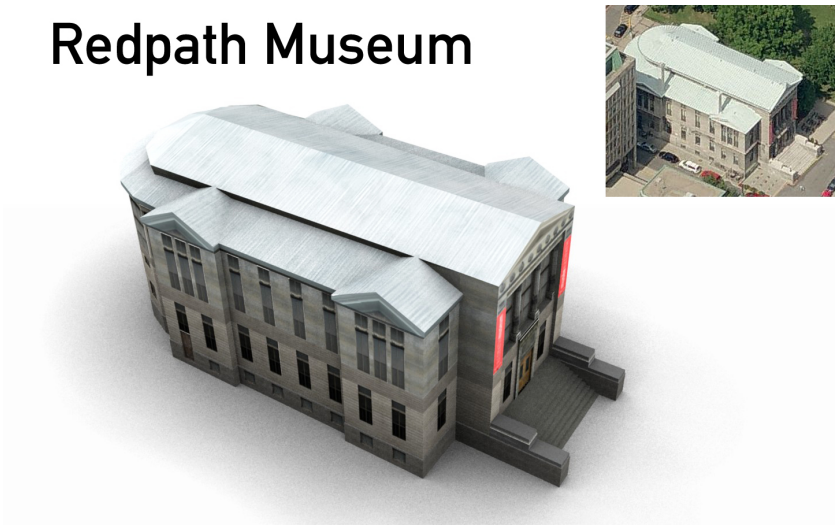


Figure 10.6: Render of Redpath Museum as drawn by Édouard

from the game, making it difficult to determine what the map would look like. The second iteration had the opposite problem; it was too closely coupled with the game and any changes made to the game graphics affected the map editor. Alexandre refactored all the graphic components into a distinct graphic package, separating it from the client's logic. The results of this refactoring can be found in 11.1.2. The second step was to separate the editor logic from the display. Tommy Sheng Liang, a science CEGEP student from Marianopolis College, was particularly helpful in improving Mape to make it more stable and user friendly.

An interesting project initiated by two undergraduate students, Ashton Anderson and Amy Goldenberg, was to integrate voice commands into Mammoth. The challenge was two-fold: integrate a speech-to-text library in Mammoth, and parse voice commands into Mammoth actions. Unfortunately, given the lack of maturity in available Java speech-to-text libraries, the voice interface never properly functioned. However, the voice parsing component was completed and allowed the command of players through natural commands in the chat box.

That summer, Yanwar Asrigo started working with Quazal's Rendez-Vous framework, exploring the possibility of integrating Rendez-Vous as the authentication and chatting infrastructure for Mammoth. Thus, Alexandre designed a generic interface

Figure 10.7: Mammoth Client in 2008

for all the non-game services in Mammoth (messaging, authentication, grouping, etc). At the same time, Alexandre created a simple implementation of these services using a stripped-down version of the RPC system found in the replication engine. Finally, Alexandre continued the work on the replication engine, enabling object migration between servers.

10.8 Fall 2008 and Winter 2009

By Fall 2008, there was growing interest in implementing a 3D interface for Mammoth. Although 3D shapes and objects could be inserted into a Mammoth game map, the 3D support was fairly limited. Three undergraduate students, Robert Rolnick, George Ciobanu and Scott Mcmurray, took on the task of implementing height map support in Mammoth, allowing a player to walk over 3D terrain. This proved to be an interesting challenge, not only in adding the 3D terrain, but allowing other components of the game, such as the player, to interact with this terrain (i.e. walk on it).

Riry Pheng started work on the PSense network layer. This project was particularly interesting as it introduced a new P2P technique for object discovery, completely different from Quazal's approach. As such, PSense also provided an alternative implementation to the replication engine. This was an important change, as it demonstrated that even the replication engine, one of the most important components in Mammoth, could be replaced.

The integration of the triangulation tools in Mape allowed for easier experiments with triangle partitions. Jonathan Pullano implemented a variation of triangulation A* [DB06] for Mammoth.

During that period, most of Alexandre's work focused on maintaining and refactoring Mammoth. At the same time, he implemented the infrastructure for load balancing.

10.9 Fall 2009 and Winter 2010

Summer 2009 was a very quiet time for Mammoth, as there was no undergraduate research project associated with Mammoth and no active developers working on the code-base. However, many projects were continued during the fall and winter period, such as Riry's PSense network engine, Kaiwen's work on persistence, and Alexandre's work on load balancing and fault tolerance.

In addition, Christopher Dragert started his work on formalizing actions in MMOGs and analyzing the sub-components of these actions. At the same time, Christopher refactored the action interface, which allows the control of player controlled and NPC characters. Theodora Dan joined the Mammoth team in the Fall, focusing on improving the user experience of Mammoth. Much of her work centered around upgrading the graphic engine to JMonkey 2.0 and improving the NPC players.

Chapter 11
The Architecture of Mammoth

The initial goal of Mammoth was to provide an implementation platform for academic research related to multiplayer and massively multiplayer games in the fields of distributed systems, fault tolerance, databases, networking and concurrency. During the last 5 years, several other projects have used Mammoth to conduct experiments in the fields of artificial intelligence, modelling and simulation, and content generation.

To allow researchers to easily conduct experiments, the Mammoth framework was designed as a collection of collaborating components that each provide a distinct set of services. The components interact with each other through two types of well-defined interfaces, engines and managers. The general architecture is depicted in figure 11.1. At the highest level, the Mammoth architecture follows the Model-View-Controller paradigm.

- The main components in the model are the World Engine, the Sub Games Manager, the Physics Engine and the PathFinding Manager.
- The main components in the view are the Persistence Manager, the Web Monitor, the Logging component, the XMLTools and various Mammoth clients, which also act as controllers.

Currently, Mammoth has a 3D client and an NPC client, which is a client without graphical user interface that executes AI algorithms which control the movements of a

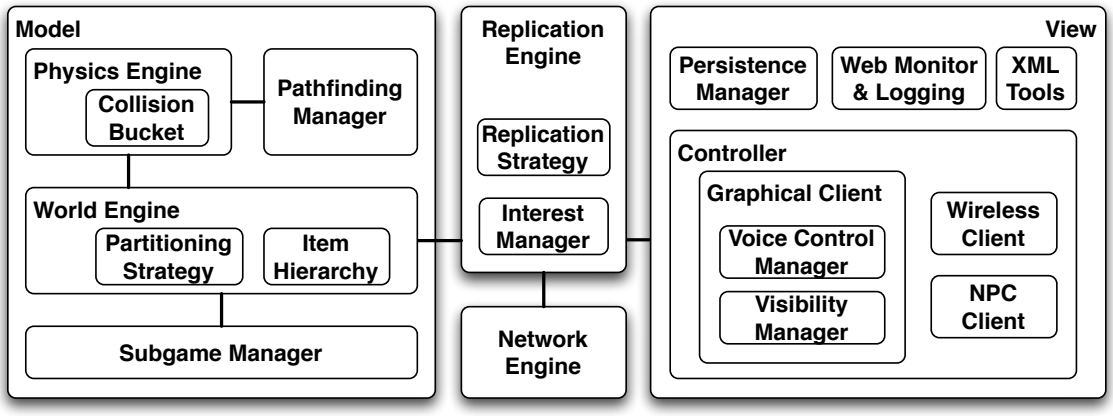


Figure 11.1: Components of the Mammoth Framework

player. The Model is connected to the Views and Controllers though the Replication Engine, which implements the run-time support for replicated objects. It contains the Interest Manager and interfaces with the Network Engine for low-level communication.

11.1 Engines

Engines are core components that can be completely replaced to experiment with alternative implementations. A classic example of this would be the multiple network engines available in Mammoth. The engines can be interchanged transparently, as long as they provide the required features determined by their interface.

11.1.1 World Engine

The World Engine stores all the components contained in the game world and provides an easy interface to retrieve these components. At first glance, the World Engine doesn't seem like a component that needs to be replaceable, since it is fundamentally a complex data structure for storing game objects. However, careful profiling reveals that a rather large percentage of CPU time (more than 20%) is spent in the World Engine searching for various game objects. Thus, optimizing the different data

structures used to store the game components in the World Engine is an interesting research problem.

The first implementation of the World Engine used one large hash table to store all the different game components. The current implementation has demonstrated an important increase in efficiency by storing the different game objects in several smaller separate hash tables.

World Object

The state of the game world is represented by a collection of *World Objects*. The objects are broken down into a hierarchy, as shown in figure 11.2. A *World Object* is either static or dynamic.

- *Static Objects* are immutable, they do not change or move over time. As such, they are always loaded locally and never transmitted over the network.
- *Dynamic Objects* are mutable objects that can change over time. They create most of the network load and are broken down themselves into two categories: *Items* and *Active objects*.

Although both *Items* and *Active Objects* are mutable, *Active Objects* are defined by their ability to change/move themselves or other objects. In programming terms, *Active Objects* have an update function, which is called at every iteration of the rendering loop. This enables the active object to execute actions that will alter itself or the world around it. *Players* are considered a special case of *Active Objects*.

11.1.2 Graphics Engine

The graphics engine displays the world to the player, and allows the player to visualize his status and interact with his environment (moving, manipulating objects, chatting to other players, etc.). The engine is built using a layered architecture, encapsulating the graphic programming and hiding it from the programmer. This is key, because

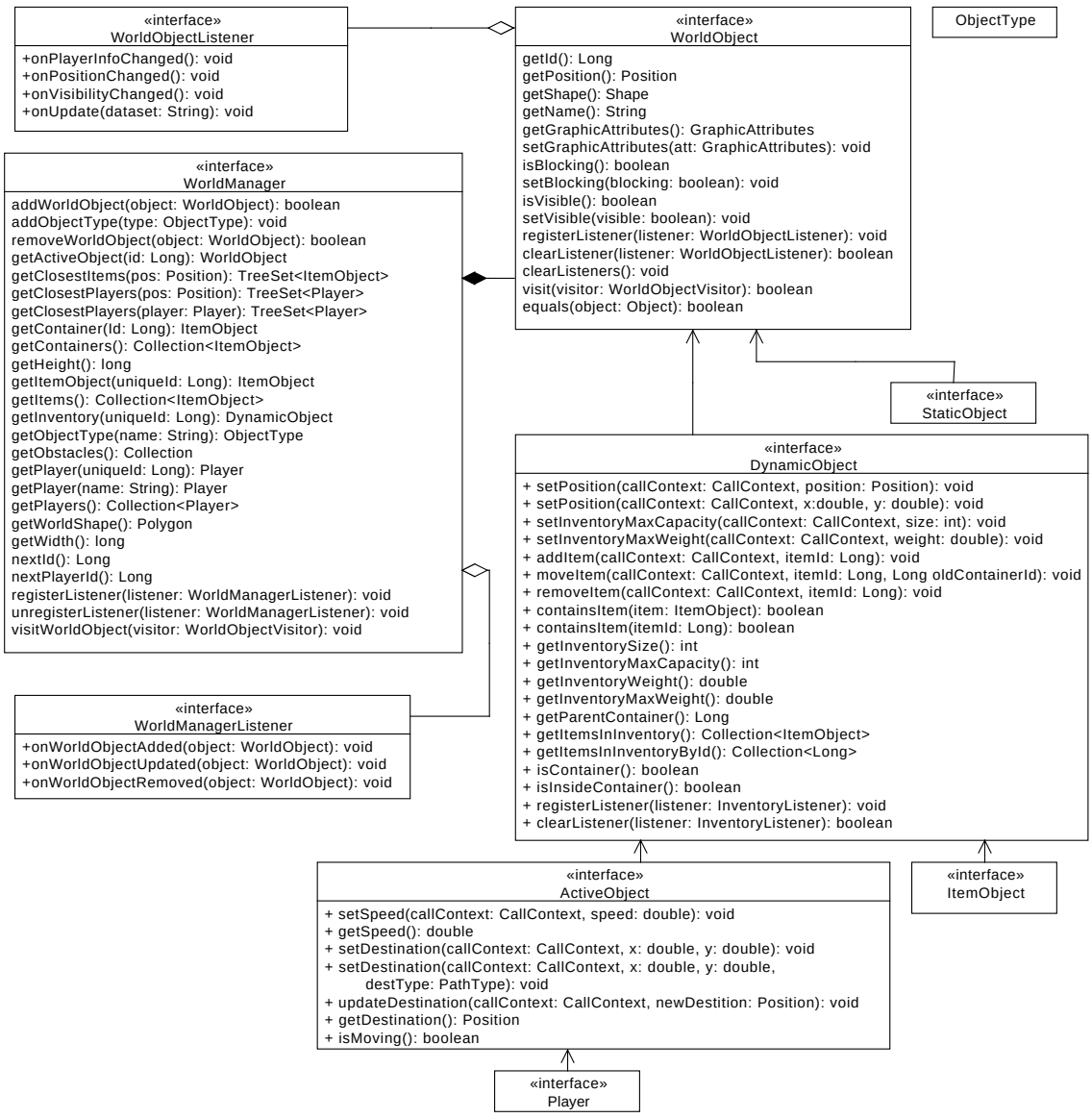


Figure 11.2: UML Diagram of the WorldEngine and WorldObject

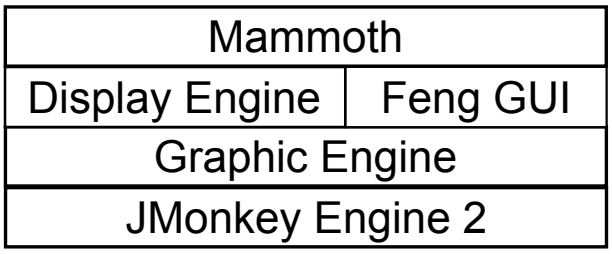


Figure 11.3: Different layers for the Graphic Engine

most developers on the Mammoth project have little or no graphic programming experience. The architecture of the graphic engine as illustrated in figure 11.3.

At its lowest layer, the engine uses JMonkey [Pow10] to render the scene. The *Graphic Engine* layer ties the upper and lower layers together, allowing a developer to switch the graphic renderer without having to touch the developer's interface. The *DisplayEngine* is the higher layer, which abstracts the graphic programming for the developers. Also used by the developers, FengGui [Sch09] provides UI elements for OpenGL, similar to Swing.

11.1.3 Physics Engine

The Physics Engine implements interactions between game objects. The current Physics Engine is very simple: it only implements basic collision detection. This particular component was subject to numerous optimization, as collision detection is very CPU intensive.

11.1.4 Replication Engine

The Replication Engine (see figure 11.4) is a component that enables the replication of objects on different nodes within a distributed system. The replication process is managed by the *Replication Space* component (see chapter 4). An alternate peer-to-peer implementation of *Replication Space*, based on the PSense [SSJ+08] technology and outside the scope of this work, can also be used.

The *Replication Strategy* acts as bridge between the Network Engine and the

Replication Engine so that different types of communication can be assigned to the Replication Engine transparently. This is especially useful with unit testing, where a dummy *Replication Strategy* is used.

11.1.5 Network Engine

The Network Engine component provides basic communication to Mammoth. In order to support the communication needs of replicated objects, the Network Engine provides the following means of asynchronous communication:

- Direct messaging.
- Global broadcasting capabilities.
- Publish/Subscribe-based broadcast capabilities.

The publish/subscribe component is key to the Network Engine, as it ensures proper propagation of updates in the replicated object infrastructure. Most often, the subscriptions are directly managed by the interest management system of the framework. The efficiency of the broadcast and publish capabilities depends on the topology (how the computers are connected to each other) of each engine.

Mammoth, currently, has multiple implementations of its Network Engine:

- Stern: the star topology Network Engine. A central hub routes the communication of all nodes connected to it. Messaging is done over TCP/IP, and is implemented using the Mina [Min10] socket framework.
- Toile: the fully connected engine (every node is connected to every other node). Toile also uses TCP/IP and Mina to manage its messaging.
- Postina: a self-organizing peer-to-peer Network Engine using tree-based broadcast. It is implemented using FreePasty [DEG+04].
- Fake: uses shared memory and emulated serialization to route messages across components. Fake is mainly used when executing unit tests on components that depend on a Network Engine.

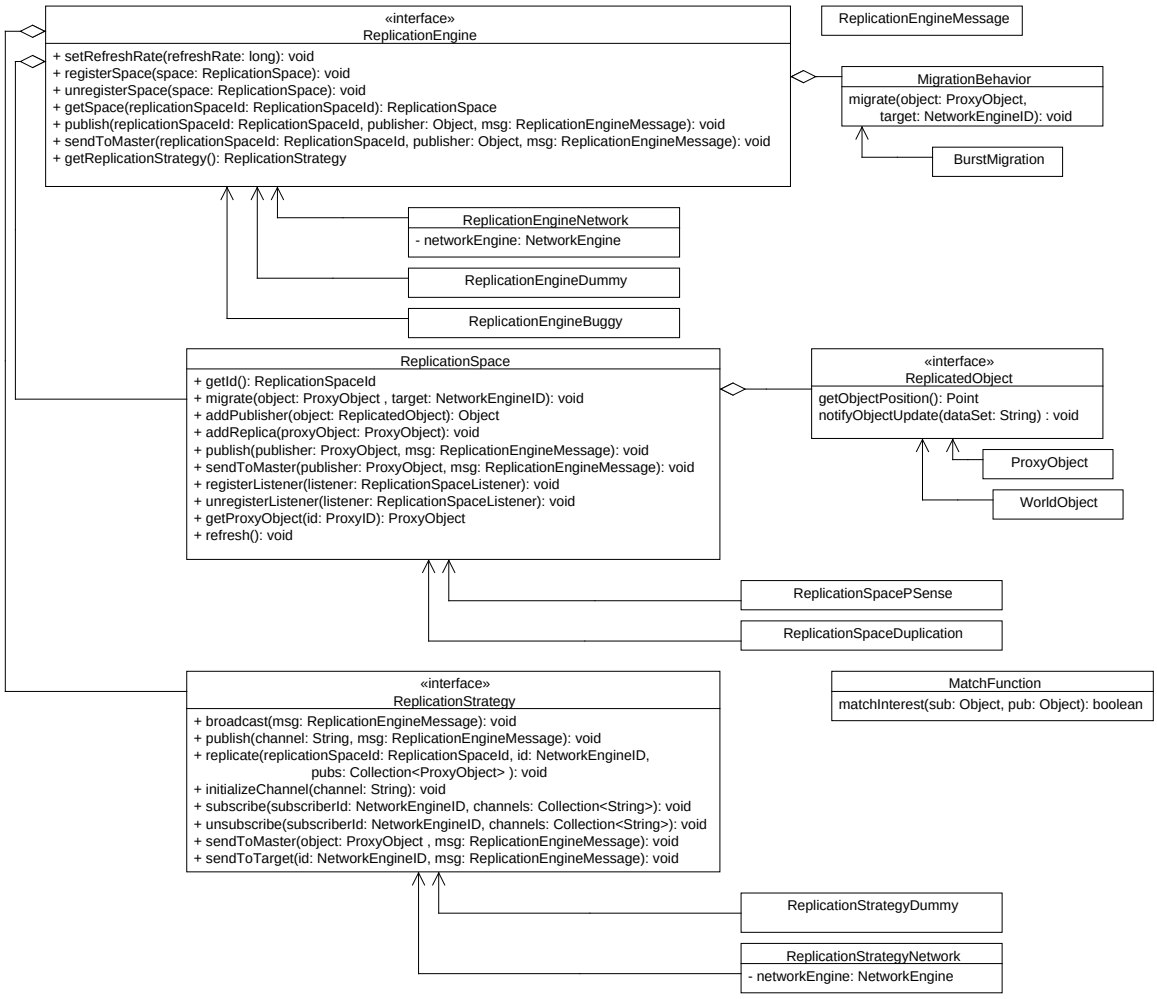


Figure 11.4: Mammoth's current Replication Engine

In addition, Mammoth has had several experimental network engines, some of them based on group communication system such as JGroups [ACL04] or Appia [Pin05]. The architecture of the network engine can be found in figure 11.8.

11.2 Managers

Managers are components designed to control multiple implementations of a given algorithm or strategy. Compared to engines, which allow a single implementation of a particular component, managers allow multiple implementations of a given functionality to be registered with the system. An example of this would be the Pathfinding Manager, which provides several different pathfinding algorithms. Different algorithms can then be assigned to different players, allowing for experimentation in a live setting.

11.2.1 Pathfinding Manager

The Pathfinding Manager (see figure 11.5) is responsible for managing the different registered pathfinding algorithms found in Mammoth. These algorithms are encapsulated in *PathFinder* objects. At least one pathfinding algorithm should be registered with the manager.

Pathfinding requests are executed asynchronously in separate threads. As the pathfinding algorithm is executing, it adds waypoints to the supplied *Path* object. The Pathfinding Manager allocates threads from the thread pool when a pathfinding requests are executed. Obsolete requests, most often because the object has changed destination, can be cancelled which allows the threads executing them to be reclaimed.

11.2.2 NPC Manager

The NPC (non-player character) Manager is the central administration unit for the behavioural AI components. In Mammoth, NPCs are controllers. Once spawned, each is assigned to a player and takes control of it. The NPC behaviour is encapsulated

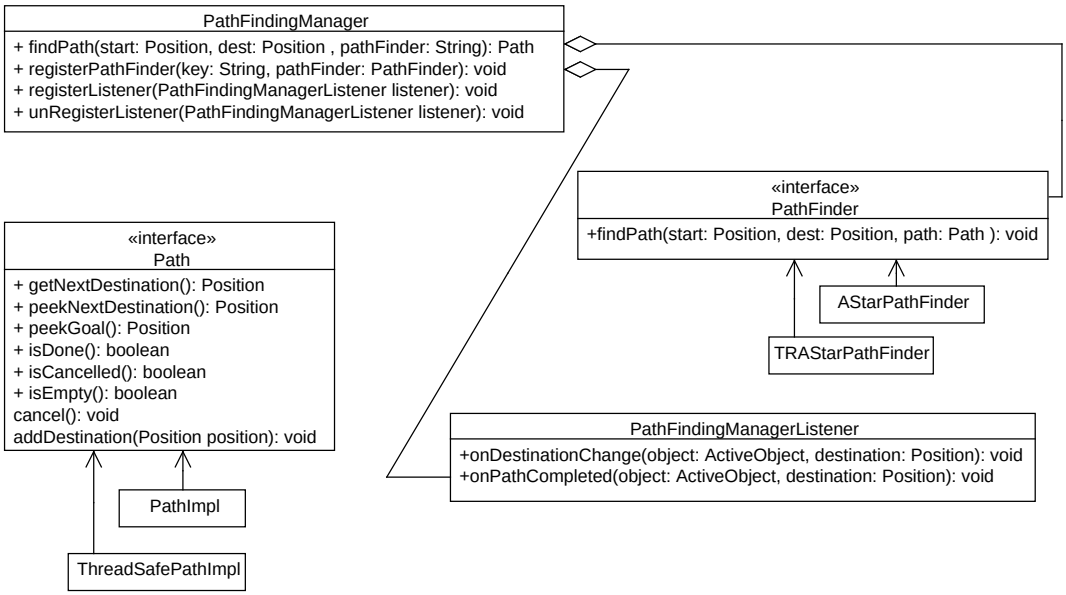


Figure 11.5: Mammoth's current PathFinding Manager

into a sub-role object, which can be broken down into several other Role objects. At every game timestep, all Roles registered with the NPC Manager are queried for a new action. The Roles that decide to take a new action return an NPC Action object, which is then queued inside the NPC Manager. The NPC Manager then shuffles the actions and resolves them in a random order. This prevents a Role from gaining an unfair advantage because it was registered first.

An NPC can observe the world through the *iSeeNewObject* and *iDontSeeObjectAnymore* methods. When an object enters a certain radius of a player, the *iSeeNewObject* method is triggered. Once that object leaves the visibility of the player, the *iDontSeeObjectAnymore* method is triggered in turn.

As mentioned, Roles can be composed of several other sub-roles. For example, the implementation of a seek and gather AI could be done using three Roles. The first Role, seek, navigates the character through the game world. The second Role, gather, analyzes the list of objects it knows about and collects some of them. The third one is the master Role, which switches between the behaviour of seeker and

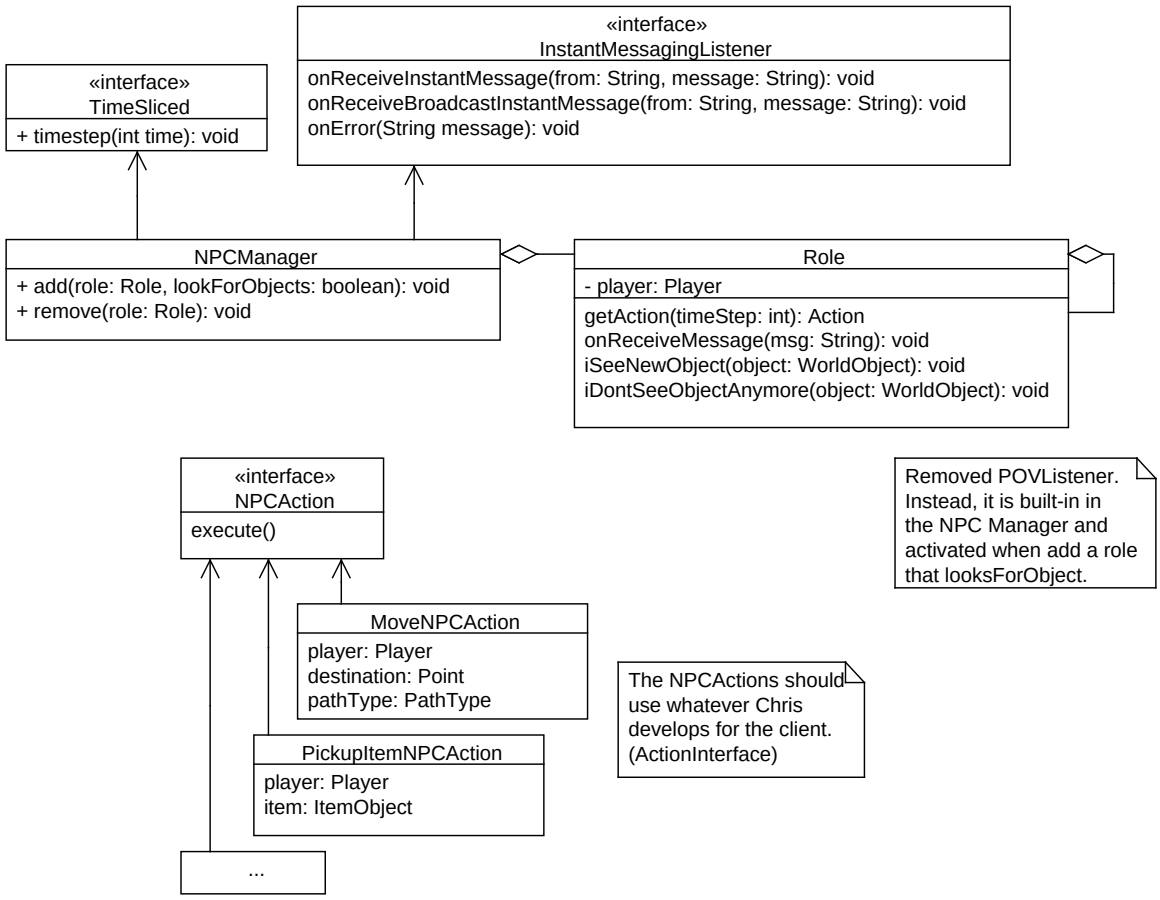


Figure 11.6: Mammoth's NPC Manager.

gather Roles. The master Role must be registered with the NPC Manager.

11.2.3 Persistence Manager

The Persistence Manager administers the different strategies used by Mammoth to save the state of the game to stable storage. Both push (automated recording of events) and pull (explicit retrieving of the state of game objects) strategies are supported. In addition, the Persistence Manager provides the necessary Data Access Objects (DAO) required to access stable storage, which is most often a relational database. Given that the persistence strategy implementations are kept separately

from the DAOs, it is trivial to experiment with different strategy / storage medium combinations.

11.3 Implementation

The implementation of Mammoth is done almost exclusively using the Java programming language. This was a practical decision. Many researchers at the School of Computer Science of McGill University use Java for their experiments, and many tools have been developed for research and performance analysis in Java. Furthermore, the cross platform nature of Java facilitates access to Mammoth for the students, and makes maintenance easier. Of course, an industrial implementation of our framework using a lower-level language such as C++ would provide even better performance. However, our experiments are still valid, since they provide insight into the complexity of our algorithms and techniques as the number of players, game objects and nodes increases.

In order to achieve flexibility and extensibility, many advanced programming techniques have been used in the development of Mammoth. Given the modularity requirements, many of the design patterns proposed in [GHJV95] are put to good use. This section outlines the importance of interfaces and listeners in the design of this modular architecture, and also describes how XML is used to successfully deal with changing data structures.

11.3.1 Interfaces

One of the key elements in the Mammoth architecture is the flexibility with which engines can be replaced, and new algorithms registered with the managers. In order to make this possible, engines and manager strategies define their own interfaces. All interactions between components in Mammoth are performed using interfaces, i.e. at the abstract level: no concrete implementation is ever directly referred to. This is very similar to the bridge design pattern [GHJV95], where abstractions are decoupled from their implementations. As a result, the implementation of a component can be

changed without the need to modify any of the depending components. However, the definition of the abstraction is fixed: changes in the interfaces themselves could require significant refactoring and should therefore be avoided. Fortunately, after over 5 years of development, the interfaces of the major components have become fairly stable.

To allow greater flexibility in the execution of Mammoth, object factories, and their respective configuration files, are used to control the instantiation of the different implementations. A researcher can simply specify the component to be used in the Mammoth configuration file before starting the game. Those factories read the researcher's choice from the configuration file and instantiate the appropriate engine or instruct the managers to use a specific strategy. In this manner, the researchers do not need not worry about the initialization details of a component. Some factories even use the Java reflection API to automatically recognize new available implementations of components. This allows the addition of new implementations without the need to modify existing factories.

11.3.2 Listeners

Modularity and separation of concerns is essential when developing a complex research framework. Strong dependencies between components greatly reduce the flexibility and maintainability of the source code. As a result, team development is complicated, since changes required to implement a specific feature within one module can have a major impact on other parts of the framework. Within Mammoth, for example, the Persistence Engine requires knowledge on how and when a World Object is modified. The World Engine could directly inform the Persistence Engine about the state update, but that would create a dependency. Changing the implementation of the Persistence Strategy might then again require the modification of the World Object. Such modifications could be risky, since the world engine is a central component of the Mammoth framework.

The Mammoth framework addresses this problem through the extensive use of

listeners. As described by the observer design pattern [GHJV95], core objects containing the game state are considered subjects. Components requiring information about a subject can register themselves as observers with the subject by implementing the appropriate listener interface. Whenever the state of a subject changes, all registered observers are notified of the change. The most notable example of the use of listeners within Mammoth is the graphical game client, which is designed as a view object registered with every World Object. However, listeners are also used in various other components, such as the Network Engine, the Replication Engine and the Pathfinding Manager.

11.3.3 XML

Constant change is one of the most difficult challenges when working on a research framework. Data structures are constantly updated to reflect new features. The perfect example of this is the Mammoth world map, whose format has changed countless times since the beginning of the project.

The problem is that creating a map is time consuming. In addition, many researchers have created custom maps for their specific research purpose. It is therefore impractical to recreate or manually convert all available maps each time the data structures changes.

In Mammoth, data is stored using XML. Objects are decomposed into attributes, the simpler attributes being stored as tags and more complicated earning themselves a dedicated subtag.

Whenever the format of a data structure changes, a new XML reader / interpreter is written for the new format. In addition, the readers of the older formats are updated to simply convert the old format to the new one. Thus, any data structure that is read from disk will be saved in the newer format, regardless of the format it was in originally. In addition, if the data structure is saved back to disk, the most recent XML format is used.

For example, the code samples 11.1 and 11.2 describe the same object, but saved using two different XML formats. Early in 2008, the notion of depth ordering, which

was used to determine the drawing order of objects when Mammoth was purely 2D, was converted to elevation. In addition, the code 11.1 is missing the "flat" type tag in the attributes section. This is understandable since at that time, all objects were flat.

If the new reader were to load the XML code from 2007, it would detect the missing type tag. Thus, the older reader would take over and fill in the missing information. Type would be defaulted to "flat" and elevation would be calculated as depth ordering divided by 10.

Listing 11.1: XML code for storing a table object January 2007

```
<worldobject blocking="true" id="1819" name="table"
    objecttype="Static" walkable="true" wx="22.28" wy="4.43">

  <attributes>
    <depthOrdering value="2"/>
    <invisiblewhenunder value="false"/>
    <forceDrawRect value="true"/>
    <textureKey value="table.png"/>
    <textureRepeat value="false"/>
  </attributes>

  <shape>
    <point x="-0.11" y="0.23"/>
    <point x="-0.11" y="-0.23"/>
    <point x="0.11" y="-0.23"/>
    <point x="0.11" y="0.23"/>
  </shape>

</worldobject>
```

Listing 11.2: XML code for storing a table object January 2009

```
<worldobject blocking="true" id="1819" name="table"
```

```
    objecttype="Static" walkable="true" wx="22.28" wy="4.43">

  <attributes type="flat">
    <elevation value="0.2"/>
    <invisiblewhenunder value="false"/>
    <forceDrawRect value="true"/>
    <textureKey value="table.png"/>
    <textureRepeat value="false"/>
  </attributes>

  <shape>
    <point x="-0.11" y="0.23"/>
    <point x="-0.11" y="-0.23"/>
    <point x="0.11" y="-0.23"/>
    <point x="0.11" y="0.23"/>
  </shape>

</worldobject>
```

11.4 Services

Most massively multiplayer games include non-gameplay related features, such as authentication, chatting, guild-like organization, and so on. In Mammoth, these features are described as services. Each of the services is defined as an interface (see section 11.3.1) and then implemented in either a centralized or a distributed way. Currently, Mammoth has interfaces defined for the following services:

- Authentification
- Instant Messaging (chat)
- Player Distribution (character assignment)

Each of these services has two implementations: one using a centralized RPC infrastructure and a dummy version used for unit testing. In addition, plans are in progress to implement these services using Quazal's Net-Z infrastructure. Other services, such as friend management, are also in development.

11.5 Evolving Architecture Example

Mammoth's architecture has been designed and redesigned several times since its conception in 2005, and it will continue to evolve as the project advances. These changes are best illustrated with an example, comparing the initial network architecture from 2005 (see figure 11.7) to the current network architecture (see figure 11.8).

One key difference is the distinct client/server architecture of the original interface. Not only does the architecture force all network engines to have a client/server topology, but it also strictly defines which role the client and the server is allowed to have. In addition, the original architecture defines network identifiers as integers. Another weakness is that several methods use the wrong parameter types. For example, the method for sending a direct message required a *Serializable* object, but latter type-casted it to a *Message* object. Mammoth's implementation had evolved faster and its API, forcing developers to make some horrible development decision.

The newer architecture is designed for flexibility, using all the strategies defined previously. The interface contains only the basic functionality that must be found in all network engine. Methods were refactored with the proper arguments, and network identifiers were defined as separate objets, allowing programmers more freedom into what can be stored in the identifier object. Postina, for example, uses the FreePastry string identifier to distinguish and address clients.

The current architecture was designed in 2007 and has been barely modified since. At least 6 separate network engines (Stern, Toile, Fake, Postina, Appia, PSense, etc.) have been developed using this interface. This illustrates how the flexible design techniques used in Mammoth development have allowed researchers to experiment with Mammoth for the last 5 years with minimal implementation effort.

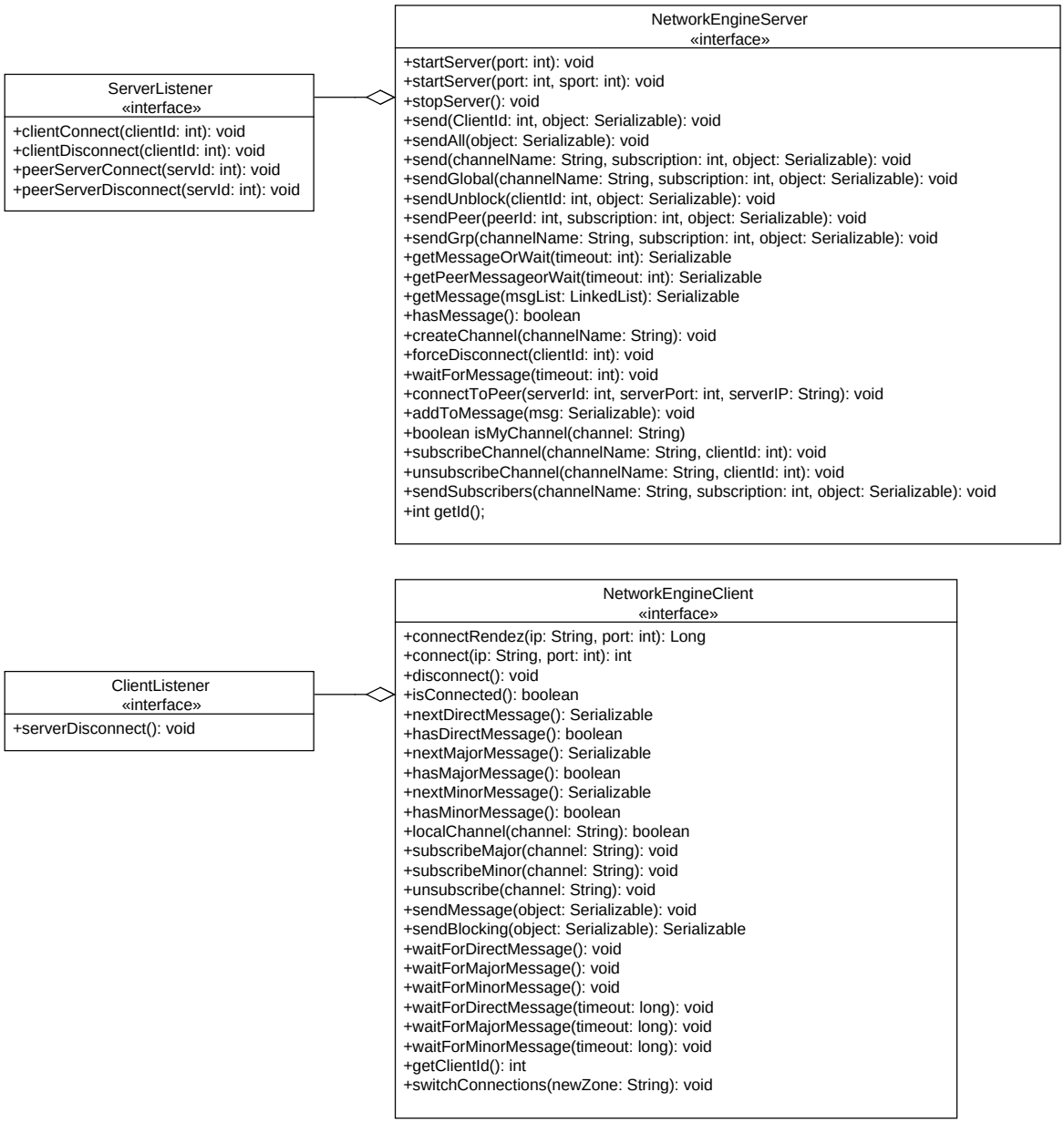


Figure 11.7: Mammoth's initial network architecture

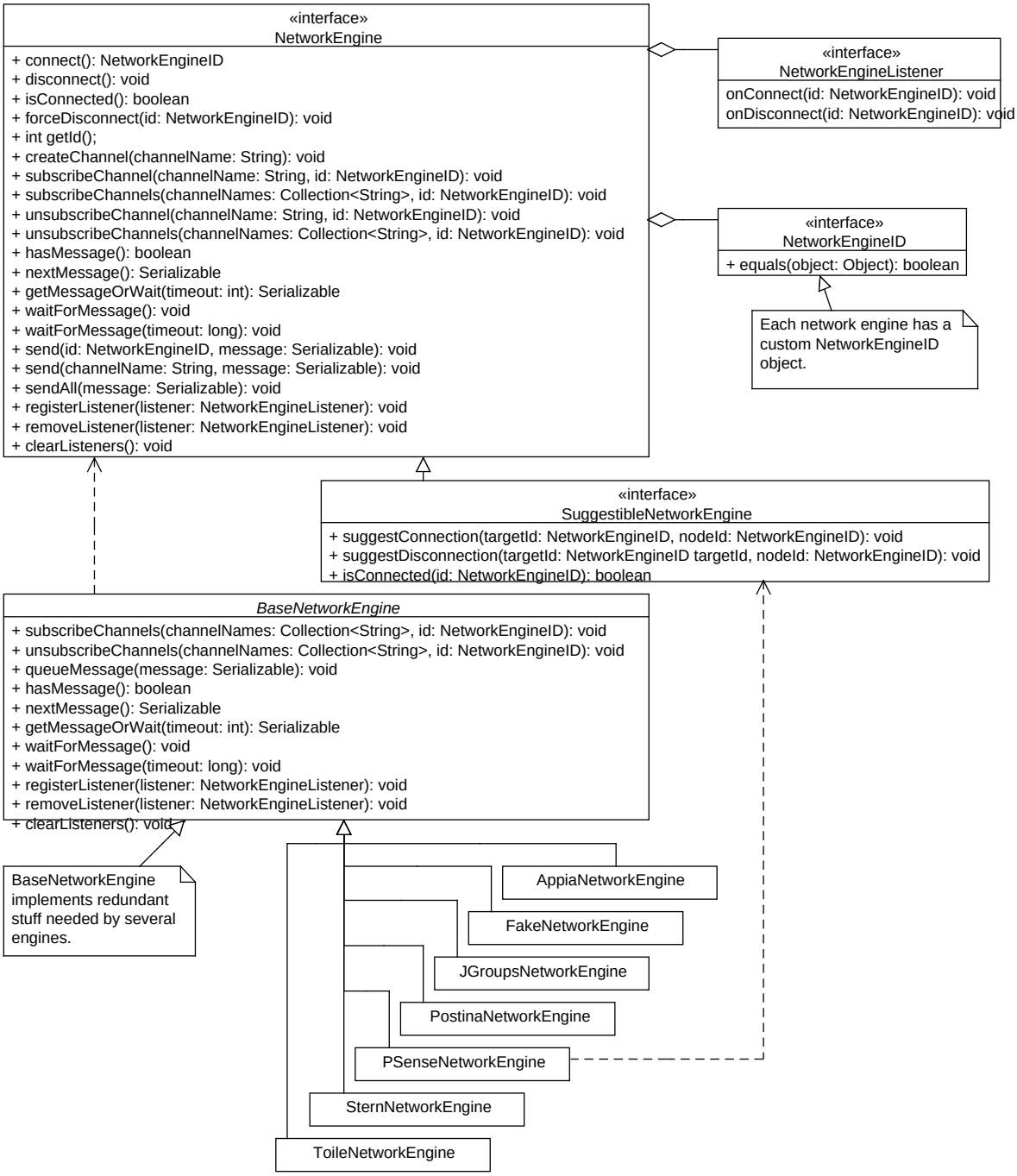


Figure 11.8: Mammoth's current network architecture

Chapter 12
Implementing Journey in Mammoth

Implementing the unified Journey approach inside Mammoth required the addition of four key functionalities: trust, load balancing, fault tolerance and auditing. Fortunately, several building blocks, such as replicated objects (see chapter 4), proxy-based RPC (see chapter 6), and services (see section 11.4) were already implemented and simplified the integration process.

This chapter describes the integration process of these four components into Mammoth.

12.1 Implementing Trust

The central component to Journey is Trust, as it is used by all the other components. Trust is also the one most difficult to implement, because it should, ideally, have perfect reliability and security. Fortunately, evaluating the potential and performance of Journey does not require a perfect trust component. Thus, the initial implementation of the trust component uses a simple RPC infrastructure.

Trust is currently implemented as three static levels in Mammoth: Trusted for Cells (level 2), Trusted for Masters (level 1) and Untrusted (level 0). When a node joins the system, it must register itself as a Trusted for Cells or Trusted for Masters node. Otherwise, it is considered Untrusted. In addition, the trust system stores two additional values for every node, a load value and an overloaded flag. Nodes are

responsible for updating their own values at regular intervals. The system can easily be expanded to hold information about past faults and cheats, useful information for upgrading and/or downgrading levels.

Clients are allowed to register themselves with the system, update their properties, get the trust level/load of the other nodes in the system and signal node failures. Currently, a single node failure signal is enough to disqualify a node and remove it from the trust system. This is allowed for simplicity sakes, although this power should be reserved for level 2 nodes. The current implementation of trust uses a pull architecture, where trust information must be retrieved from a centralized location.

12.2 Implementing Load Balancing

Adding load balancing functionalities to Mammoth required implementing three features: object migrations, load calculations and a rule system for the load balancing itself.

12.2.1 Migration

In Mammoth, migration is defined as the ability to move the master copy of an object from one node to another. This ability is critical in load balancing, as spreading the load across nodes requires moving master objects from an overloaded node to a different node. Migration is implemented using the traditional interface system in Mammoth, allowing for the implementation of multiple migration algorithms.

Burst migration is the first migration algorithm implemented in Mammoth (see section 8.3.1). The name originates from the fact that objects are migrated without warning to the destination node. In other words, a master object is transferred from one node to another and the destination node cannot refuse the migration. There is very little reliability in this migration algorithm, because there is no checks to ensure that the object was successfully migrated. However, when executed over a reliable communication channel (e.g. TCP network), this has not been a problem.

Burst migration is the fastest migration that can be implemented in Mammoth:

it uses only one message passing round (the sending itself). However, to satisfy reliability concerns, a fault-tolerant version of burst migration was also developed. Its description can be found in section 9.3.4.

12.2.2 Load Calculation

In Mammoth, the load model allows the system to calculate the amount of load found on a node. The current implementation is fairly straightforward. A different integer weight is assigned to each element (items, players and cells) in the load model. The total load on a node is the sum of the number of each type of elements multiplied by their weight. More information on the load model used in Journey can be found in section 8.1.3. The weights currently used in Mammoth were set experimentally.

12.2.3 Rules for Load Balancing

In Mammoth, load balancing is implemented as a rule-based system. A single threshold is used to detect if a node is overloaded, as described in section 8.3. Load balancing rules will compare this threshold value to the current load to determine if any action should be taken.

Initially, rules were designed to be independent of each other. This greatly simplified the design of rules, because no knowledge of other existing rules was required. However, early experiments demonstrated that rules had a tendency to contradict each other: rules would migrating objects back and forth between the same two nodes. Thus, the rules were re-written to take into accounts all other existing rules.

The following load balancing rules can be found in Mammoth:

- Rule 1: KeepLoadUnderThreshold: if the threshold is reached, it randomly migrates master objects to other level 2 nodes. This is the first rule that was implemented but is now obsolete, because a combination of rule 5 and 3 or rule 5 and 4 achieves similar results.

- Rule 2: SpillObjectsToTrusted: If level 1 nodes are present in the system, this rule slowly transfers a few master objects to level 1 nodes. Given that these

nodes are not completely trusted, only a few master objects are stored on any given level 1 node.

- Rule 3: RandomTransferTileUnderThreshold: This rule was implemented to test the growing/shrinking functionalities of cells. If a node is above threshold, it transfers a small number of tiles from its cell to another cell.

- Rule 4: AdjacentTransferTileUnderThreshold: Similar to rule 3, this rule transfers adjacent tiles to the destination cell. This allows for a more coherent cell division, limiting the migration and registration cost of objects traveling from one cell to another. The algorithm for choosing which tile to transfer can be found in section 8.3.3.

- Rule 5: MasterObjectNotOnTheirMasterCell : This rule migrates master objects if they are located on the same node as their corresponding master cell. Contrary to the other rules, the purpose of this rule is to ensure that the fault tolerance algorithms can be properly executed (see section 9.3.1). However, when combined with rules 3 or 4, this rule will usually balance objects adequately across level 2 nodes.

The experiments presented in chapter 13 use rules 2, 4 and 5.

12.3 Implementing Fault Tolerance

Fault tolerance can be achieved in two steps, detecting faults and dealing with them. To this end, Mammoth defines two interfaces, one for fault detectors and one for fault handlers.

12.3.1 Fault Detector

As the name implies, fault detectors detect and identify faulty nodes. Fault detectors in Mammoth are currently designed to detect crashed nodes: either nodes no longer connected to the system or no longer responsive. Once such a node is detected, the

system will iterate over all duplicate object to find which, if any, are hosted on the failed node. These objects are then tagged as orphaned and must be recovered by a fault handler.

Each node has its own fault detector and is responsible for detecting faulty nodes itself. A more centralized system was considered, but was found too vulnerable to cheating. The current fault detector is registered with the network engine to detect nodes that have disconnect from the system.

12.3.2 Fault Handlers

Fault handlers are designed to recover different types of objects. Two fault handlers are needed, one for cells and one for objects. Their implementation is rather straightforward and their behaviour is described in sections 9.3.2 and 9.3.1.

12.4 Implementing Auditing

Developing the auditing system required enhancements to several existing features in Mammoth.

The first enhancement was to allow execution of RPC calls on multiple nodes. This allows an RPC call to be executed on both the master object and the auditor. However, the auditor must execute this RPC call in a sandbox environment, so that it did not modify any objects that might be used by the client. This is not an issue when auditing method calls such as setDestination, because only the target object is modified. Auditing multi-object actions, such as picking up objects is more difficult, as the sandbox must contain both the target object and any other objects which might also be modified. Handling multi-object actions is outside the scope of this work.

Audit requests are asynchronous, there is no way to control the order audit requests are received in. This in turn, allows audits on past states of an object. This modification was simple, given the flexible architecture of proxy objects (see section 6.3.2).The states are serialized and stored in byte format in a first-in last-out (FILO)

queue. Only a fixed number of states are stored in the proxy, older states are discarded as new ones are recorded. The history feature can easily be enabled or disabled on a per object basis, limiting the recording overhead to only the objects that require auditing.

The biggest challenge in implementing the auditing system was deciding how auditing requests should be forwarded and processed. This is heavily discussed in section 9.4.3. The current auditing system will detect calls that are incorrectly processed, or not processed at all. Using the decision algorithms discussed in section 9.4.5, it will accuse a node of cheating if a certain number of faults is detected.

Chapter 13
Experiments

The effectiveness of the proposed unified solutions was evaluated using Mammoth, the massively multiplayer research framework. Situations similar to real-world scenarios can be generated by creating hundreds of artificial players and having them wander over the game map. These movements trigger different load balancing algorithms, allowing the measurement of their effectiveness. In addition, the other components of the system are tested by having clients either crash or cheat.

The effectiveness of the proposed system is evaluated by running Mammoth in several different configurations, enabling or disabling the proposed components, as to better evaluate their individual and unified performance.

13.1 Validation Techniques

Accurately conducting experiments measuring the performance of an MMOG framework would require hundreds of human players. This is, of course, impractical, especially if repeated experiments are to be conducted. The use of human players can introduce bias in the experimental data, especially if several experiments are conducted, given that it is impossible to get players to play a game in a consistent fashion over a long period of time. As players learn the game, their skill level increases, changing their style of play and biasing the results.

If using human players is impractical, both for logistic and consistency reasons, how can the framework be validated? Many papers present experiments with a small number of players and then scale up their results to illustrate how the system would cope with a large number of players. Another strategy is to simply simulate thousands of players using a simulator. Although these experiments can quickly determine metrics for several thousands of players, the fidelity of the implementation of the simulation can affect the validity of the results [DK10]. A third option has recently gained popularity in this research domain: using AI players to simulate player activity and produce an accurate activity load [RK07, Mat03]. Automated player behaviour, also called *Artificial Intelligence* (AI) in computer games, is usually programmed using some sort of scripts [OCS+05], or even modelled using a graphical modelling notation [FH02, Unr07, KDV07].

13.1.1 Role of AI in Games

It should be noted that the term "AI", when used in a video game context, is vastly different from the Computer Science term. When used in a scientific context, the term "AI" is used to denoted a learning system that takes activity data and tries to improve itself through experimentation and evaluation. In the video game domain, the term "AI" is used to describe any component of the game that is controlled by some form of automated logic. Most "AIs" are scripted, their behaviour being described in a particular scripting language. Using a scripting language to control "AI" is purely a practical decision, it allows for developers and game designers to tweak the behaviour of components without having to recompile the game. Although most game engines use their custom scripting solutions, current popular non-engine specific scripting languages for games are LUA [Lua10] and Javascript [Moz10].

13.1.2 AI for Testing

By studying how people play a game, it is possible to determine patterns in their game play. These patterns can be used to generate artificial players. These techniques have been successfully used in commercial games to create computer opponents for players

in various types of game. In First-Person-Shooter (FPS) games, these artificial players are called Bots. The use of Bots for testing is very common, both in commercial games and academic research. However, their use in load testing a system is fairly new.

The advantage with bot players is that their play style is consistent. Two rounds of experiments with similar AI player configurations should yield similar results. In addition, creating an experiment with several thousand participants only requires a large number of computers. The activation and deactivation of such players can easily be automated (scripted). Finally, as long as the behaviour of the bots is representative of player behaviour, the experiment can be considered reasonably realistic. Bots can also be used transparently in an experiment, given that the node hosting the game does not know the difference between PCs and NPCs. This is demonstrated in [LB06] where Quake 3 bots are used to analyze the effect of local lag and time warp on gameplay.

13.2 Experimental Setup

Great care and planning must be used when using AI players for load testing. [DK10] describe various configurations used to run the experiments in Mammoth, all achieved by switching the network engine and by running the NPC agents in different processes. The conclusion of this paper showed that CPU load, memory and bandwidth usage showed non-negligible performance variations, demonstrating the important influence of the simulation setup on the performance results. Following the suggestions of [DK10], this work uses a configuration where NPC agents are executed on seperate computers, as this is the most realistic experimentation setting: it is in fact just like an MMOG environment where each player runs the game client on his own machine.

13.2.1 Player Behaviour

Two different types of NPC control algorithms were used to control the behaviour of the avatars. The first, *RandomWanderer*, instructs the characters to wander randomly in strait lines, changing direction approximatively every few seconds. Although

the choice of direction is also random, previous experiments with approximately 40 students [BKV06] have shown that this type of computer-controlled players generate similar network load (message passing) to real human players. From a computational point of view, instructing an NPC to move randomly is easy for a computer. As such, this type of NPC generates very little CPU load on the node that is hosting the master of the player object, and can therefore not be used when stress testing components.

The second NPC control algorithm, *Waypoint*, directs characters from waypoints to other waypoints, using path-finding algorithms to avoid obstacles. This idea originated from the Orbius experiments [Lan05], where a group of 30 players were separated into 6 teams and asked to gather orbs of different sizes. The first team that collected four orbs of different sizes won. However, players from an opposing team could steal orbs from other players. A careful analysis of movement data in Orbius showed a pattern in the movement data: players would have a tendency to move along logical paths (streets, sidewalks, etc) and would gather at specific points, most often the intersection of the logical paths. The waypoint AI reproduces this behaviour: it is configured with specific points in the world that are called waypoints. When it takes control, it steers the player from one waypoint to the other. This behaviour generates a much heavier load on the nodes hosting the player masters, as extensive pathfinding is used to navigate between waypoints. The capacity of a server node is about 25 players, when those players are actively pathfinding between nodes. It should be noted that the pathfinding algorithm is purposely not optimized, as to create the load necessary to replicate a real game environment.

13.2.2 Faulty Behaviour

To properly evaluate the performance of the fault tolerance and auditing component, faulty and cheating behaviour are simulated during the experiments. Three types of behaviour are simulated: crashing clients (level 1 nodes), crashing servers (level 2 nodes) and cheating clients (level 1 nodes). Simulating faults in level 0 nodes does not make sense, as they have no responsibility in the system; their failure has no adverse

effect on the system. Cheating servers (level 2 nodes) are not used, since level 2 nodes are defined as nodes with absolute trust.

Crashing nodes (both level 1 and level 2) are used to test the fault tolerance component. These are nodes that disconnect from the system without warning. Their responsibilities must then be redistributed to the remaining nodes in the system.

Cheating nodes (level 1 only) are used to test the auditing component. The current cheat detection algorithm is designed to detect state inconsistencies between replicas and the auditing replicas. When a cheating player is present, these inconsistencies are caused by master objects incorrectly executing RPC calls. For the purpose of these experiments, a cheating node is defined as a node that either fails to execute a certain percentage of RPC calls, or that returns incorrect results. A specific number of cheating clients are introduced in the system, as specified by the experiment.

13.2.3 Network Model

The Stern network engine was used for all the experiments presented in this chapter. A single high-powered machine was used as the centralized network hub for all message passing between clients. Although a centralized approach limits the scalability of the system, it provides a central point to measure the network bandwidth circulating through the system. In addition, a stress test indicated that the single hub could easily handle 180 client connections, and provided a maximum throughput of 101 mBits/sec. Given that most experiments limit themselves to 100 clients and require a maximum of 50 mBits/sec, the Stern network engine is more than sufficient for the experiments found in this section.

13.2.4 Physical and Logical Setup

Three powerful servers were extensively used in the experiments: Halo, Oni and Rogue. Equipped with Quad-Xeon processors and over 8Gb of main memory, these machines were used to host bottleneck components of the experiments. Oni was used as the network hub, routing all network traffic through its network interface. Halo was used to host the first server (level 2) in all the experiments. Finally, Rogue, hosted

the non-gameplay services and ran the scripts to launch other clients. All clients and secondary servers were hosted on lab machines, with hardware configuration ranging from Pentium 4 to Core Duo processors. On average, about 85 of these machines were available, although some of the experiments used as many as 110 machines.

In some load related experiments, the number of clients used is capped at 100 or 150 for practical reasons: configurations unable to deal with a high load terminate anyway with a much smaller number clients. When experimenting with configurations supporting a high number of clients, the behaviour when exceeding 100 clients was shown to be predictable and followed the already established curved. Some initial experimentation was done with as many as 250 clients, but provided little more insight into the behaviour of the system. Other experiments use as little as 50 clients, as the purpose of the experiment is not to determine scalability, but to measure the performance impact of the evaluated component.

All experiments are done using the Town20-2 map (see figure 5.12). Given its relatively small surface area, the experiments using 100 clients reproduce a heavy load environment similar to many commercials MMOGs.

13.3 Experiments

The three main components of the unified approach are first analyzed separately, and then in a unified fashion.

13.3.1 Load Testing

The framework's ability to cope with load is first evaluated using nine scenarios, as defined in table 13.1. In all scenarios, the servers are first started, and then players join the game successively. A new AI player connects to the system every 29 seconds, and announces its interest to receive the necessary replicas. Four measurements are taken during the experiments: RPC time on clients (i.e. how long it takes for a client to execute a remote method on a server and obtain a result), CPU usage on the first server Halo, load levels on Halo and the total network throughput. Experiments

Scenarios	Number of Servers	Types of Cells	Type of AIs	Are master objects migrated to trusted clients when available?
Single Server - Wanderer AI	1	None	Wanderer	No
Single Server - Waypoint AI	1	None	Waypoint	No
Single Server - Trusted Clients - Waypoint AI	1	None	Waypoint	Yes
Dual Server - Static - Waypoint AI	2	Static Rectangles	Waypoint	No
Quad Server - Static - Waypoint AI	4	Static Rectangles	Waypoint	No
Dual Server - Dynamic - Waypoint AI	2	Dynamic Triangles	Waypoint	No
Quad Server - Dynamic - Waypoint AI	4	Dynamic Triangles	Waypoint	No
Dual Server - Dynamic - Trusted Clients - Waypoint AI	2	Dynamic Triangles	Waypoint	Yes
Quad Server - Dynamic - Trusted Clients - Waypoint AI	4	Dynamic Triangles	Waypoint	Yes

Table 13.1: Scenarios used for Load Balancing Experiments.

that use dynamic load balancing use a triangular cell tiling, and resize their cells to equally distribute load across trustworthy nodes. Static configurations rely on rectangular partitions, which remain unchanged, regardless of the load distribution. Since fault tolerance is not required in these experiments, master objects are hosted on their corresponding cell master.

The four observed measurements for the nine experiments are graphically represented in figures 13.1, 13.2, 13.3 and 13.4 respectively. Each figure is presented as three sub-figures, separating the single, dual and quad servers experiments. In addition, two experiments demonstrating Journey's ability to cope with flocking situations are discussed at the end of the section.

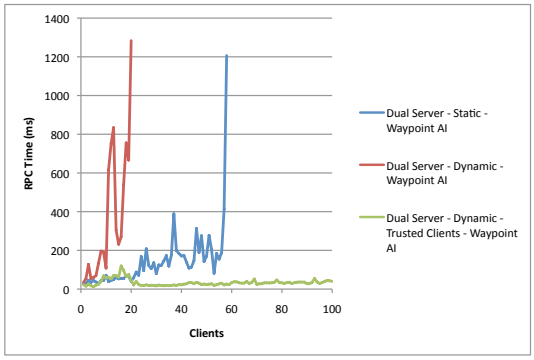
RPC Time

Looking at figure 13.1, we can estimate the maximum client capacity of each scenario. The maximum load of a particular configuration is determined by its ability to handle RPC calls. Once the time to execute a RPC call goes beyond 500 ms, we can consider the system as unresponsive. Given that remote calls will continuously accumulate at the node, few nodes recover from response times above the 500 ms threshold.

The single server Wanderer scenario (see figure 13.1(a)) features near-instantaneous RPC calls, a side-effect of random walking actions requiring no CPU power and thus generating almost no delay. However, the Waypoint AI, when used with a single server, generates a tremendous amount of load and greatly limits scalability, since all the pathfinding is executed on the server which holds the master objects of the players. This is shown by the exponentially growing red curve. If the player objects are moved to trusted clients, the system becomes much more scalable again (green curve). The pathfinding load is distributed to the clients, and the server only needs to process state updates to the player replicas and perform interest management. All of the measured RPC times are below 150ms, which is an acceptable delay for a MMOG, as shown by the green curve.

The dual and quad servers experiments (see figure 13.1(b) and 13.1(c)) offer insight into scalability by adding more servers to the system. These servers will share the

(a) Single server environments

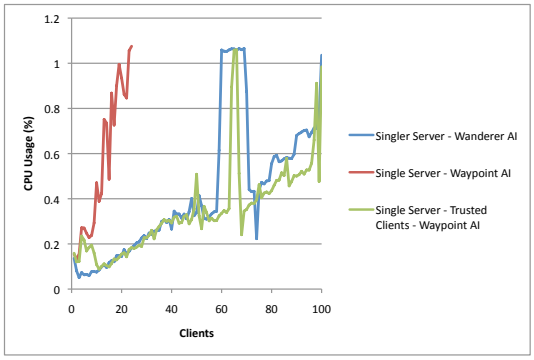


(b) Dual servers environments

(c) Quad servers environments

Figure 13.1: RPC Time for Clients in Load Balanced Environment.

workload created by player objects, increasing the scalability as long as the workload is properly divided among the servers. Given this fact, server configurations featuring dynamic cells (red curves) should outperform server configurations with static cells (blue curves), because they are best suited to adapt mutable workloads. However, the servers from dynamic cells configurations not only perform pathfinding calculations, but must also take care of load balancing. This seriously impacts their ability to scale up player capacity. Fortunately, if the pathfinding workload is assigned to trusted level 1 nodes (green curve), the RPC time for clients drop to minimal level, even when compared to the corresponding single server scenario (green curve in figure 13.1(a)).



(a) Single server environments

(b) Dual servers environments

(c) Quad servers environments

Figure 13.2: CPU Usage on First Server (Halo) in Load Balanced Environment.

CPU Usage

The graphs for CPU usage, shown in figure 13.2, were created by measuring the CPU usage on the server Halo. In all three single server experiments (see figure 13.2(a)), load increases linearly as the number of players increases. The CPU measurements are in accordance with the RPC time measurements. In the single server waypoint AI case (red curve), the CPU overloads at around 20 players, which explains why RPC time increases drastically. The CPU usage for both other single server scenarios increases linearly as the interest management complexifies with the increased number of players. Both of these scenarios see a dramatic CPU spike at around 60 players, which corresponds to approximately 30 minutes into the experiment. This CPU spike can be safely ignored as the JVM on Halo has a tendency to spike at different

moments in the execution, most likely an artifact of garbage collection. In all single server configurations, overload occurs when usage reaches 1 and the CPU handling the RPC calls is saturated.

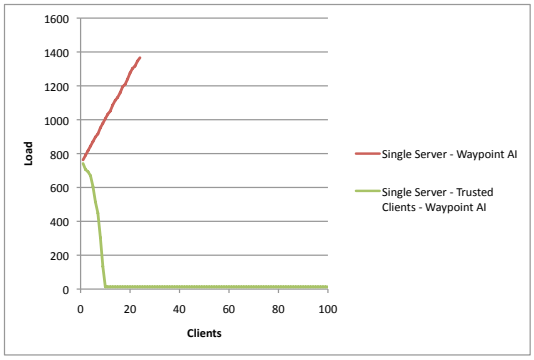
The CPU increase found in the multi-servers configurations (see figure 13.2(b) and 13.2(c)) using static cells (blue curves) is linear, similar to the single server experiments. However, the slope of the curve is noticeably lower. This is because the workload generated by the player objects is shared among the multiple servers.

The server, Halo, is equipped with 4 CPUs (quad-core). As such, when more than one CPU is active, usage can increase beyond 1. This is immediately noticeable in the multi-server experiments featuring dynamic cells (red and green curves). The load balancing algorithm used by these configuration is very CPU intensive and extensively uses a single core. These experimental configuration will usually overload when the CPU reaches 1.7: both cores are saturated and the server is unable to handle more RPC calls. However, the configurations using trusted clients (green curves) are much more scalable than their untrusted counterpart (red curves). Given the workload generated by the pathfinding requests is assigned to the clients, the servers are free to concentrate on load balancing and interest management.

Load

Figure 13.3 shows the calculated logical load of Halo during the experiments. By comparing these measurements to CPU usage and RPC time, the accuracy of the load model, as established in section 8.1.3, can be verified. The results from the wanderer AI experiment is omitted from this figure, since this particular AI does not create any load. The other experiments can be divided into two categories, the ones with trusted clients (green curves) and the ones with untrusted clients (blue and red curves).

When trusted clients are present in the system, servers will migrate a few master objects to each of these clients, to reduce their own load. In all three experiments, the load steadily decreases until no master objects are left on the server. In the multi-server configurations, the remaining load represents the master cells, which are



(a) Single server environments

(b) Dual servers environments

(c) Quad servers environments

Figure 13.3: Load on First Server (Halo) in Load Balanced Environment.

never migrated to clients.

When no trusted clients are present in the system, the load on the server steadily increases. The initial load on the system, when no active players are present, is about 800. When starting in a static cells dual or quad servers environment (blue curves), the load is respectively approximately 400 and 200. This is because the workload is already divided among the servers. With dynamic cells environment (red curves in (b) and (c)), the initial load is much higher. Dynamic configurations do little partitioning when starting up, they rely on the load balancing algorithm to progressively distribute the workload. This can be seen in the several plateaus on the curve, where the load increase pauses as the load balancing algorithm is distributing the load.

Fundamentally, the load model accurately models the activity found on each node.

The correlation between CPU usage increase and load increase is easy to see for some configurations. In addition, several of the experiments that quickly terminate because of CPU usage overload display sharp increase in load. Unfortunately, there are several weaknesses in the model.

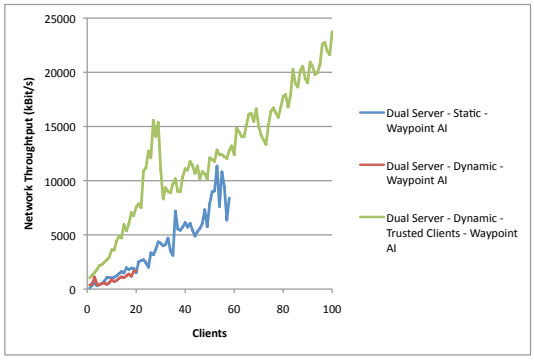
The flat curves of the experiments with trusted clients (green curves) do not accurately reflect the CPU usage of the server. This is especially true for the single server experiment, which suffers from CPU overload at 100 clients, even though its logical load is 0. The model must be expanded to reflect other CPU intensive tasks, such as interest management which consumes more CPU as more clients join the system. In addition, the model does not accurately reflect a unified saturation point. Regardless of the experimental configuration, a given machine should always reach CPU saturation at approximately the same load. Taking into account the experiments with untrusted clients, Halo suffers from CPU overload when the load varies between 500 and 1400. This is a strong indication that either the weights in the load model could be improved, or more CPU intensive activities are missing from the model.

Network Throughput

The final measurement for these nine experiments is the total network throughput: the total number of kB/s sent by all the nodes in the system. The graphs for this measurement, depicted in figure 13.4, illustrate that network traffic increases linearly as clients connect in every scenario.

This linear increase can be explained by the interest management, which reduces the number of updates that must be propagated. Without interest management, the increase would be exponential (see section 5.3.3). In addition, the linear increase features both small and large oscillations. These oscillations are caused by the players continuously on the move, thus constantly changing their field of interest. When a group of players converges towards a particular area, new replicas are created. Network traffic increases, both because of the creation of these replicas and the necessity to update them as players move around. If the players move apart, the updates stop and network throughput decreases. One particular instance of this flocking can be

(a) Single server environments



(b) Dual servers environments

(c) Quad servers environments

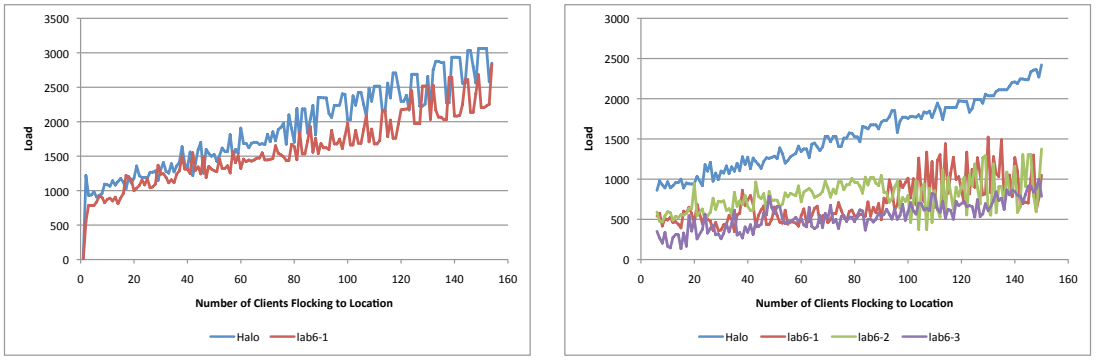
Figure 13.4: Total Network Throughput in Load Balanced Environment.

found in figure 13.4(b) (green curve), once 20 clients had joined the system.

Experimental environments featuring trusted clients generate much more traffic (green curves). This increase is caused by the additional communication between the trusted clients hosting master objects, and the servers, where the replicas are stored. Although the total network throughput increases, the network traffic at each server is reduced. Unfortunately, the network throughput of single servers was not measured.

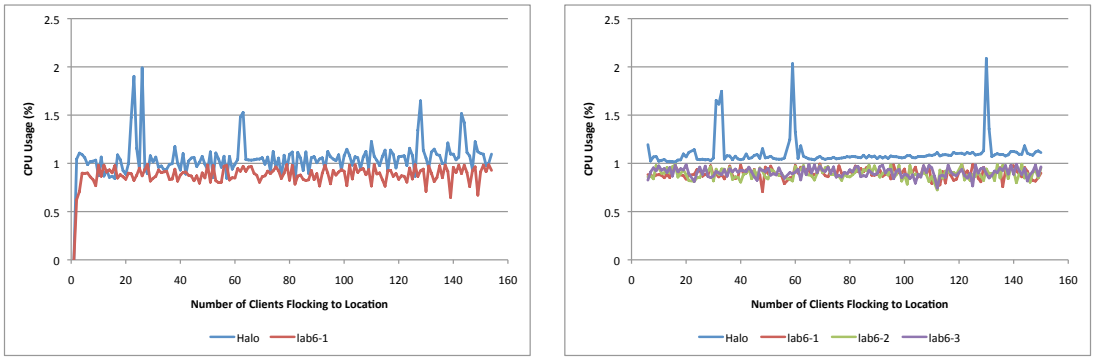
Flocking

An important advantage of dynamic partitioning is the ability to deal with players flocking to a particular location. Although experiments featuring static cells offer



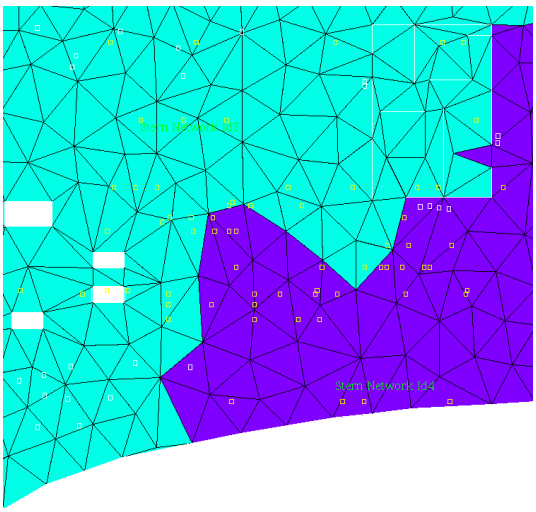
(a) 2 servers (b) 4 servers

Figure 13.5: Load of Servers in Flocking Situation.

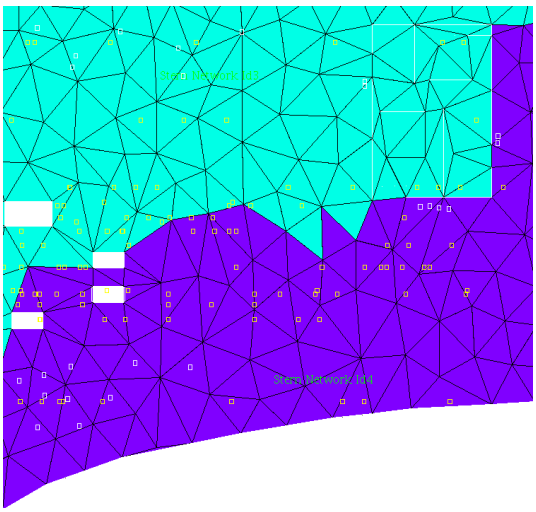


(a) 2 servers (b) 4 servers

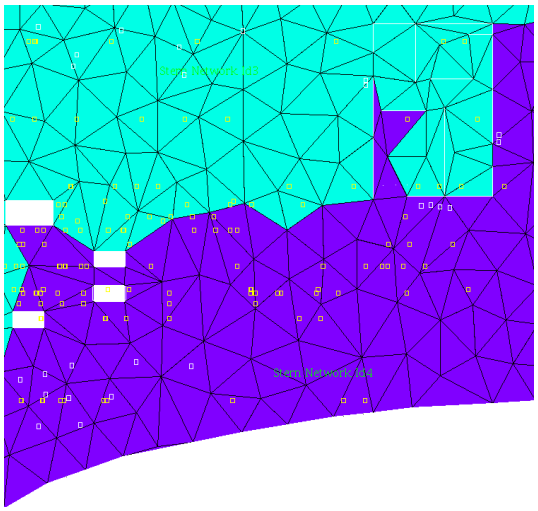
Figure 13.6: CPU Usage of Servers in Flocking Situation.



(a) Initital distribution of cells

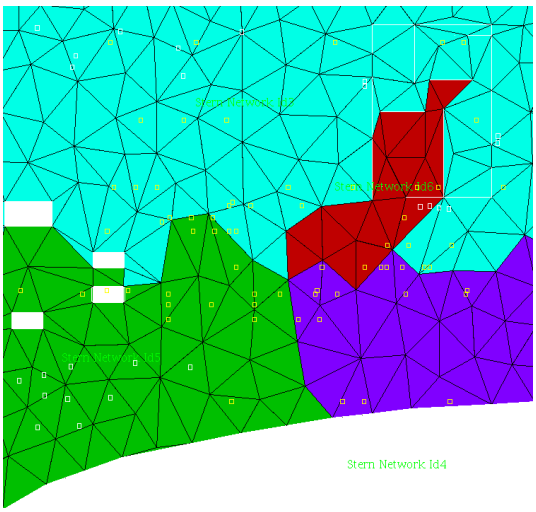


(b) Distribution of cells after 75 clients

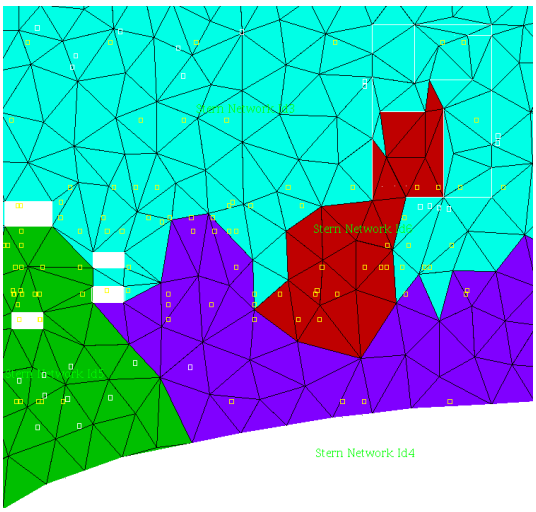


(c) Distribution of cells after 150 clients

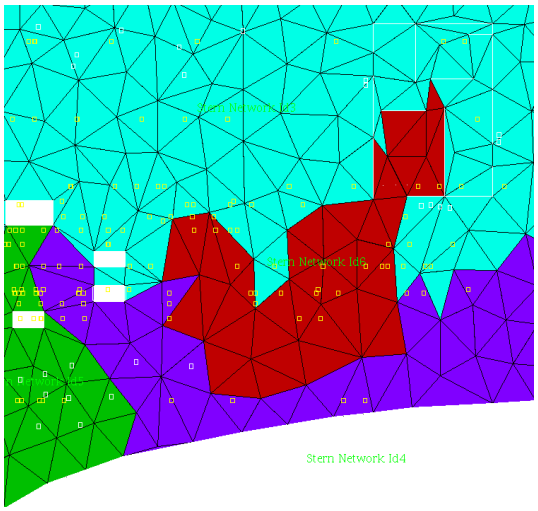
Figure 13.7: Distributions of cells during flocking experiment in a 2 servers scenario.



(a) Initital distribution of cells



(b) Distribution of cells after 75 clients



(c) Distribution of cells after 150 clients

Figure 13.8: Distributions of cells during flocking experiment in a 4 servers scenario.

Figure 13.9: Players flocking to the left side of the map.

better performance in the previous sections, all the benefits of multi-server configurations would be lost if players flocked to a single location. The following two experiments demonstrate how the proposed dynamic cells can cope with load in a flocking situation.

A special hybrid of the wanderer and waypoint AI is used to flock players to the left side of the map (see figure 13.9). The waypoint component is used to move players toward the target area, while the wanderer component is used to keep players active once they have reached the target area. Level 1 trusted clients are used, allowing the experiment to scale to a maximum number of players. The experiment is done using two server configurations, one with 2 servers and one with 4 servers. Load and CPU usage is measured on all servers.

Load linearly increases at the same relative pace on all servers (see figure 13.5), even if players are flocking to an area controlled by a single server. Since level 1 nodes are used, the load presented in this figure is mostly generated by master cells. As load increases on a server, responsibilities, triangle tiles in this case, are transferred to an adjacent server. The algorithm can successfully deal with flocking scenarios, given that servers average a similar load. However, Halo, the initializing server, always has

a higher load. This is a side effect of the startup procedure, where Halo is given all the load and must share it as new trustworthy nodes connect to the system.

A close look at figure 13.5(a) reveals correlated oscillations between the two servers. This behaviour can also be found in the 4 servers experiments, but is difficult to see given the interactions between all 4 servers. These oscillations reveal a weakness in the load balancing algorithm, where tiles are transferred back and forth between two cells. These oscillations can be eliminated using different strategies, such as migrating tiles only once a particular threshold is met.

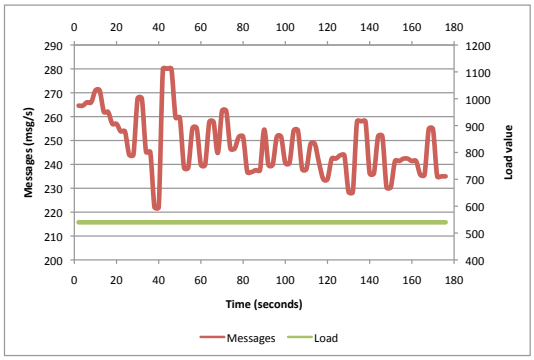
Throughout both experiments, CPU usage remains relatively stable on all servers (see figure 13.6). This demonstrates that load is effectively distributed among the servers as players flocked to the left side of the map. As the first server in the environment and the biggest owner of tiles, Halo is particularly vulnerable to the flocking in this experiment. This explains why Halo's curve (blue) is always slightly above the other servers. The spikes in CPU usage can be safely ignored as artifacts from the JVM, as previously discussed.

The load balancing algorithm was able to share most of Halo's left-side tiles with the other servers, allowing it to cope with the increasing load. This is illustrated in figures 13.7 and 13.8 where triangles are migrated progressively between nodes. In both scenarios, the tiles colored in light blue are owned by Halo. Looking at the dual servers experiment (see figure 13.7), at the beginning of the experiment, Halo owns a little over half of the map and almost all of the left-side tiles. As more clients flock to the left, the second server receives half of the left tiles of the map. In the quad servers experiment (see figure 13.8), the left tiles are divided between Halo and one other server. However, as time progresses, the other servers are given some left-side tiles, sharing the load between the four.

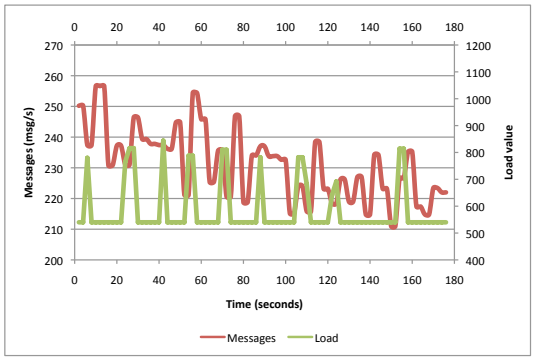
13.3.2 Fault Tolerance

The second set of experiments demonstrates Journey's ability to deal with uncontrolled node disconnection, both in single server and multi-servers environments.

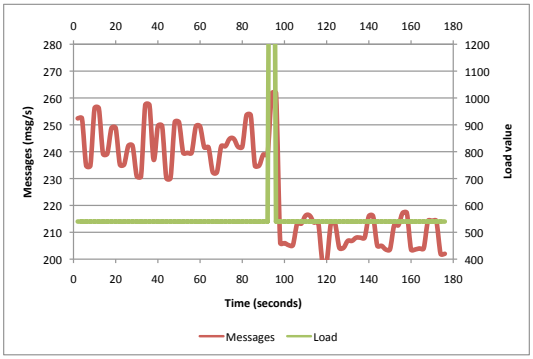
Client Failure



(a) No faults in system



(b) One client crashing every 20 seconds



(c) Crashing 10 clients at one time

Figure 13.10: Effects on Network Throughput of Fault Tolerance System.

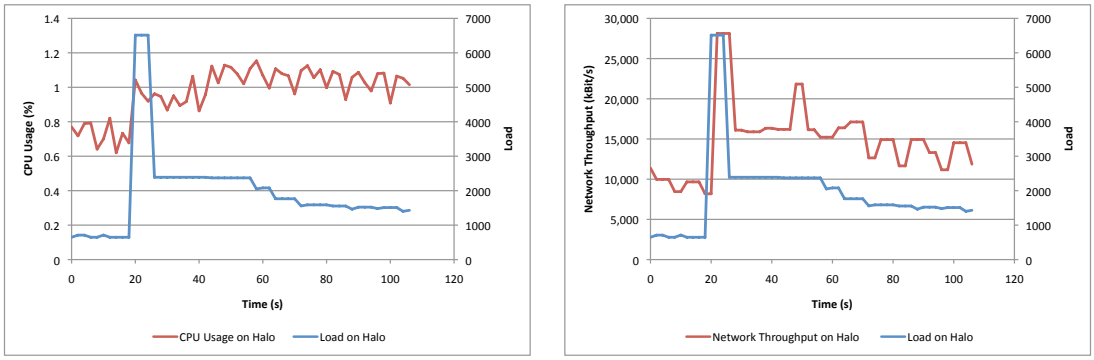
The first set of experiments deals with the loss of trusted clients (level 1 nodes). A fixed number of 50 trusted wandering clients and one server is used in these experiments. Three scenarios are evaluated: no fault, one client disconnecting every 17 seconds and 10 clients disconnecting simultaneously. The fault tolerant component is passive, it remains inactive if no faults are present in the system. Thus, the scenario with no faults can serve as a baseline, it represents the performance of the system without the overhead of the fault tolerant components. The other scenarios provide insight into the impact of recovering from lost clients. Each experiment lasts 3 minutes and both CPU usage on the server and the message throughput (msg/s) are measured. However, the data on CPU usage is not shown, as the experiments

revealed no changes in the CPU usage when clients are lost.

The green curve in figure 13.10 represents the total load on the server. The spikes in the curve indicate the loss of one (see figure 13.10(b)) or 10 clients (see figure 13.10(c)). When a trusted client fails, the server recovers the master objects hosted on that client. This causes a temporary load increase, which disappears when the server migrates these objects to other trusted nodes.

The red curves in figure 13.10 demonstrates a slight increase in the number of network messages when the system is recovering from the loss of a trusted client. The wanderer AIs are unpredictable in their movements and create an uneven network load. However, with 10 simultaneous faults, throughput raises to its highest level. It should also be noted that the number of network message decreases with time because less clients are connected to the system. Thus, any increases in messaging created by the fault tolerant component is hidden by the loss of clients.

Server Failure



(a) Effect on CPU Usage of First Server (Halo) (b) Effect on Total Network Throughput

Figure 13.11: Effect of the loss of a server.

The loss of a server node (level 2) is much more severe, as server nodes host both cells and objects. The following experiment demonstrates Journey's ability to recover from the loss of a server failure with minimal impact. The experimental setup consists of 2 servers and 50 untrusted wandering clients. Untrusted clients are

used this time to keep master objects on the servers, making the recovery process even more resource intensive. Observations on the first server's CPU usage and total network throughput are taken during the first 2 minutes of the experiments, after all the clients are connected. The second server is crashed after 20 seconds of starting the observations, and is restarted 20 seconds latter.

The blue curves in figure 13.11 illustrate the logical load on the first server. After 20 seconds, when the second server is crashed, there is a sharp increase in load. This load spike is caused by the large number of recovered objects: the lost objects are recovered as active, thus earning the higher load value (see section 8.1.3). However, the load decreases as the server determines that some of these objects are not active and assigns them a lower load value. At around 20 seconds, a new server node is connected to the system and the first server starts slowly migrating objects to the new server.

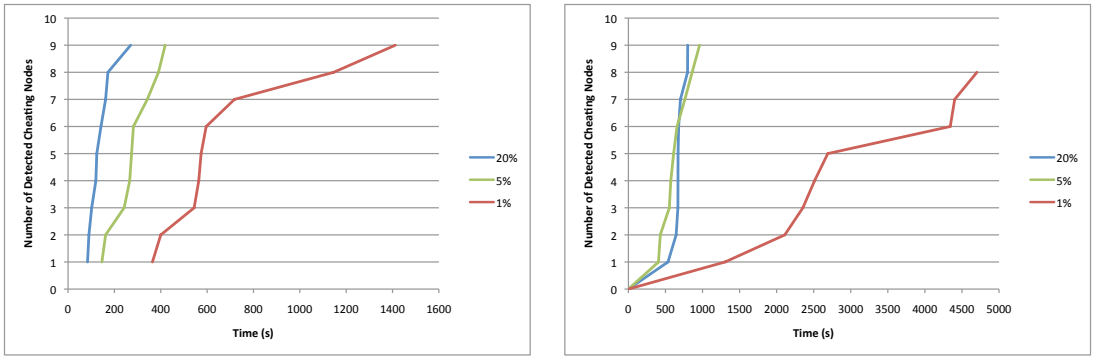
The loss of a server has a definite effect on the remaining server (see the red curve in figure 13.11(a)). The average CPU usage increases by 0.3 when the second server is lost, and remains high, even thought a new server is added to the system. CPU power is needed by the load balancing component to migrate tiles to the new server, so CPU will not drop until the system fully stabilizes itself.

As for total network throughput, there is a sharp increase both at the loss and recovery of the server (see the red curve in figure 13.11(b)). These increases are caused by the recovery of the master objects and their migration to a new server: as new states for these objects are published to the replicas in the system.

This increase in CPU usage and total network throughput would be a concern, if the loss of a server was not an exceptional occurrence. By definition, servers (or level 2 nodes) are the most reliable nodes in the system. Even though the recovery process is costly, the system does recover with time.

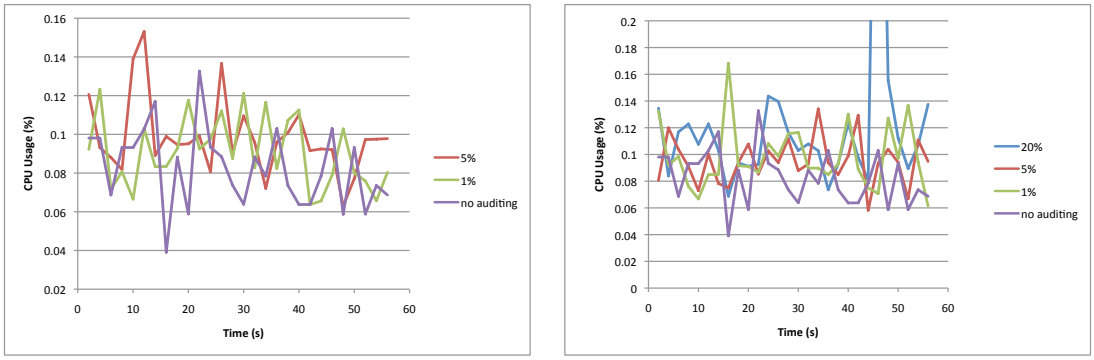
13.3.3 Cheating Players

The experiments in this section demonstrate that a cheater introduced into the system is eventually detected and with minimal performance impact. CPU usage and



(a) Aggressive Auditing (b) Conservative Auditing

Figure 13.12: Time needed to detect cheating players using Auditing.

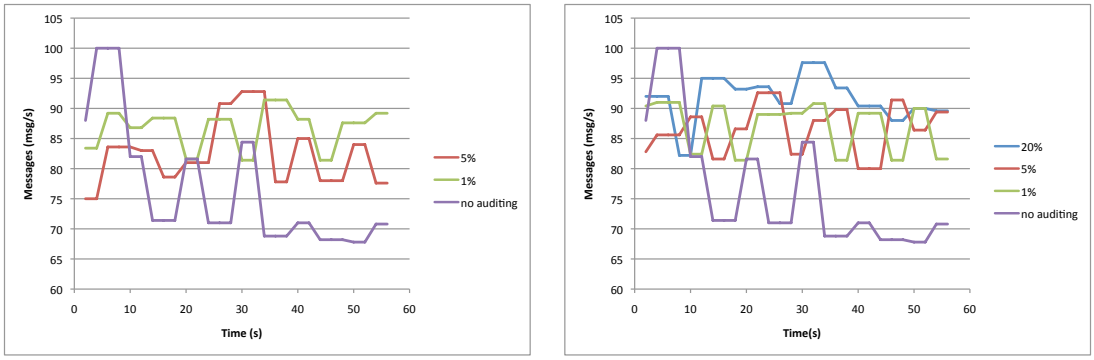


(a) Aggressive Auditing (b) Conservative Auditing

Figure 13.13: CPU Usage on main server (Halo) using Auditing.

message throughput are measured to evaluate the performance impact. Similar to the fault tolerance experiments, a fixed number of 50 trusted players are used in the experiment.. However, 10 of these players are cheaters. Since these players are connected first, they are assigned most of the responsibilities (master objects), and hence given the most opportunities to cheat. Clients are connected progressively to the system, as to not overload the auditing server.

The auditing component is active, as opposed to the passive fault tolerance component: audits are performed regardless of the presence of cheating players. As such, results for experiments without cheaters in the system are nearly identical to those



(a) Aggressive Auditing

(b) Conservative Auditing

Figure 13.14: Message Throughput in System using Auditing.

with cheaters. Thus, the baseline experiment was performed with the auditing system completely disabled, as to better reflect the overhead of using the auditing system.

A cheating node ignores 40% of RPC calls, and moves characters twice as far requested. Only setDestination calls are audited in these experiments. It should be noted that not all faulty setDestination calls are detected, since positions are not compared for equality, but within an equality threshold. If the destination is close to the player, the cheating destination is still considered valid.

Two decision algorithms are used to decide if a faulty player is cheating. The first is quite aggressive and declares a player a cheater after 6 failed audits, regardless, of the number of successful audits. The second, is much more conservative and declares a node a cheater after it has failed 5% of its audits, but only after at least 50 audits are done on that object.

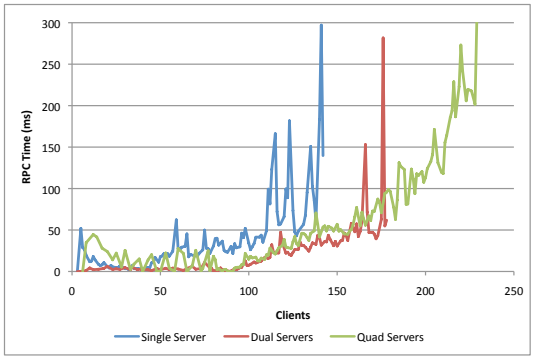
The first figure (see 13.12) illustrates how much time is actually required to detect all the cheaters, depending on the percentage of calls that are audited. The experiments only detected 9 cheaters, as responsability would typically be assigned to only 9 of the cheating nodes. Since a cheating node without responsibilities cannot cheat, it cannot be detected. When auditing a small percentage of RPC calls, a great deal of time is required to detect all cheaters. Nodes that host more than one active players have a significantly higher chance of being audited. If cheating, these nodes

are usually detected first. In addition, it should be noted that the detection speed for the 20% and 5% audits are fairly similar when using the conservative auditing (see blue and green curves in figure 13.12(b)), even though the 20% audit should be much faster. This can be explained by the fact that in our experimental setup, players were added gradually over time, and it takes a certain amount of time for enough players to be in the system to reach the 50 RPC calls minimum. Once that number of calls is reached, the cheaters are quickly identified.

The performance impact of the auditing system on server CPU usage and message throughput can be found in figure 13.13 and figure 13.14. A block of 60 seconds was chosen towards the beginning of the experiment. Note that the 20% auditing rate is not present when using the aggressive auditing, as the 9 cheaters were found and disconnected from the system before proper data could be collected.

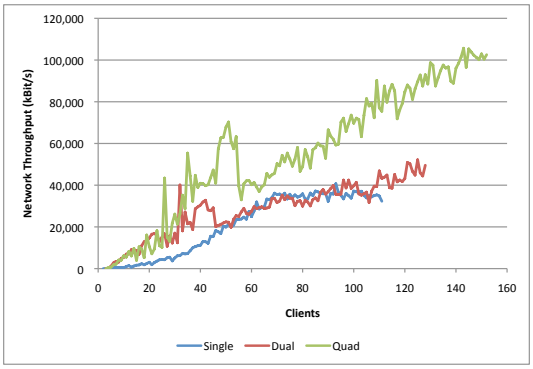
CPU usage (see figure 13.13) varies over time, even when auditing is not used (purple curve). However, on average the server that is responsible for the extra auditing workload, shows some CPU usage increase (blue, red and green curve). In addition, some of the CPU spikes are caused by the JVM, as described in the previous experiments. This is particularly the case in figure 13.12(b) at 50 seconds for the 20% audit.

Message throughput also varies over time (see figure 13.14) ranging from 67 to 100 messages per second, regardless of whether if auditing is used or not. In addition, all curves have numerous spikes that represent sharp increases of messages over short amounts of time. These increases are caused by new clients joining the system every 10 seconds. The curves featuring auditing (blue, red and green curves) are almost always above the no auditing curve (purple curve) confirming that auditing does increase message throughput. Figure 13.14(b) suggests that on average an additional 15 messages per second are added to the system when using the 20% auditing rate. Although not catastrophic, this does represent an approximate 18% increase in the message throughput, which corresponds well with the 20% audit rate.



(a) RPC Time

(b) CPU Usage on Main Server (Halo)



(c) Total Network Throughput

Figure 13.15: Measures taken on Unified Framework.

13.3.4 Load Balancing, Cheating and Auditing Combined

The following set of experiments have one goal, to scale Journey to the highest number of players possible using all its components. As such, Journey is configured using its most efficient setup: using trusted clients and dynamic load balancing. Since these tests measure the overall effectiveness of the system, auditing and fault tolerance are enabled.

Clients are created using machines from the school's computer lab. As such, there is no guarantee that no other user is using the machine during the experiment. As a result, these nodes are sometimes overloaded, even before the experiment. This was not a concern in previous experiments, given that a smaller number of computers

(less than 80) were used. For this experiments, all available computers were used. Fortunately, the overloaded nodes will quickly fail their audit, since they are unable to promptly execute RPC calls. The auditor disconnects these nodes, and the fault tolerance engine recovers the objects hosted on that node. These objects are then quickly redistributed to another node.

The experiment was done using one/two/four servers, and a large number of trusted Waypoint AI clients. Clients were added progressively, every 21 seconds. In all three cases, the experiment is very stable until the average RPC call reaches 300 ms (see figure 13.15(a)). At such time, the servers are saturated and cannot load balance or do proper interest management. Correlating these saturation point with CPU usage data (see figure 13.15(b)), servers cannot cope with the load once utilization reaches 1.5. The CPU usage spike at 60 clients for the dual server configuration can be ignored as a JVM artifact. In a quad server scenario, the system is limited 225 clients. At that point, the network hub is overloaded and cannot route traffic anymore (see figure 13.15(c)). This means the system could be scaled even further if the network hub and the load balancing algorithms were further optimized, as they are the greatest bottleneck in the system right now.

Part IV

Conclusion

Chapter 14
Summary of Work

This thesis introduced Journey, a framework providing a unified approach to load balancing, fault tolerance and cheat detection for massively multiplayer online games. By exploiting the fact that load balancing, fault tolerance and cheat detection can share common data, algorithms and implementation components, a unified framework can provide features that would be impossible to get using these components individually. Most important is the notion of trust, which defines which nodes can be given responsibilities and brings these components together.

This thesis described how such a system can be designed, and demonstrated that the design is valid by implementing the proposed design for the Mammoth MMOG research framework. Finally, extensive experiments have proven that the unified approach proposed by Journey indeed provides more elaborate load balancing, fault tolerance and cheat detection capabilities for MMOGs than would be possible if these problems were addressed individually.

At its core, the Journey framework uses replicated objects. This technology allows game data to be shared across nodes in a distributed system following the object-oriented paradigm in a location-transparent manner. The state of objects in the system is updated using remote procedure calls (RPC), which were optimized for use in a game environment. State updates are propagated to replicas using publish /subscribe technology. The RPC system is implemented using the proxy design pattern, which also simplifies the addition of services relating to logging and auditing.

To provide scalability, Journey partitions the game world using a novel, obstacle-aware triangular partitioning which is used both for interest management and load balancing. Interest management can take advantage of the obstacle awareness of the triangulation to determine a player's area of interest efficiently without actually having to calculate a player's visibility area. Load balancing assigns responsibility of managing the interest of objects within the game world to servers by splitting the world into cells. Cells are composed of a connected set of triangles, and hence also nicely flow around obstacles in the world. They can be resized dynamically depending on the load of each node, or the position of players in the world. This is especially critical in cases of players flocking to a location, leaving a single node to handle the load. If the nodes currently found in the system are unable to handle the total system load, new nodes can be added and load will be dynamically redirected to them.

Given that massively multiplayer systems often involve thousands of participants, either players or servers, it is not unlikely for a node to fail. Fault tolerance is even more important in a framework like Journey, where responsibility is given to trusted clients. Journey provides fault tolerance using the redundant data in the system. The fault tolerance component can tolerate either the loss of a single server, or any number of trusted clients. Lost objects are recovered by the node hosting its corresponding cell. As for cells, they are linked together in a ring topology. If a cell is lost, it is recovered by the node hosting the previous cell in the topology. A fault tolerant migration algorithm for object is also presented, using the node hosting the corresponding cell as a third party to audit the migration and ensure it is correctly completed.

Journey heavily relies on trusted clients to achieve greater scalability. Unfortunately, this makes Journey vulnerable to cheating, as a trusted clients can still cheat. By auditing a small percentage of calls on a trusted node, inconsistent states changes are quickly detected. Nodes hosting cells have absolute trust in the system, and are used as auditors. RPCs to be audited are both executed as normal and forwarded to auditors. These call are executed by the auditor in a separate sandbox environment. This result is then compared to the update propagated by the original executor of the RPC. Although the presence of inconsistencies is possible, the presence of numerous

inconsistencies is a strong indicator of a cheating node. Decision algorithms are used to determined when a node is accused of cheating and what is done with the offending node.

The various experiments done in Mammoth using hundreds of NPC players serve to illustrate the numerous advantages of the proposed system. The load balancing experiments show that Journey is scalable, and that the proposed dynamic cell partitioning can tolerate flocking behaviour of players. Meanwhile, the fault tolerance experiments demonstrate Journey's ability to cope with player and server faults with little overhead. The cheating experiment clearly establishes that the auditing system is able to detect inconsistencies in state changes, while adding minimal overhead to the system.

Finally, a set of experiments with all three components allows Mammoth to scale to 225 clients, a 750% increase beyond its original 30 clients limit. All three components are needed to allow Mammoth to scale beyond its original limit. The massive increase in scalability is provided by the load balancing engine, which migrates resource intensive objects to trusted clients. Even though these clients are trusted, they can still experience faults. The fault tolerance component restores critical data that would be otherwise lost if a client experiences a fault. Trusted clients can still be dishonest; the auditing component is needed to detect and remove them from the system. Without fault tolerance or auditing, it would not be safe to migrate resources to trusted clients. And without the load balancing component, a single server would be overwhelmed by its interest management, fault tolerance and auditing responsibilities. The unified solutions requires all three components, but provides impressive performance improvement while maintaining high reliability.

Chapter 15
Future Work

The research on load balancing, fault tolerance and cheat detection presented in this thesis opens the door for many directions of future work, expanding and improving Journey, as well as its implementation within the Mammoth framework. At the time of writing, there are no less than 15 ideas on the "open projects" page of the Mammoth website, many of them related to replication technology. This chapter focuses on the many possible expansions and improvements.

15.1 Improving the Building Blocks of Journey

Journey is not in any way limited to the features presented in this thesis. Much work can be done to improve the building blocks of Journey, most notably in the RPC, triangle-based partitioning and networking components.

15.1.1 Remote Procedure Calls

The annotation system presented in section 6.2.1 makes the individual customization of remote calls possible. The work presented in this thesis uses simple annotations, such as specifying datasets for method calls. However, the annotation system has much more potential, allowing for customization of how RPCs are executed and how state changes are propagated.

One such example is "premature" remote calls, where method calls that result in updating the state of a game object are executed immediately on the local replica of an object. The call is then forwarded to the master object and executed remotely. If the call succeeds, and the resulting state change is the same as the one executed prematurely on the local replica, the state change is propagated to the other replicas. Otherwise, an error message is sent back to the originating replica and the object is rolled back to its previous state. Although this annotation is not suitable for all calls, actions with high visibility and low side effects such as movement can benefit greatly from this annotation.

15.1.2 Triangle-Based Partitioning

The partition optimizations presented in chapter 5 only scratch the surface of the work that can be done in this field. One key area for improvement is the analysis of the provided geometry that is performed when creating the obstacle map. Recognizing specific elements of the topology of a virtual world is particularly tricky, but would allow for specific optimizations. For example, the current algorithms attempt to recognize walls to transform them into lines and merge the corners, in order to decrease the number of small triangles generated between these edges (see section 5.2.4).

Another key area is the optimization of the obstacle map itself. The optimizations proposed in this work are designed to work best in a 2D space, which is common in modern computer games. Even though most modern MMOG worlds are 3D, the proposed algorithms can still be applied to a projection of the world on two dimensions. However, it would be interesting to see if additional optimizations can be done on a real 3D obstacle map. For example, an optimization that transforms polygonal walls into lines would, in 3D, transform walls into planes instead. Merging points in 3D might also be more complicated, as the closeness threshold might not be the same on each axis.

Finally, the triangulation metric proposed in this work does not evaluate the loss of obstacle data caused by the simplification. In this situation, a simplification that

removes obstacle data most likely improves the score of a triangulation, as less constraints will be supplied to the partitioning algorithm. Although the simplifications proposed in this work are fairly conservative in removing obstacles, an improved metric that accounts for the loss of obstacle data would be useful when developing new simplifications.

15.1.3 Network Engine

A major scalability bottleneck in Journey is Mammoth's network engine. In the unified framework experiment (see section 13.3.4), Mammoth is limited to 225 clients because the network hub is overloaded. Scalability could be greatly increased with a multi-hub system: using multiple hubs to relay network messages to all the nodes. It would also be interesting to explore how a peer-to-peer network engine that uses the existing clients in the system to relay network messages can be used to increase network performance.

15.2 Improving Journey

Although effective, the algorithms used in the design of Journey are sometimes suboptimal. Many optimizations can be done to improve Journey's performance and reliability, most notably in the areas of trust, load balancing, migration, fault tolerance and auditing.

15.3 Trust Service

In Journey, trust data can be acquired from various sources. For example, the current load of a node could be monitored: nodes with low loads could be considered trustworthy. As the same time, a node could be monitored for faults or cheats. Nodes with exemplary records would be considered trustworthy. However, a node disconnecting often, or failing audits should not be considered trustworthy.

Once trust data is gathered, decisions must be made. This is a non trivial problem as false positives, demoting a good and reliable node, should be avoided at all costs. Promoting a non-reliable node is also a problem, as that node can corrupt or cause data loss in case of a fault. The decision process can be improved with a historical trust metric kept in persistent storage; decisions can then be made based on the node's reliability and trustworthiness from previous game sessions.

A key feature in trust systems is the ability to earn or lose trust, which in turn affects the responsibilities that a node can take on within Journey. The current version of the trust service does not allow promotions or demotions, mostly because it does not have the ability to gather trust data and interpret it.

15.4 Load Balancing and Migration

The proposed load balancing algorithms (see section 8.3) in Journey are simplistic in nature: using a rudimentary metric, they try to share the load across trusted nodes in a greedy fashion, making the locally optimal choice at each step. However, existing literature on load balancing suggest other efficient ways to distribute the load. For example, the load balancing algorithm could use global state information about the system to make better decisions. Improving the way load is determined would also be beneficial, as overloaded node would be more effectively detected.

One key element of the load balancing component is the migration technique used to move objects from one node to another. Although both burst and fault-tolerant migration is presented in this work, it would be interesting to explore different migration strategies.

In addition, load balancing would produce more efficient results if object were migrated in groups of related objects. As more work is done on complex actions (actions on multiple objects), the notion of object grouping can be better defined.

15.5 Fault Tolerance and Auditing

The fault tolerance techniques in Journey are only effective under the assumptions presented in section 9.1.3, i.e. that at most one level 2 node fails at a given time, or any number of level 1 nodes but no level 2 node.

Although the unlikely loss of 2 level 2 nodes does not necessarily cause problems, there is a non-negligible risk that some master objects are lost. Future work could investigate if the use of probabilistic replication can provide stronger fault tolerance guarantees. Another way of providing stronger fault tolerance would be to periodically store the state of master objects on persistent storage. During recovery, the state of lost master objects can then be reloaded from persistent storage.

In addition, much work is needed on the decision algorithm of the auditing system. Even if faults and inconsistencies are detected, proper care must be taken before a node is declared a cheater. Journey would greatly benefit from a more elaborate decision algorithm, possibly even a distributed decision algorithm which would require several nodes to declare a node a cheater before actions to ban the node from the game are taken. This avoids giving too much power to a single auditing node.

Appendix A
Appendix A : Algorithms

These are some of the algorithms most commonly used in Mammoth.

When describing the algorithms, the following conventions will be followed.

Letters	Convention
X,Y,Z	Variable amounts of quantity (integer variables).
A,B,C,D,E,F, etc.	Nodes participating to the system.
x,y,z	Coordinates in space.
α, β, γ, etc.	Game objects or cell objects.
mα,dα	Respectively, the master and duplica versions of α.

Table A.1: Letter conventions for algorithms

In addition, the following functions/keywords are defined:

- Node(*object*) : Return the node hosting the master of *object*.
- State(*object*) : Return the state of *object*.
- Send(*message, target*) : Send a *message* to *target* node.
- Publish(*object*): Publish state of *object* to nodes interested in *object*.
- Receive(*message*) : Receive and store the last sent *message*.

- Update(*object, message*): Update *object* using data from *message*.
- Create(*object, message*): Create *object* using data from *message*.
- Wait(*n, function*) : Wait for n rounds of time, then executes *func*.
- Migrate(*object, target*) : Migrate an *object* to target *node*.
- SwitchToDuplica(*object*) : Transform *object* into a duplica.
- SwitchToMaster(*object*) : Transform *object* into a master.
- Execute(*request*) : Execute the RPC call in *request*.
- Audit(*request, ...*): Audit the RPC call in *request* using the other parameters.
- SelectTiles(*source, destination*) : Select tiles found in node A's cell adjacent to node B's cell.

The algorithms used in Journey are as follows:

Algorithm A.1 Burst Migration

Migrating mα from node A to node B.

A SwitchToDuplica(α)
message = State(α on A)
A Send(*message*, B)
B Receive(*message*)
if dα exits on B **then**
  B Update(dα, *message*)
  B SwitchToMaster(α)
else
  B Create(mα, *message*)
end if
B Publish(α)

Algorithm A.2 Fault Tolerant Burst Migration

Migrate mα from node A to node B. The cell master of mα is stored on node C. Lines colored in blue originate from the previous non-fault tolerant burst migration algorithm.

migration_request = Wish to migrate mα to host B.
A send (*migration_request*, C)
C receive(*migration_request*)
C waits(2, *check_for_failure()*)
A SwitchToDuplica(α)
message = STATE (α on A)
A Send(*message*, B)
if Send fails **then**
  A SwitchToMaster(α)
  A Publish(α)
  migration_cancelled = Migration α cancelled.
  A send(*migration_cancelled*, C)
  return
end if
B Receive(*message*)
if dα exists on B **then**
  B Update(dα, *message*)
  B SwitchToMaster(α)
else
  B Create(mα, *message*)
end if
B Publish(α)

check_for_failure():
if (C Not Receive(publication from B) OR (C Not Receive(*migration_cancelled*)
then
  C SwitchToMaster(α)
  B Publish(α)
end if

Algorithm A.3 Recovery of Lost Master Object

In this scenario, Node A has suffered a critical failure. It was hosting the master of α, located in cell β. Node B has noticed the failure and is hosting the master of β.

if host(mβ) = A **then**

The lost of $m\alpha$ will be dealt after the recovery of $m\beta$.

else

B SwitchToMaster(α)

B Publish(α)

B Migrate(α ,different node)

end if

Algorithm A.4 Recovery of Lost Master Cell

In this scenario, node A has suffered a critical failure. It was hosting the master of cell β. Node B has noticed the failure and is hosting the master of cell γ, the predecessor of β.

B SwitchToMaster(β)

B Publish(β)

β will then be migrated to another node the next time a node trusted to receive cells joins the system.

Algorithm A.5 Audit of Method Call

In this scenario, node A wants to execute an RPC call on object α, whose Node(α) is node B. The auditing node for α is C.

request = RPC to execute on α
A Send(*request*, B)
A Send(*request*, C)
B Receive(*request*)
B Execute(*request*)
B Publish(α)
C Receive(*request*)
C Receive(*state_update_of_*α)
if C Audit(*request, state_update_of_*α) = NO **then**
  B might be cheating.
end if

Bibliography

[ACL04] T. Abdellatif, E. Cecchet, and R. Lachaize. Evaluation of a group communication middleware for clustered J2EE application servers. *On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE*, pages 1571–1589, 2004.

[AK88] P.E. Ammann and J.C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, 1988.

[ALR04] A. Avizienis, J.C. Laprie, and B. Randell. Dependability and its threats: a taxonomy. In *Building the information society: IFIP 18th World Computer Congress: Topical sessions 22-27 August 2004, Toulouse, France*, page 91. Kluwer Academic Pub, 2004.

[ANS93] DIS ANSI. IEEE std 1278-1993. *Standard for information technology, Protocols for distributed interactive simulation*, 1993.

[Apa09] Apache. Velocity, September 2009. `http://velocity.apache.org/`.

[Art10] Electronic Arts. Ultima online, March 2010. `http://www.uoherald.com/`.

[ASdO09] D.T. Ahmed, S. Shirmohammadi, and J.C. de Oliveira. A hybrid P2P communications architecture for zonal MMOGs. *Multimedia Tools and Applications*, 45(1):313–345, 2009.

[Asp10] AspectJ Development Team. AspectJ, July 2010. `http://www.eclipse.org/aspectj/`.

[AT06] M. Assiotis and V. Tzanov. A distributed architecture for MMORPG. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 4. ACM, 2006.

[Avi95] A. Avizienis. The methodology of n-version programming. *Software fault tolerance*, pages 23–46, 1995.

[BA08] Eliya Buyukkaya and Maha Abdallah. Efficient triangulation for p2p networked virtual environments. In *NetGames '08: Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 34–39, New York, NY, USA, 2008. ACM.

[BCL+04] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool. The effects of loss and latency on user performance in unreal tournament 2003®. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 144–151. ACM New York, NY, USA, 2004.

[BECM05] R.K. Balan, M. Ebling, P. Castro, and A. Misra. Matrix: Adaptive middleware for distributed multiplayer games. *Lecture notes in computer science*, 3790:390, 2005.

[BK07] H. Backhaus and S. 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*, pages 49–54. ACM New York, NY, USA, 2007.

[BKV06] J.S. Boulanger, J. Kienzle, and C. Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*. ACM New York, NY, USA, 2006.

[BPS06] A. Bharambe, J. Pang, and S. Seshan. Colyseus: A distributed architecture for online multiplayer games. *Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design and Implementation - Volume 3*, 2006.

[Bre10] Seraphina Brennan. Eve online celebrates 54,446 simultaneous users, a new record. *Massively.com*, January 2010. http://www.massively.com/.

[BRS02] A.R. Bharambe, S. Rao, and S. Seshan. Mercury: a scalable publish-subscribe system for internet games. In *Proceedings of the 1st workshop on Network and system support for games*, pages 3–9. ACM New York, NY, USA, 2002.

[CA78] L. Chen and A. Avizienis. N-version programming: a fault-tolerance approach to reliability of software operation. *Fault Tolerant Computing, FTCS*, 8:3–9, 1978.

[Cad08] O. Cado. Propagation of Visual Entity Properties Under Bandwidth Constraints. *Gamasutra. com*, 2008.

[CC06] M. Claypool and K. Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):45, 2006.

[CCP10a] CCP. Eve online, January 2010. `http://www.eveonline.com/`.

[CCP10b] CCP. Eve online faq, January 2010. `http://www.eveonline.com/faq/faq_*01.asp`.

[CDKR02] M. Castro, P. Druschel, A.M. Kermarrec, and A.I.T. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications*, 20(8):1489–1499, 2002.

[CFSS05] C. Chambers, W. Feng, S. Sahu, and D. Saha. Measurement-based characterization of a collection of on-line games. In *Proceedings of the 5th*

ACM SIGCOMM conference on Internet Measurement, page 1. USENIX Association, 2005.

[Cha98] D. Chappell. The trouble with CORBA. *Object News, May*, 1998.

[CP09] Sophie Bernard Charles Premont, Steve Laprise. *Guide de l'industrie jeux video*. Lien Multimedia, 2009.

[CTR06] CTR. Sms case study - eve online and ccp games. *Computer Technology Review*, March 2006. http://www.wwpi.com/.

[CWD+05] J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 289–300. ACM New York, NY, USA, 2005.

[CXTL02] W. Cai, P. Xavier, S.J. Turner, and B.S. Lee. A scalable architecture for supporting interactive games on the internet. In *Proceedings of the sixteenth workshop on Parallel and distributed simulation*, pages 60–67. IEEE Computer Society Washington, DC, USA, 2002.

[CYB+07] L. Chan, J. Yong, J. Bai, B. Leong, and R. 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*, pages 37–42. ACM New York, NY, USA, 2007.

[DB06] D. Demyen and M. Buro. Efficient triangulation-based pathfinding. In *Proceedings of the National Conference on Artificial Intelligence*, page 942. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[DBM07] DBMS2. The database technology of guild wars. *Monash Research Publication*, March 2007. `http://www.dbms2.com/`.

[DEG+04] P. Druschel, E. Engineer, R. Gil, J. Hoye, YC Hu, S. Iyer, A. Ladd, A. Mislove, A. Nandi, A. Post, et al. Freepastry, 2004.

[DG99] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applicationson the Internet. *IEEE network*, 13(4):6–15, 1999.

[Dig09] Screen Digest. There is life beyond World of Warcraft. 2009. `http://www.screendigest.com/`.

[DK06] A. Denault and J. Kienzle. Minueto, a game development framework for teaching object-oriented software design techniques. In *FuturePlay 2006: The International Conference on the Future of Game Design and Technology*, 2006.

[DK10] A. Denault and J. Kienzle. The Perils of Using Simulations to Evaluate Massively Multiplayer Online Game Performance. *DIstributed SImulation & Online gaming (DISIO)*, 2010.

[DTHK05] S. Douglas, E. Tanin, A. Harwood, and S. Karunasekera. Enabling massively multi-player online gaming applications on a P2P architecture. In *Proceedings of the IEEE International Conference on Information and Automation*, pages 7–12. IEEE, 2005.

[DWW05] M. Dick, O. Wellnitz, and L. Wolf. Analysis of factors affecting players' performance and perception in multiplayer games. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, page 7. ACM, 2005.

[DZ03] T.N.B. Duong and S. Zhou. A dynamic load sharing algorithm for massively multiplayer online games. In *Networks, 2003. ICON2003. The 11th IEEE International Conference on*, pages 131–136, 2003.

[EFB01] T. Elrad, R.E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[Ent09] Blizzard Entertainment. World of warcraft, July 2009. `http://www.worldofwarcraft.com/`.

[Exp08a] CCP Explorer. Eve64. *Eve Insider Dev Blog*, October 2008. http://www.eveonline.com/.

[Exp08b] CCP Explorer. stacklessio or: how we reduced lag. *Eve Insider Dev Blog*, September 2008. http://www.eveonline.com/.

[FB74] RA Finkel and JL Bentley. Quad trees a data structure for retrieval on composite keys. *Acta informatica*, 4(1):1–9, 1974.

[FBS07] W. Feng, D. Brandt, and D. Saha. A long-term study of a popular MMORPG. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 19–24. ACM New York, NY, USA, 2007.

[FF03] W. Feng and W. Feng. On the geographic distribution of on-line game servers and players. In *Proceedings of the 2nd workshop on Network and system support for games*, page 179. ACM, 2003.

[FGW06] R.D.S. Fletcher, T.C.N. Graham, and C. Wolfe. Plug-replaceable consistency maintenance for multiplayer games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*. ACM New York, NY, USA, 2006.

[FH02] Daniel Fu and Ryan T. Houlette. Putting AI in entertainment: An AI authoring tool for simulation and games. *IEEE Intelligent Systems*, 17(4):81–84, 2002.

[Fis83] M.J. Fischer. The consensus problem in unreliable distributed systems (a brief survey), 1983.

[FR05] S. Ferretti and M. Roccetti. Fast delivery of game events with an optimistic synchronization mechanism in massive multiplayer online games.

In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 405–412. ACM New York, NY, USA, 2005.

[FTT07] L. Fan, H. Taylor, and P. Trinder. Mediator: a design framework for P2P MMOGs. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 43–48. ACM New York, NY, USA, 2007.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Elements of reusable object-oriented design, 1995.

[Gra72] RL Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[Gro03] A. Grossman. *Postmortems from Game Developer: Insights from the Developers of Unreal Tournament, Black and White, Age of Empires, and Other Top-Selling Games*. CMP Books, 2003.

[GV08] Josh Goodman and Clark Verbrugge. A peer auditing scheme for cheat detection in MMOGs. In *NetGames 2008: 7th Workshop on Network & System Support for Games*, Worcester, MA, USA, oct 2008.

[Haw08] Michael A. Hawker. Subgames in massively multiplayer online games. Master's thesis, McGill University, 2008.

[Hen01] T. Henderson. Latency and user behaviour on a multiplayer game server. *Lecture notes in computer science*, pages 1–13, 2001.

[Hen06] M. Henning. The rise and fall of CORBA. *Queue*, 4(5):34, 2006.

[HK97] Markus Horstmann and Mary Kirtland. DCOM Architecture. *MSDN*, 1997.

[HKU] B. Hardekopf, K. Kwiat, and S. Upadhyaya. *Secure and fault-tolerant voting in distributed systems.*

[HL04] S.Y. Hu and G.M. Liao. Scalable peer-to-peer networked virtual environment. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 129–133. ACM New York, NY, USA, 2004.

[HSPA09] B. Hariri, S. Shirmohammadi, M.R. Pakravan, and M.H. Alavi. An adaptive latency mitigation scheme for massively multiuser virtual environments. *Journal of Network and Computer Applications*, 32(5):1049–1063, 2009.

[IHK04] T. Iimura, H. Hazeyama, and Y. Kadobayashi. Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 116–120. ACM New York, NY, USA, 2004.

[Inc10] Gravity Interactive Inc. Ragnarok online, March 2010. `http://iro.ragnarokonline.com/`.

[KAS07] I. Kazem, D.T. Ahmed, and S. Shirmohammadi. A visibility-driven approach to managing interest in distributed simulations with dynamic load balancing. In *Proceedings of the 11th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, pages 31–38. IEEE Computer Society, 2007.

[KDV07] J. Kienzle, A. Denault, and H. Vangheluwe. Model-based Design of Computer-Controlled Game Character Behavior. *Lecture Notes in Computer Science*, 4735:650, 2007.

[KLXH04] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, 2004.

[KVK+09] J Kienzle, C Verbrugge, B Kemme, A Denault, and M Hawker. Mammoth: a massively multiplayer game research framework. In *Proceedings*

of 5th ACM SIGCOMM workshop on Network and system support for games, Orlando, Florida, USA, 2009. ACM.

[Lab10] Liden Labs. Second life, January 2010. `http://www.secondlife.com/`.

[Lan05] Marc Lanctot. Adaptive virtual environments in modern multi-player computer games. Master's thesis, McGill University, 2005.

[LB06] D. Liang and P. Boustead. Using local lag and timewarp to improve performance for real life multi-player online games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*. ACM New York, NY, USA, 2006.

[LC02] JCS Lui and MF Chan. An efficient partitioning algorithm for distributed virtualenvironment systems. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):193–211, 2002.

[Les05] P. Lester. A* pathfinding for beginners. *Almanac of Policy Issues*, 2005.

[LL03] K. Lee and D. Lee. A scalable dynamic load distribution scheme for multi-server distributed virtual environment systems with highly-skewed user distribution. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 160–168. ACM New York, NY, USA, 2003.

[LLL04] F.W.B. Li, L.W.F. Li, and R.W.H. Lau. Supporting continuous consistency in multiplayer online games. In *Proceedings of the 12th annual ACM international conference on Multimedia*, pages 388–391. ACM New York, NY, USA, 2004.

[LPM06] F. Lu, S. Parkin, and G. Morgan. Load balancing for massively multiplayer online games. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 1. ACM, 2006.

[LS06] Hsiu-Hui Lee and Chin-Hua Sun. Load-balancing for peer-to-peer networked virtual environment. In *NetGames '06: Proceedings of 5th ACM*

SIGCOMM workshop on Network and system support for games, page 14, New York, NY, USA, 2006. ACM.

[LSV06] M. Lanctot, N.N.M. Sun, and C. Verbrugge. Path-finding for large scale multiplayer computer games. In *Proceedings of the 2nd Annual North American Game-On Conference (GameOn'NA 2006)*, pages 26–33, 2006.

[Lua10] Lua Team, Catholic University of Rio de Janeiro. The Lua Programming Language, April 2010. `http://www.lua.org/`.

[Mat03] M. Matskin. Scalable agent-based simulation of players in massively multiplayer online games. In *Eighth Scandinavian Conference on Artificial Intelligence: SCAI'03*, page 153. IOS Press, 2003.

[Mee06] M. Meehan. Virtual property: protecting bits in context. *Rich. JL & Tech.*, 13:1, 2006.

[MFW02] M. Mauve, S. Fischer, and J. Widmer. A generic proxy system for networked computer games. In *Proceedings of the 1st workshop on Network and system support for games*, pages 25–28. ACM New York, NY, USA, 2002.

[MGM06] S. Marti and H. Garcia-Molina. Taxonomy of trust: Categorizing p2p reputation systems. *Computer Networks*, 50(4):472–484, 2006.

[Mic06] Sun Microsystems. Java rmi specification. *Java SE Documentation*, 2006. http://java.sun.com/.

[Mic10] Sun Microsystems. Java remote method invocation. *Java SE Documentation*, January 2010. http://java.sun.com/.

[Min08] CCP Mindstar. my node was equipped with the following... *Eve Insider Dev Blog*, October 2008. http://www.eveonline.com/devblog.asp?a=blog&bid=589.

[Min09] CCP Mindstar. apocrypharrrrrdware! *Eve Insider Dev Blog*, February 2009. http://www.eveonline.com/.

[Min10] Mina Development Team. Apache Mina, August 2010. `http://mina.apache.org/`.

[MLS05a] G. Morgan, F. Lu, and K. Storey. Interest management middleware for networked games. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 57–64. ACM New York, NY, USA, 2005.

[MLS05b] Graham Morgan, Fengyun Lu, and Kier Storey. Interest management middleware for networked games. In *I3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 57–64, New York, NY, USA, 2005. ACM Press.

[Mon07] Curt Monash. The technology of guild wars (overview). *The Monash Report*, June 2007. http://www.monashreport.com/.

[Moz10] Mozilla Developer Center. About Javascript, April 2010. `https://developer.mozilla.org/en/About_JavaScript/`.

[NC04] J. Nichols and M. Claypool. The effects of latency on online madden NFL football. In *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 146–151. ACM New York, NY, USA, 2004.

[Net10] Arena Net. Guild wars, March 2010. `http://www.guildwars.com/`.

[New09] Virtual Good News. Maple story nets $150-500m in revenue for 2008. February 2009. `http://www.virtualgoodsnews.com/`.

[OCS+05] C. Onuczko, M. Cutumisu, D. Szafron, J. Schaeffer, M. McNaughton, T. Roy, K. Waugh, M. Carbonaro, and J. Siegel. A Pattern Catalog For Computer Role Playing Games. In *Game-On-NA 2005 - 1st International North American Conference on Intelligent Games and Simulation*, pages 33 – 38. Eurosis, August 2005.

[OMG09] OMG. Corba basics. *Object Management Group*, November 2009.

[Pau10] Paul Chew. Voronoi Diagram / Delaunay Triangulation, 2010. http://www.cs.cornell.edu/home/chew/Delaunay.html.

[PG07] D. Pittman and C. GauthierDickey. A measurement study of virtual populations in massively multiplayer online games. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 25–30. ACM New York, NY, USA, 2007.

[Pin05] A. Pinto. Appia group communication, 2005.

[Pow10] Mark Powell. Jmonkey engine, January 2010. `http://www.jmonkeyengine.com/`.

[PSL80] M Pease, R Shostak, and L Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[PW02] L. Pantel and L.C. Wolf. On the impact of delay on real-time multiplayer games. In *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, page 29. ACM, 2002.

[QML+04] P. Quax, P. Monsieurs, W. Lamotte, D. De Vleeschauwer, and N. Degrande. Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 152–156. ACM New York, NY, USA, 2004.

[Qua01] Quazal. Quazal Net-Z 2.0, Technical Overview, 2001.

[Qua02] Quazal. Duplication Spaces, Quazal Multiplayer Connectivity, 2002.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, pages 329–350, 2001.

[Rep07] The Monash Report. The technology of guild wars (overview). *Monash Research Publication*, June 2007. `http://www.monashreport.com//`.

[RK07] A. Raja and M. Katchabaw. Using Synthetic Players to Generate Workloads for Networked Multiplayer Games. In *3rd International North American Conference on Intelligent Games and Simulation*, 2007.

[RLT78] B. Randell, P. Lee, and PC Treleaven. Reliability issues in computing system design. *ACM Computing Surveys (CSUR)*, 10(2):123–165, 1978.

[SC94] Sandeep K. Singhal and David R. Cheriton. Using a position history-based protocol for distributed object visualization, 1994.

[Sch09] Johannes Schaback. Feng gui, November 2009. `http://www.fenggui.org/`.

[SGB+03] N. Sheldon, E. Girard, S. Borg, M. Claypool, and E. Agu. The effect of latency on user performance in Warcraft III. In *Proceedings of the 2nd workshop on Network and system support for games*, pages 3–14. ACM New York, NY, USA, 2003.

[She96] J.R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. *Lecture notes in computer science*, 1148:203–222, 1996.

[SKH02] J. Smed, T. Kaukoranta, and H. Hakonen. Aspects of networking in multiplayer computer games. *The Electronic Library*, 20(2):87–97, 2002.

[SM95] L.S. Sabel and K. Marzullo. Election vs. consensus in asynchronous systems. *Computer Science Technical Report ncstrl. cornell/TR95-1488, Cornell University*, 1995.

[Sri95] R. Srinivasan. RFC 1831: RPC: Remote procedure call protocol specification version 2, 1995.

[SSJ+08] A. Schmieg, M. Stieler, S. Jeckel, P. Kabus, B. Kemme, and A. Buchmann. pSense-Maintaining a dynamic localized peer-to-peer structure for position based multicast in games. In *Eighth International Conference on Peer-to-Peer Computing (P2P'08)*, pages 247–256. IEEE, 2008.

[SV98] D.C. Schmidt and S. Vinoski. Object Interconnections: An Introduction to CORBA Messaging. *C++ Report*, 1998.

[Unr07] Unreal Technology. The Unreal Engine 3, 2007. http://www.unrealtechnology.com/.

[V+97] S. Vinoski et al. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, 1997.

[Val09] Valve. Half-life 2, July 2009. `http://orange.half-life2.com/`.

[VBD07] M. Varvello, E. Biersack, and C. Diot. Dynamic clustering in delaunay-based P2P networked virtual environments. In *Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 105–110. ACM, 2007.

[Whi75] J.E. White. RFC 707: High-level framework for network-based resource sharing, 1975.

[WK94] Sara Williams and Charlie Kindel. The Component Object Model: A Technical Overview. *MSDN*, 1994.

[WRW96] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system, 1996.

[YMYI05] S. Yamamoto, Y. Murata, K. Yasumoto, and M. Ito. A distributed event delivery method with load balancing for mmorpg. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, pages 1–8. ACM New York, NY, USA, 2005.

[ZWZ+06] Z. Zhou, H. Wang, J. Zhou, L. Tang, K. Li, W. Zheng, and M. Fang. Pigeon: a framework for testing peer-to-peer massively multiplayer online games over heterogeneous network. In *3rd IEEE Consumer Communications and Networking Conference, 2006. CCNC 2006*, volume 2, 2006.