The Design & Implementation of a Facebook-based Interactive MMOG

Fedor Jutte
Jerry de Swart
The Design & Implementation of a Facebook-based Interactive MMOG

Bachelor’s Thesis in Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Fedor Jutte
Jerry de Swart

2nd July 2010
Author
Fedor Jutte
fejuto@hotmail.com

Jerry de Swart
J.deSwart@student.tudelft.nl

Title
Facebook MMOG

BSc presentation
July 8, 2010

Graduation Committee
Drs. P.R. van Nieuwenhuizen
Delft University of Technology
Dr. A Iosup
Delft University of Technology
Abstract

Massive Multiplayer Online Games (MMOGs) continue to grow in user numbers as well as in the size of the virtual worlds. Another popular trend is Facebook. Facebook is a very popular social networking website. The website currently has more than 400 million active users worldwide. Facebook has diverse social features, and is also a platform for social applications and games from third parties. Even with the growth of MMOGs, very few MMOGs are seen on Facebook. The MMOGs that can be found on Facebook tend to be not very interactive, and distribute users in separate virtual worlds. MMOG architectures struggle with scalability issues that arise when trying to get many players in the same virtual world. The traditional and most common client-server architecture has to cope with the massive bandwidth load required to host an interactive MMOG. Many alternatives and techniques have been proposed to overcome scalability bottlenecks, each with their pro’s and con’s. In this paper we design, implement, and performance test a Facebook-based Interactive MMOG, using a scalability technique called Area-of-Interest (AOI) filtering.
Preface

At the Delft University of Technology, students majoring in Computer Science finish their Bachelors by doing a BSc project. Our particular project is a continuation of a previous research we did as part from an earlier course named BSc Seminarium, where we, together with Peter van der Tak and Simon Smit, did a survey of existing Massive Multiplayer Online Game (MMOG) system architectures. Part of that research would go in depth on which methods MMOGs use to overcome scalability problems in order to support hundreds if not thousands of concurrent players. One particular scalability technique caught our interest, namely Area-of-Interest (AOI) filtering, which is now a major part of our BSc project. The work lying before you is our BSc thesis, in which a description is given of the work done and the results achieved during the project.

Before continuing with our thesis, we would like to thank some people who have been of great help during our BSc project. First of all, we would like to thank Alexandru Iosup for being a fantastic mentor in both our BSc Seminarium and BSc Project. We would also like to thank Peter and Simon for their contributions in the BSc Seminarium. Thanks go to the System administrators Paulo Anita and Munire van der Kyurk for providing and supporting us with test machines. Our last thanks go to the many people we have spoken with on many coffee breaks, in which thoughts were shared on a variety of topics.

Fedor Jutte
Jerry de Swart

Delft, The Netherlands
2nd July 2010
Contents

Preface v

1 Introduction 1
  1.1 Problems of MMOGs . . . . . . . . . . . . . . . . . . . . . . . . 1
  1.2 A scaling technique: Area-of-Interest filtering . . . . . . . . . . . 2
  1.3 Facebook and MMOGs . . . . . . . . . . . . . . . . . . . . . . 2
  1.4 Research Purpose . . . . . . . . . . . . . . . . . . . . . . . . . . 2
  1.5 Approach in Short . . . . . . . . . . . . . . . . . . . . . . . . . 3
  1.6 Thesis Contributions . . . . . . . . . . . . . . . . . . . . . . . . 3
  1.7 Thesis Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2 Previous Work 5
  2.1 A Taxonomy of MMOG Design Topics . . . . . . . . . . . . . . . 5
    2.1.1 Security . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
    2.1.2 Responsiveness . . . . . . . . . . . . . . . . . . . . . . . 5
    2.1.3 Reliability . . . . . . . . . . . . . . . . . . . . . . . . . . 6
    2.1.4 Scalability . . . . . . . . . . . . . . . . . . . . . . . . . . 6
    2.1.5 Resource Ownership . . . . . . . . . . . . . . . . . . . . . 6
  2.2 A Survey of MMOG Systems and Architectures . . . . . . . . . . . 8
  2.3 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

3 Approach 9
  3.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
  3.2 Game Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
  3.3 Game Architecture . . . . . . . . . . . . . . . . . . . . . . . . . 10
    3.3.1 Server . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
    3.3.2 Facebook-Client . . . . . . . . . . . . . . . . . . . . . . . . 10
  3.4 Area-of-Interest Algorithms . . . . . . . . . . . . . . . . . . . . . 10
  3.5 Validation and Testing . . . . . . . . . . . . . . . . . . . . . . . . 10
  3.6 Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

4 Implementation and Validation 13
  4.1 Game Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 13
    4.1.1 Server . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
Chapter 1

Introduction

Massive Multiplayer Online Games (MMOGs) have been a huge commercial success in recent years. Today, millions of people meet and play in virtual worlds with friends and strangers. Game developers use one of the many existing MMOG architectures.

A successful example of an MMOG is Blizzards World of Warcraft. The game has over 11 million players globally, and approximately 1 million players online at any one time. With so many players, it is readily apparent that MMOGs require a significant amount of computing and networking resources. Regardless of its design, an MMOG architecture should keep the game consistent between multiple users, should scale well in order to serve a high number of players and game objects.

1.1 Problems of MMOGs

One mayor issue for an MMOG architecture is to be able to scale. Scalability is generally achieved by sharding: splitting the global game state in smaller independent chunks, called shards, which are then distributed over a server cluster using the client-server architecture. This architecture results in a fragmented gaming community with no guarantees on server or network performance, and often leads to poor gameplay experience. Although this architecture scales well with the number of players, it lacks flexibility and the servers have to be over-provisioned to handle peak loads.

A recent development to MMOG architecture design is to apply peer-to-peer networks in MMOGs. Peer-to-peer networking has so far been used to harness idle storage and network bandwidth, including storage systems, content distribution, and instant messaging. Peer-to-peer architectures in MMOGs utilize memory and CPU cycles of player machines to maintain the shared game state. This architecture opens up possibilities to make games scalable without fragmenting the player base. However, peer-to-peer architectures are complex by nature, introduce new security issues and other concerns to the MMOG architecture design.
1.2 A scaling technique: Area-of-Interest filtering

Most MMOGs will find that bandwidth will be the scaling bottleneck. The most obvious solution to overcome this is to just send less game-state updates. However, this directly results in lag for the player, and thus having a negative effect on the game experience. Area-of-Interest (AOI) filtering is a technique to send less updates, intelligently. The idea of AOI-filtering is to skip sending game-state updates to players that are unlikely to be interested in the update, and in doing so saving bandwidth while maintaining a good game experience. For example, imagine a first person shooter game like Quake, played by 16 simultaneous players. Normally the Quake server broadcasts all game-state changes to all the players. However, a player is not interested equally in each of his opponents. Roughly you could say that a player is more interested in the opponents near him than the opponents farther away from him. Because a player is less interested in the players farther away from him, it won’t hurt his game-play experience much if those player’s who are farther away have a small error in their position. This very fact is used by AOI-filters. An AOI-filter will deduce what game-state updates a player is likely to be interested in, and saves bandwidth by saving bandwidth on all the game-state updates the player is probably not interested in.

1.3 Facebook and MMOGs

Facebook is a social networking website launched in February 2004 and is operated and privately owned by Facebook, Inc. Users can add people as friends and send them messages, and update their personal profiles to notify friends about themselves. Additionally, users can join networks organized by workplace, school, or college. The website’s name stems from the colloquial name of books given to students at the start of the academic year by university administrations in the US with the intention of helping students to get to know each other better. Anyone age 13 or older can become a Facebook user. The website currently has over 400 million active users worldwide. Facebook also hosts many browser based games. It is notable that even though MMOGs have been such a success, there are relatively only a few MMOGs on Facebook. The MMOGs that are playable on Facebook often lack to be very interactive, or use heavy forms of sharding. The very successful MMOGs on Facebook with millions of monthly players even have to use both techniques.

1.4 Research Purpose

The main question we seek to answer is whether it is possible to create a fast paced MMOG for Facebook that can host hundreds of concurrent players.

In order to answer this question, some sub-questions have been formed which focus on the performance of the system:
1. What are the scalability problems of the traditional client-server model?

2. Is the use of an AOI-filtering algorithm a valid approach for scaling MMOGs based on the client-server architecture, in order to support hundreds instead of tenths of concurrent players?

3. How well do “certain” AOI-filtering algorithms make an MMOG for Facebook scale? How many players can be supported by an MMOG that implements a particular AOI-filtering algorithm before the game becomes unplayable? How well does a game using different AOI-filtering algorithms scale under different player densities?

4. Do previous results (showing that an AOI-filtering algorithm is a valid approach for scaling MMOGs) also apply for Facebook-integrated, interactive MMOGs?

1.5 Approach in Short

In order to answer the questions of Section 1.4 a custom 2D game was designed and built. The game is similar to existing mainstream games, thus showing all findings in this thesis are likely also applicable to other games. The game will feature the most commonly used MMOG architecture, which is a client-server setup. Furthermore, the game uses AOI-filtering algorithms for achieving scalability. Multiple filters are implemented, to show the effect of different filters on system resources, and on the player’s game-experience. Then, for answering the first 3 sub-questions from section 1.4, many statistics are gathered by running performance evaluation tests on the system. The game is integrated in Facebook, thus answering the final sub-question.

1.6 Thesis Contributions

In this work our contribution is threefold:

1. we show that AOI-filtering is a viable scaling technique, even when sticking to the traditional client-server architecture

2. we show how different types of AOI-filtering algorithms influence system performance

3. we show that integration of AOI-filtering algorithms into a Facebook-integrated game is possible
1.7 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 will discuss related previous work. Chapter 3 will describe the approach taken. Chapter 4 will go in depth of the inner workings of the build system, including workings of a testing setup used for collecting statistics. Chapter 5 will show and discuss the collected statistics. Chapter 6 will give suggestions for future work. Finally Chapter 7 will shortly summarize the thesis.
Chapter 2

Previous Work

This chapter goes in depth about previously work done. Contents of this chapter are inspired from an earlier thesis, written by us, Peter van der Tak and Simon Smit. The complete thesis can be found in appendix D.

2.1 A Taxonomy of MMOG Design Topics

A taxonomy is required to compare different MMOG architectures. The taxonomy we propose for this purpose consists of security, responsiveness, reliability, scalability, and resource ownership. Each of these directions will be elaborated in this section.

2.1.1 Security

In most multiplayer games security (cheat prevention) is an ongoing issue, and this becomes even more important for MMOG architectures that potentially outsource part of the game state to the clients. This paper only considers in-game security: the ability of individual players to manipulate the game outside of the rules and gain an unfair advantage. We do not consider the problem of hackers attacking the system in order to deliberately shut down part of the system.

2.1.2 Responsiveness

Because MMOGs support many simultaneously playing players it is unlikely that the full game state can be sent to every player every frame. Players however do not want other players to appear to be teleporting from one place to the other because state updates are sent infrequently. Responsiveness considers the techniques that architectures employ to prevent this kind of teleporting. We have selected three commonly used techniques: prefetching, message forwarding, and entity count limiting. Prefetching tries to predict what objects players will be interested in in the near future, sending object updates right before the player actually needs them. Message forwarding improves responsiveness by allowing players with low
upload capacity to forward messages through players with higher upload capacity in order to deliver more updates on time. Finally, player count limiting simply avoids excessive load by limiting the amount of players within a specific part of the world.

2.1.3 Reliability

A reliable MMOG system architecture is resilient to failures of some of its components. Architectures differ greatly in the amount of components that are allowed to fail and what is done to avoid the complete system to go down. In order to compare systems on reliability, we only consider whether a design contains a single point of failure. An architecture has a single point of failure iff at least one component exists that interrupts the game if only that component fails.

2.1.4 Scalability

Different architectures use different techniques to allow many players to simultaneously play a MMOG, among these techniques are interest set filtering, sharding, and zoning. By interest set filtering we mean that peers only receive updates (or receive updates more frequently) for entities they are probably interested in. By exploiting a player’s interests, the bandwidth requirements for updates can be greatly reduced. Another technique to support more players to play at the same time is to divide the game world into zones and having different servers manage different zones. Zones are distributed over servers so that servers only need to communicate with players within their own zone(s). The last technique for scalability is sharding: running multiple copies of the (same part of the) game world without interaction. Each shard is maintained by one server, which is scalable because each server only needs to support a subset of the active players.

2.1.5 Resource Ownership

Because purchasing and maintaining (or renting) servers costs a lot of money, some designs have devised a way to allow players to host part of the MMOG. Allowing players to host part of the game greatly reduces hosting costs, but it also jeopardizes security. In order to keep the game secure, a middle path can be chosen by using (trusted) community hosted servers which avoids outsourcing control completely to individual players. There are therefore three features to be considered: player owned resources, community owned resources and publisher owned resources.
Table 2.1: Comparison of MMOG system architectures based on the taxonomy introduced in Section refsec:taxonomy. Each row represents a single MMOG system architecture. Each column represents a single feature from the taxonomy. Architectures implementing that feature are marked with an ‘x’ in the corresponding cell.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Security</th>
<th>Responsiveness</th>
<th>Reliability</th>
<th>Scalability</th>
<th>Resource ownership</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIVE (1998)</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Mercury (2002)</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MiMaze (2002)</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grid (2003)</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVE Online (2003)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zoned model (2004)</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEO (2004)</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peer-to-peer (2004)</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>IBM (2006)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM Grid (2006)</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>VON (2006)</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Colyseus (2006)</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PSMMO (2006)</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Hydra (2007)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zonal MMOGs (2007)</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid (2008)</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Donnybrook (2008)</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.2 A Survey of MMOG Systems and Architectures

2.3 Discussion

The architectures described in Section 2.2 mainly focus on the technical aspects of game operations such as responsiveness, reliability, and scalability; security is generally considered to be out of scope. Peer-to-peer architectures typically shift control from publisher-owned servers towards potentially untrustworthy peers, which introduces security risks that do not exist in client-server architectures; for example, players can add items which they are not supposed to have to their inventory.

The use of community owned resources in a peer-to-peer based architecture has not yet been studied thoroughly. Table 2.1 shows that community resources are only used in one architecture, which studies the use of community resources in a client-server based architecture. Community resources can potentially be used to overcome resource limitations, such as bandwidth limits.

Other aspects exist than those listed in Section 2.1; for example consistency – which differences are condoned to exist in the local view of the game world between different players – and content generation – to what extent can players or communities add game content, changing game objects or the game world.

What’s notable is that a considerable amount of Architectures used some form of Area of Interest filtering for scalability.

To our knowledge, none of these architectures have been applied directly on browser-based games, like those playable on Facebook.
Chapter 3

Approach

This section chapter describes which approach is taken in order to answer the questions of Section 1.4. The chapter starts with a section overview, giving a birds-eye view of the approach as a whole, followed by sections describing the approach of individual components.

3.1 Overview

To answer the questions from Section 1.4 we will build a custom MMOG. There are four major components needed for achieving this. First, there is the game-design. Game-design meaning the workings of the game from a players perspective. This includes what a player will see on the screen, and which actions a player can take. Game-design is discussed in Section 3.2. The second component is actually building the game. Which MMOG architecture is chosen, and which approach is taken to build it is discussed in 3.3. The third component is the approach to setting up a testing environment. This includes how to simulate real players, and which parameters are varied. The approach taken to set up a testing environment is discussed in Section 3.5. The final component is which statistics should be gathered to get an effective view of the effects the parameters have. Which statistics are chosen to be gathered are discussed in Section 3.6.

3.2 Game Design

The game-design should be simple to implement while being representative for MMOGs. The game design is meant to be a basis for a very effective platform to conduct tests on and measurements system performance. Therefore the game is a free-for-all, 2D space shooter. The game world will be a single circular plane whereon player-controlled spaceships will battle. Each player gets to control a spaceship. A spaceship will always fly at a constant speed forward. The player can manoeuvre his spaceship through the game world by steering left or right. Furthermore, a player can make his spaceship shoot a bullet every few seconds.
A bullet has a maximum lifetime set to a few seconds, and will be removed from the game-world if its lifetime expires. While alive, a bullet will fly in a straight line with constant speed until impact. If a bullet hits a spaceship both the bullet and spaceship will die, and will be removed from the game-world. The player who shot the bullet will get a point, and the killed player is able to re-spawn in a new random location in the game world.

3.3 Game Architecture

The game will be built as a client-server architecture. This means there is a single server, where multiple clients can connect to. There is no communication between the clients themselves. The server and client implementations are discussed in the next two sections.

3.3.1 Server

The server will maintain the (true) game-state, and run all game logic. Game-state updates will be sent to the clients. In order to answer the questions from Section 1.4, the server will be designed to be able to apply various AOI-filtering algorithms.

3.3.2 Facebook-Client

The Facebook-client is the visual portal for a human player to the game world managed by the server. The player sees his and other ships, and control his ship. Furthermore, there will be heavy integration with Facebook by making use of the features that Facebook provides trough the Facebook-API.

3.4 Area-of-Interest Algorithms

To achieve scalability, we will make use of various AOI-filtering algorithms. All game-state will be managed by the Server. The traditional approach is to send each game-state update to each client. Instead of sending game-state updates to clients directly, all game-state updates will first be put through an AOI-filter implementation. The AOI-filters will filter updates based on how likely a player is interested in a particular game-state update.

3.5 Validation and Testing

In order to gather statistics, we build a testing environment. The testing environment consists out of two machines: the server described in ??, and a seperate bot machine. The server will run in the same manner, but instead of human players connecting, the bot machine will request connections. The bot machine will send the same messages as if a Facebook-Client was connected. This setup gives the
freedom to easily change conditions. Both the server and bot machines will keep track of various statistics, to be evaluated later. The conditions we will vary are:

1. different quantities of players
2. different AOI-filtering implementations
3. different player behaviours
4. different world sizes

To evaluate whether the use of an AOI-filtering algorithm is a valid approach for scaling up to hundreds of simultaneously playing players, the same measurements will be taken using different AOI-filtering algorithms. Then the bottleneck(s) are investigated. By using an AOI-filtering algorithm, certain messages will be filtered out. This approach would not be of any use if the game-play suffers too much from it. Therefore the difference in the player’s view and the game-state of the server will also be taken into account in the measurements.

3.6 Metrics

This section describes which statistics will be gathered. The metrics we will use are listed here, and the elements will be explained directly after.

1. System responsiveness
2. Delta view
3. Messages per second
4. CPU-load
5. Bandwidth

System responsiveness  Measuring the response-time of the server by measuring the difference between sending a ”shoot” message to the server, and receiving a the resulting message.

Delta view  Measures how much the view the client sees on his/her screen differs from the actual game-state present on the server.

Messages per second  The number of messages sent and received by the server.

CPU-load  Percentage of usage of the server-CPU.

Bandwidth  Used bandwith from the server in KB/s in both up and downstream.
Chapter 4

Implementation and Validation

4.1 Game Architecture

As mentioned in 3.3, the server will be based on a Server-Client architecture. In this section, we will show how the Server, Facebook-client and message-protocol are implemented. Rather than going into every implementation detail, the complete implementation was stripped down to its essentials for explanation purposes. This section’s goal is to show out of what components the implementation consists, and how they are coupled to give the reader insight in the implementation’s structure.

4.1.1 Server

This subsection will give insight in the server-side implementation. For this purpose, first we will list used design patterns. Then, several UML diagrams are used to illustrate the implementation. A class diagram is presented with commentary on the different classes’s responsibilities. Then, two sequence diagrams are presented. The first diagram shows the initialization sequence of the Server, and the second diagrams shows how received messages from a client ultimately lead to game-state updates.

Core Patterns

The design is based on three design-patterns: The Client-Server pattern, Observer pattern and model-view-controller pattern.

The client-server pattern (http://en.wikipedia.org/wiki/Client-server_model) is used for establishing connections. It’s considered to be the easiest and most popular pattern on the multiplayer-game market today. In the client-server pattern a single server will own and run all game-state. Clients connect to this server, receive game-state updates from the server and send player actions to the server. There is no clustering or peer-to-peer communication. In the design we will take this basic pattern and add AOI-filters for scalability.
The server uses the Observer pattern (http://en.wikipedia.org/wiki/Observer_pattern) to implement various AOI-filtering algorithms. The idea is to have a central class, the Subject, that owns the complete game-state. An AOI-filter, the Observer, is notified on all game-state changes and can then filter the changes before sending updates to the clients.

The server uses the model-view-controller pattern (http://en.wikipedia.org/wiki/Modelviewcontroller) as the central server-side pattern. Game-state is the model, the AOI-filter is the viewer, and a controller which takes input from clients to manipulate the game-state.

Class diagram
The server consists of 6 core classes. These classes with their interconnections are shown in laid out in Figure 4.1.

![Class Diagram](#)

**Figure 4.1: The server-design classdiagram.**

The **Main** class is the entry point. The class is responsible for global initialization and **Server** start-up.

The **Server** class is responsible for accepting incoming connections, and handling them by creating **Client** instances.

The **Client** class represents a client being served. This class handles communication between **Game** and a client.

A **Socket** instance is used by a **Client** instance for pushing and pulling communica-
tion messages from and to the network. Paragraph 4.1.3 will give more information on what data exactly is transported over the network.

The **Game** class represents the game as a whole. This includes all game-logic and game-state. When game-state changes, the **Game** instance will notify filters of this change.

The **Filter** class is the class that will perform AOI filtering. The filter will get notifications of Game-state updates, and decide whether or not to send these updates to the client.

**Sequence diagrams**

![Sequence diagram](image)

Figure 4.2: Server initialization sequence diagram.

Figure 4.2 shows the start-up sequence of the Server. **Main** creates and starts a **Server** instance. **Server** creates in turn a **Game** instance. The **Server** instance will then in an infinite loop wait for client connection requests. If a connection request is received, all the server does is create a **Client** instance to handle the new connection, and start with it’s next iteration. **Client** will create a **Filter** instance and a **Socket** instance, which will continuously be open for messages send from the client. This completes the initialization process.
What happens when a message is received from a client is detailed in Figure 4.3. The received message will be passed to the corresponding Client instance, which will process the message, to then be translated into game-state changes on the Game instance. Game will notify all Filter instances of these changes. A Filter could then decide to send its client the update or not. This sequence will repeat itself for every message received from the client.

### 4.1.2 Facebook-Client

The client runs on Adobe’s Flash platform, with Facebook integration features. The game features six integration points with Facebook. They will be listed here, followed by an explanation for each of them.

1. Facebook Canvas
2. Stream publishing
3. Friend invitations
4. Bookmarking
5. Highscore with pictures
6. Indication of friends
**Facebook Canvas**  Facebook’s Canvas is a webpage hosted on Facebook, which will load the game in an iframe. The Canvas helps players stay within the Facebook experience, because they never have to leave Facebook to play the game.

**Stream publishing**  The stream is shown immediately to users upon logging into Facebook, making it core to the Facebook experience. The game prompts a user to publish stories about what she is doing in the game to the user’s wall. If the user published the story, it will appear on his/her wall and in all his/her friend’s streams. Stories published from the game will include a link to the game’s canvas page. Stream publishing enables new users to discover the game, and existing users to re-engage with it.

**Friend invitations**  Friend invitations are a great way to enable users to invite their friends to the game, and for users to confirm connections with the game. As an example: if user A wants to invite user B to join the game, user A could send user B a request, asking if user B would like to join. User B will receive the request and have the opportunity to respond. Users can easily send requests by simply clicking on the invite link above the game.

**Bookmarking**  Bookmarks enable users to easily navigate back to the game. Above the game next to the invite link, there is a Bookmark button on the canvas page. It enables users to add the game to their list of bookmarks on the left column of the homepage.

**Highscore with pictures**  Using personal pictures in any application greatly helps with a social experience. The game features a high score list, keeping track of the winning players in real time throughout the game. The high score list shows the top few players, where next to each player’s name and score, his/her picture is shown.

**Indication of friends**  Some players popping up in the high-score list might be one of a user’s friends. The game will highlight the user’s friends in the high score list so that the player can see one of his/her friends is doing well.

### 4.1.3 Communication protocol

Client and server communicate by means of messages. There are two types of messages: the ones that the clients send to the server, and the ones the server sends to the clients. All messages are listed in Table 4.1. Each message starts with an header byte, with all the data contents followed (if any). Messages with the header 0 through 8 are messages that clients can send to the server, and the remainder of the messages 9 through 20 are the server can send to the clients.

When a client wants to connect to the server, there is a predefined start-up sequence. The start-up sequence is visualized in Figure 4.4. Right after the client
requests a connection from the server, the client sends an identify message. With the identify message the client tells the server what his nick-name is, and other client specific information. The server will then respond with a welcome message, providing his unique playerID. Finally, the server sends a game snapshot. The game snapshot contains all of the game-state, including which players are already playing, and where all the ships are. After this, the client is fully connected. From here on the client can send any of the other to-server messages, and receive any of the other to-client messages.

![Diagram](image)

Figure 4.4: Connecting to the server.

4.2 Area-of-Interest Algorithms

As explained in 4.1.1, an AOI-filter gets notified of game-state updates, and can then decide to send this update to its corresponding client. The game can opt to use one of three implemented filters. How these three filters work will be explained in this section. Figures 4.5, 4.6 and 4.7 visually illustrate the filters. Each figure is seen from a player’s perspective. The playing player is the blue ship, the ships from which game-state updates are filtered are coloured red, and likewise, the player keeps receiving updates from the green coloured ships.

**No-Filter** The No-Filter won’t filter any update. The Filter is illustrated in Figure 4.5. This filter will represent the traditional case where every update is broadcasted to every player. Since no updates are skipped, lag is minimized. However, since no update is filtered, this filter will use most bandwidth too.

**Threshold-Filter** This filter will filter any game-state update that is further away from a player’s ship than a predefined distance. The filter is illustrated in Figure
<table>
<thead>
<tr>
<th>Message</th>
<th>Header</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identify</td>
<td>0</td>
<td>name:String</td>
</tr>
<tr>
<td></td>
<td></td>
<td>isBot:boolean</td>
</tr>
<tr>
<td></td>
<td></td>
<td>isFacebook:boolean</td>
</tr>
<tr>
<td></td>
<td></td>
<td>facebookID:String</td>
</tr>
<tr>
<td></td>
<td></td>
<td>friendIDs:int[]</td>
</tr>
<tr>
<td>GetTime</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SpawnMe</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>StartShooting</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>StopShooting</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>StartRotatingLeft</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>StopRotatingLeft</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>StartRotatingRight</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>StopRotatingRight</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Welcome</td>
<td>9</td>
<td>playerID:int</td>
</tr>
<tr>
<td>Time</td>
<td>10</td>
<td>serverTime:long</td>
</tr>
<tr>
<td>GameSnapshot</td>
<td>11</td>
<td>gs:GameSnapshot (complete game-state)</td>
</tr>
<tr>
<td>MoveMoveableResult</td>
<td>12</td>
<td>moveableID:int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>newX:float</td>
</tr>
<tr>
<td></td>
<td></td>
<td>newY:float</td>
</tr>
<tr>
<td>SetRotationMoveableResult</td>
<td>13</td>
<td>moveableID:int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>newRotation:float</td>
</tr>
<tr>
<td>CreatePlayerResult</td>
<td>14</td>
<td>playerID:int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>name:String</td>
</tr>
<tr>
<td></td>
<td></td>
<td>facebookID:String</td>
</tr>
<tr>
<td>RemovePlayerResult</td>
<td>15</td>
<td>playerID:int</td>
</tr>
<tr>
<td>SetScoreResult</td>
<td>16</td>
<td>playerID:int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>newScore:int</td>
</tr>
<tr>
<td>CreateShipResult</td>
<td>17</td>
<td>shipID:int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x:float</td>
</tr>
<tr>
<td></td>
<td></td>
<td>y:float</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rotation:float</td>
</tr>
<tr>
<td>RemoveShipResult</td>
<td>18</td>
<td>shipID:int</td>
</tr>
<tr>
<td>FireResult</td>
<td>19</td>
<td>bulletID:int</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x:float</td>
</tr>
<tr>
<td></td>
<td></td>
<td>y:float</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rotation:float</td>
</tr>
<tr>
<td>RemoveBulletResult</td>
<td>20</td>
<td>bulletID:int</td>
</tr>
</tbody>
</table>

Table 4.1: Protocol messages.
4.6. The distance is chosen so that the threshold is bigger than that’s visible on the player’s screen. Because the threshold is outside of the player’s screen, the experience the player gets is equal to that of the No-Filter. This filter is expected to be most efficient when players are distributed over the play-field and there is little clustering.

**Closest-Filter** This filter will only send updates of the six closest entities (either a ship or bullet) relative to the player’s ship. For illustration purposes, Figure 4.7 updates only the closest two ships. If there are less than six entities on the screen of a player, the player will achieve the same experience as provided by the No-Filter. What’s most important to note here is that this filter is the only implemented filter that has a mechanism to deal with clustering. Even when all players of the game fly near each other, each player would still only receive updates from the six closest entities, preventing the bandwidth usage to exponentially grow.

### 4.3 Validation and Testing

TODO: Follow chapter 3.
4.3.1 Bots

In order to simulate different kinds of human behaviour there are 3 kinds of bots. The first type is the *defensive* bot. This bot will defend the area where it spawns. When an opponent comes too close, the bot will attack. In a game with only this type of bot, the bots remain fairly spread out across the field because they will stick around their spawning location, which is random. The second type of bot is the *offensive* bot. The offensive bot searches for an opponent all the time. When selecting one, it has a 90% chance of focusing the closest bot, and a 10% chance of picking an uniform randomly picked target. When a target is focused, it will only choose a new target when the current target died. The third type of bot is the *king-of-the-hill* bot. This bots simply targets the opponent that was the last to spawn, and will only fire when the focused target is in sight. In a game with only this type of bot, a lot of clustering will happen, because all bots will follow one target.

4.3.2 Logging the measurements

In order to measure system-performance of the server and bots, data has to be logged while interrupting the other components as less as possible. For this purpose, a logging system was created that runs in a separate thread, and uses a large buffer to keep the logs in memory. At the end of the test-session, the logs are being written to the harddrive just before the server or bot is being killed. During the start-up of the server and bots, the right logging-level can be set in order to enable...
logging. The output will be raw data, ordered in a one-line String. A typical String output containing a single measurement used the following format:

- the measurement symbol: "*>
- the day, month and year of logging: for example "28-5-2010"
- the exact time containing the hour, minute, second and millisecond of logging: for example "1:39:40:724"
- the log-level: for example "FINE" or "INFO"
- the raw data to be logged: for example "29.131391240583962 4642.490500633291"

The output will be put into a file containing the name of the class that is doing the log. This way the data can easily be found.

4.3.3 Testsuite

The testsuite consists of a series of scripts written in bash. It is fully automated, so an entire test-run that can possibly go on for days can be started by a single command and does not require any user intervention. Its main responsibilities are to upload the server-application to the server and the bots to the workstation, start the server and bots with the appropriate settings, retrieve the logfiles after the test-run, and analyse them by extracting the needed statistics and creating the graphs.
Chapter 5

Experimental Results

In this chapter we present the setup and results of the experiments to see if it is possible to create an MMOG for Facebook that can host hundreds of concurrent players. To answer sub-questions as stated in Section 1.4, several experiment-setups were formulated in Section 3.5. The results of these setups are presented and described in Sections 5.2 to 5.5. Table 5.1 shows an overview of what each experiment covers.

Table 5.1: The parameter selection for the experiments presented in this section. The parameters in bold indicate the main focus of each section.

<table>
<thead>
<tr>
<th>Section</th>
<th>Number of Players</th>
<th>AI Type</th>
<th>Field Size</th>
<th>AOI Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section 5.2</td>
<td>10-120</td>
<td>Playermix 3*</td>
<td>4000</td>
<td>No-filter</td>
</tr>
<tr>
<td>Section 5.3</td>
<td>10-150</td>
<td>Playermix 3*</td>
<td>8000</td>
<td>ALL</td>
</tr>
<tr>
<td>Section 5.4</td>
<td>32</td>
<td><strong>Playermix 1-7</strong></td>
<td>4000</td>
<td>ALL</td>
</tr>
<tr>
<td>Section 5.5</td>
<td>32</td>
<td>Playermix 3*</td>
<td><strong>1000-6600</strong></td>
<td>ALL</td>
</tr>
</tbody>
</table>

*a mix of 33% defensive, and 66% offensive bots.

5.1 Experimental Setup

This section describes the experimental setup used to measure system performance in Sections 5.2 to 5.5.

5.1.1 Hardware

All tests were executed using two computers, one for the server-application and one for the bots. The server-application was running on a local (headless) server, and for the emulation of the bots a workstation was used. Both computers were interconnected by a gigabit LAN-connection.
Server specifications

Table 5.2 shows the specification of the server used to run the server-application.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2x Intel Xeon (X5570)</td>
</tr>
<tr>
<td>CPU frequency</td>
<td>2.93 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>8</td>
</tr>
<tr>
<td>Threads</td>
<td>16</td>
</tr>
<tr>
<td>Ram</td>
<td>32GB</td>
</tr>
<tr>
<td>Operating System</td>
<td>64-bit Fedora Linux</td>
</tr>
<tr>
<td>Kernel</td>
<td>Linux 2.6.32.11-99.fc12.x86_64</td>
</tr>
</tbody>
</table>

Table 5.2: Server specifications.

Workstation specifications

Table 5.3 shows the specification of the workstation that executed the bots.

<table>
<thead>
<tr>
<th>Specification</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2x Intel Xeon (E5345)</td>
</tr>
<tr>
<td>CPU frequency</td>
<td>2.33 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>8</td>
</tr>
<tr>
<td>Threads</td>
<td>8</td>
</tr>
<tr>
<td>Ram</td>
<td>32GB</td>
</tr>
<tr>
<td>Operating System</td>
<td>64-bit Fedora Linux</td>
</tr>
<tr>
<td>Kernel</td>
<td>Linux 2.6.23.15-137.fc8</td>
</tr>
</tbody>
</table>

Table 5.3: Workstation specifications.

5.2 The Impact of the Number of Players on System Performance

In this section we present the experimental evaluation of the traditional client-server model that does not make use of any AOI-filtering algorithm. The experiments are aimed at uncovering the scalability limitations of such a system.

5.2.1 Results

Figure 5.1 shows the responsiveness of the system in terms of the delay between a spawn-request and server-reaction. This measurement is important to the players, because it shows the delay between an action and server-reaction.

Until 60 bots all spawn-responses are comfortably being received within the 100 milliseconds range. With a 100 concurrent playing bots the mean response-times are still well below 100 milliseconds, however some unacceptable exceptions
(response-times far above 100 milliseconds) start to show. At 110 bots, not a single spawn-response is being received within 300 milliseconds. Therefore the responsiveness of the system suffers too much for the game to remain playable with this amount of players.

Figure 5.1: The delay between a spawn-request and -response.

Figure 5.2 shows the difference between the game-state of the server and the game-state of the bots within the view of a bot. The positions of all movables that are within the view of a bot are compared to the positions of these movables on the server, and result in an average offset of these movables.

In these results, the same trend can be seen as in the graph of the system responsiveness (Figure 5.1). The game-states are almost perfectly synchronized up to 40 bots, and until 100 bots the difference in game-states is very low (with an average difference of about 10 pixels). At 110 bots the situation suddenly changes a lot. With 218 pixels average offset, and peaks up to 413 pixels difference, the clients and server game-states become very desynchronized and therefore the game becomes unplayable.
Figure 5.2 shows the average amount of messages that the server sent and received.

Up to 100 bots the graph shows a clear exponential growth of the amount of messages being sent and received. Where we expect the exponential growth to continue at 110 bots, this does not happen. The stop in exponential growth suggests a bottleneck somewhere in the system. At 120 bots the amount of messages being sent and received is even less when compared to 110 bots, and therefore suggests that the bottleneck is not the maximum amount of messages that can be sent and received, but must be somewhere else in the system.
Figure 5.3: The amount of messages being sent and received by the server.

Figure 5.4 shows the average CPU-load, split into user- and system time.

The graph shows a similar curve as Figure 5.3 does. At first the graph keeps growing faster up to 100 bots, and grows only a little at 110 bots, indicating the same bottleneck as mentioned before. At 120 bots, the average CPU-load suddenly decreases.
Figure 5.20 shows the average bandwidth usage of the server. The average bandwidth usage of the server is expected to be very similar to the average amount of messages being sent and received. This is true indeed up to 110 bots, however at 120 bots, when the system becomes overloaded, the bandwidth usage graph does not collapse as much as the graph in Figure 5.3 showing the average amount of messages sent and received.
5.2.2 Conclusion

This section evaluates the scalability problems of the traditional client-server architecture (without the use of any AOI-filtering algorithm) in order to answer the first question in Section 1.4. The measurements show an exponential growth in messages, bandwidth-usage and CPU-load until some bottleneck shows up at 110 bots, and renders the game unplayable. At first sight, the bottleneck of the system is not clear since it could be both bandwidth and CPU. However, the bottleneck can be identified. The yet to be discussed graph in figure 5.9 shows that other AOI-filters can handle over 110 bots without hitting a CPU-bottleneck. Also, the graph in figure 5.20 shows that bandwidth does not increase or decrease at 120 bots. Based on these facts it can be concluded that the bottleneck is bandwidth. The decrease in CPU load can be explained by the bandwidth bottleneck. Because the network is unable to handle all the messages, some messages are never read from the clients, meaning the server has viewer messages to handle, and thus a lower CPU load. The main scalability problems of the traditional client-server architecture lays in the exponential growth of the bandwidth usage. However, CPU-load can too be a major issue.
5.3 The Impact of Different Types of Area-of-Interest Algorithms on System Performance

This section evaluates whether the use of an AOI-filtering algorithm is a valid approach for scaling a 2D game based on the traditional client-server architecture.

5.3.1 Results

Figure 5.6 shows the average spawn-delay of the three different types of filters: No-filter, Closest-filter and Threshold-filter. As noticed in Section 5.2.1, the server is not able to handle 110 concurrent bots. However, the average spawn-delays of the Closest- and Threshold-filters are still below 50 milliseconds, and therefore the game is responsive enough to remain playable.

![Figure 5.6: The delay between a spawn-request and -response.](image)

Figure 5.7 shows the difference between the game-state of the server and the game-state of the bots within the view of a bot. Again, at 110 bots the difference in game-states becomes too large when not using an AOI-filter. The threshold filter does not show any difference in game-states up to 130 bots, and only little difference at 150 bots.
Figure 5.7: The difference between the game-states of the client and server.

Figure 5.8 shows the average amount of messages being sent and received by the server. Here, a huge difference between the No-filter and Threshold-filter or Closest-filter can be seen. When the system becomes overloaded, the average amount of messages being sent and received collapses, which is clear at 130 bots with the No-filter, and at 150 bots with the Closest-filter. The threshold filter does not show such a decline.
Figure 5.8: The amount of messages being sent and received by the server.

In Figure 5.9 the average CPU-load is shown. When compared to the CPU-loads of the No-filter, the Closest- and Threshold filters require much less CPU cycles. However, the increase of the CPU-load of the two filters is larger than the increase in the amount of messages being sent and received.
Figure 5.10 shows a huge decrease in average bandwidth usage when an AOI-filter is used. At 110 bots, the average bandwidth usage of the No-filter is about 4.8 times greater than the Threshold-filter! The graph clearly shows, that the increase in usage is more linearly than exponential when using an AOI-filter.
5.3.2 Conclusion

The use of an AOI-filtering algorithm makes the MMOG scale better because response-times and Delta game-states only become too high at 130 bots for the Closest-filter, and even at 150 for the Threshold filter. When not using a filter, these values already become too high at 110 bots. The smaller increase of bandwidth usage and CPU-load when compared to the No-filter, makes a system that implements such a filter scale better. Since the CPU-usage seems to scale worse than the bandwidth-usage, the bottleneck for scaling shifts towards the CPU. The bottlenecks that show up at 130 bots for the Closest filter and at 150 bots for the Threshold-filter are probably caused by the fact that only a single thread can access the server game-state at a time.

5.4 The Impact of Different Types of AI on System Performance

The goal of this section is to evaluate the influence of different player behavior on the system. For this study, several mixes of the three bot-types described in Section 4.3.1 have been created. They are as follows:
Mix 1 100% Defensive bots
Mix 2 66% Defensive bots, 33% Offensive bots
Mix 3 66% Offensive bots, 33% Defensive bots
Mix 4 100% Offensive bots
Mix 5 66% Offensive bots, 33% King-of-the-hill bots
Mix 6 66% King-of-the-hill bots, 33% Offensive bots
Mix 7 100% King-of-the-hill

5.4.1 Results

Figure 5.12 shows the responsiveness of the system. As can be seen, on average all spawn-responses are being returned well within 50 milliseconds, and therefore the server is well provisioned.

![Graph showing responsiveness of the system](image)

Figure 5.11: The delay between a spawn-request and -response.

In Figure 5.12 the average deltas of the game-states are 0 for all bot-mixes with the No- and Threshold-filter. However, with the Closest-filter minor differences can be seen at bot-mix 3 and 7, indicating that on average more bots are within
the view of a bot than the Closest-filter allows to show in real-time. The testrun containing only King-of-the-hill type of bots shows the biggest difference, because these bots tend to cluster the most. At bot-mix 6, no difference can be seen, because the Offensive type of bots keep the King-of-the-hill type of bots from clustering.

![Figure 5.12: The difference between the game-states of the client and server.](image)

Figure 5.12 shows only slight variation in average amount of messages being sent and received by the server. Only some noteworthy differences can be seen at bot-mix 7. The decrease of the graph of the No-filter indicates that less shots were fired, and the increase at the Threshold-filter shows that clustering of players does mean an increase of amount of messages being sent and received for this filter.
Figure 5.13: The amount of messages being sent and received by the server.

In Figure 5.14 the CPU-load for the various bot-mixes is shown. The CPU was loaded only very little, and different player behaviour does not show major differences in CPU-load.
In Figure 5.15, similar behaviour as with the previous graphs can be spotted. There are no major differences when the player behaviour changes.
5.4.2 Conclusion

Player behaviour does influence the system, but only very little. Player behaviour does get important in cases where players cluster together, and try to follow the same target.

5.5 The Impact of the Size of the Field on System Performance

say: density does matter!

1: GOAL of this section 2: General info about where results are (figures data etc). Include text talking about how variation in results are handled: mean min max? Possibly include a extra table with those results. 3: comments about facts/findings. Statement of fact/finding = one liner. Support with graph/data: The Y being equal to X indicates that the finding is true. MAX 3 findings per section.

5.5.1 Results

Figure 5.16 shows the system response time. As can be seen, field size does not have a impact on system response time, at least in the scenario of 32 bots.
Figure 5.16: The delay between a spawn-request and -response.

Figure 5.17 shows the difference between the game-state of the server and the game-state of the bots within the view of a bot. Interestingly, closest-filter shows some deviations on low field sizes.
Figure 5.17: The difference between the game-states of the client and server.

Figure 5.18 shows the amount of messages sent from and to the server. It’s directly visible that performance of different AOI-filters is heavily impacted by field-size. Closest-filter proves to work most efficient when operating in a environment with a high density of players. However, in threshold-filter outperforms closest-filter on low player densities.
Figure 5.18: The amount of messages being sent and received by the server.

Figure 5.19 shows the server-side CPU load. The graph shows similar characteristics to figure 5.18.
Figure 5.19: Average server user and system CPU-load

Figure 5.20 shows the bandwidth usage of the server. Again, the graph shows similarities to figures 5.18 and 5.19.
5.5.2 Conclusion

From this section it can be concluded that field-size affects system performance greatly. Closest-filter proves to be able to deal with high player densities.
Chapter 6

Future Work

This section describes five directions for future work that we recommend. These five directions are: game architecture, Facebook integration, AOI filters, user tests, and metrics.

The game architecture is based on a server-client model, without any form of clustering. It’s possible to distribute server load among more server machines, to overcome bottlenecks found in Chapter 4. Trying to distribute load among multiple servers is a direction for future research.

The game already contains key Facebook integrations. However, there is room for even further integration. The game could be extended with RPG-like elements like the ability to collect items. These items then can be used to send gift to friends, giving the players an even more social experience. The game could also be extended with another social feature, which is being able to group/allies with your friends during battle. Doing more integration with Facebook is left for future work.

The thesis researched three different AOI-filters: No-Filter, Threshold-Filter, and Closest-Filter. These three purely decide to filter messages based on positions. More advanced filters could also integrate more of the game logic in the decision making, like prioritizing the ships that fly directly before the player above the players besides the player because it’s only possible to shoot forwards. A more in depth study on AOI-filters is left for future work.

The approach used for validation and testing was to use artificial bots. Different types of AI’s were used to mimic player behaviour. However, no tests were done using real players. A user test with real players is recommended to get more realistic statistics.

During tests, different metrics were measured. However there are other factors that could be measured for a more in depth analysis. One metric could be how well the system can deal with high package loss, and lag spikes. Doing a more in depth analysis with more metrics is left for future work.
Chapter 7

Conclusion

Both MMOGs and Facebook are popular. Though many MMOG architectures are proposed, interactive and scalable MMOGs are not seen on Facebook. To show interactive MMOGs are possible to run on Facebook, we built a game that is similar to mainstream games. The game is interactive and has various integration points with Facebook. For scalability, the game uses a scalability technique called AOI filters. The implementation is measured under various conditions to get a realistic view of the characteristics of the system. Finally, we have established five directions for future work: game architecture, Facebook integration, AOI filters, user tests, and metrics.
Part I

Appendices
Appendix A

Assignment Description
Students
Fedor Jutte, 1374141
fejuto@hotmail.com
Jerry de Swart, 1392050
j.deswart@student.tudelft.nl

Background & Main Goal
Massively Multiplayer Online Games (MMOGs) continue to grow in user numbers as well as in the size of the virtual worlds. This increase introduces many problems. A single machine is usually unable to handle the massive load, so an architecture is required to distribute the load over multiple machines. Many architectures have been proposed that use a server cluster, use peer-to-peer technologies, or use a hybrid between the two.

For the course IN3305 (Bachelorsemiarnium) the students made a survey of many existing MMOG architectures. The assignment is to use the knowledge and techniques acquired during the Bachelorsemiarnium to create a scalable MMOG that is integrated with Facebook. Facebook is a social networking website with over 350 million users and is ripe for more interactive online games.

The assignment
"Create a highly interactive game that is integrated with Facebook"

The game will be a 3rd person shooter similar to BZFlag (http://bzflag.org/). The game is required to have ways for players to join games & invite their friends from within FaceBook which will be a major achievement of your work. Since it is Facebook, the game is targeted at a grand audience, which in turn should lead to many players within the same session/map. The latter requires investigating scalability techniques from which Area-of-Interest-based techniques are what the students will focus on.

The students will evaluate different options for Area-of-Interest. For this purpose the students will make a test suite. To help producing test results students will also create various bots that are able to reproduce human-like play behavior, which can then be used by the test suite. Test results produced by the test suite will produce data relevant to the project.

Other

Accompanying teacher and mentor:
Dr. Alexandru Iosup
Parallel and Distributed Systems Group
Delft University of Technology.
A.iosup@tudelft.nl

Dates:
The project will run from 19th of April through 11th of June. (8 weeks)

Place:
Delft University of Technology.
Appendix B

Planning Document
The Design & Implementation of a Facebook-based interactive MMOG

Planning Document
Bachelorthesis (IN3405)

Fedor Jutte
1374141

Jerry de Swart
1392050

23 April 2010,
Delft University of Technology

1 Introduction
Massively Multiplayer Online Games (MMOGs) continue to grow in user numbers as well as in the size of the virtual worlds. This increase introduces many problems. A central server is usually unable to handle the massive load with the traditional client-server architecture, so a different approach is required to distribute the load over multiple machines or decrease the total load in some other way. Alternative architectures have been proposed that use a server cluster, use peer-to-peer technologies, or use a hybrid between the two. With the recent introduction of game portals on social networking websites such as Facebook and Hyves, thousands of single-player games with very little interaction between friends have been deployed. These websites now are ripe for more interactive online games. For this bachelor thesis a simple MMOG for Facebook based on the traditional client-server architecture will be developed. For this MMOG, the use of several area of interest (AOI) algorithms will be researched in order to improve scalability.

The development and research will be conducted by a team of two students consisting of Fedor Jutte and Jerry de Swart during 11 weeks. The project will be under the supervision of dr. Alexandru Iosup who is the mentor and creator of the assignment. The research of several AOI algorithms is based on previous work for the course bachelor-seminarium (IN3305). In this course, a survey of existing MMOG architectures was produced.

The remainder of this document is structured as follows. In section 2 we provide the project description setting the goals of the project and defining the scope and requirements of the project. In section 3 the approach is being discussed, answering the question of how the given goals will be achieved. In section 4 we provide a brief overview of the allocation of resources and section 5 provides the planning of the activities. Section 6 discusses the quality assurance of the delivered products and section 7 contains the attachments of this document.

2 Project description
In this chapter the goal of the project will be discussed. Furthermore the scope and requirements of the project will be defined to avoid misunderstandings between the mentor and interns.
2.1 Context
The project is a bachelor thesis conducted by students of the Delft University of Technology which involves a strong element of research. In order for the results and products to be usable by other researchers, the deliverables will be open source and licensed accordingly.

2.2 Motivation
With a game portal, social networking websites provide an easy way for millions of people to play with or versus their friends. However, the majority of games is currently single player and make minimal use of the multiplayer possibilities provided by the websites. In order to create an online realtime game that can be simultaneously played by a few hundred players per server, scalability becomes a major problem. The result of this thesis shows the possibility of an MMOG on a social networking website.

2.3 Main Goal
The goal of the project is to create a highly interactive game that is integrated with Facebook. The game will be a 3rd person shooter similar to BZFlag (http://bzflag.org/). The game is required to have ways for players to join games and invite their friends from within FaceBook, which will be a major achievement of the project. Since it is Facebook, the game is targeted at a grand audience, which in turn should lead to many players within the same session/map. The latter requires investigating scalability techniques from which area of interest-based techniques are what the students will focus on.

The students will evaluate different options for area of interest. For this purpose the students will make a test suite. To help producing test results students will also create various bots that are able to reproduce human-like play behavior, which can then be used by the test suite. Test results produced by the test suite will produce data relevant to the project.

2.4 Deliverables
During the project, several deliverables will be created of which three are of main importance. These are the final implementation of the server and game on Facebook, testing results (including play-testing with humans), and the final report. The final report shows the results of the overall project and includes technical documentation such as system- and test-design as well.

Other deliverables include the test-suite (in order to measure server-performance) and early prototypes of the server and game (to demonstrate the technical possibilities).

2.5 Requirements and Restrictions
The final implementation is capable of handling atleast 100 simultaneous players.

The game will be a 3rd person shooter, and will at least include a 2D map, where avatars (controlled by players) can freely move on. Furthermore, for the game element, each avatar is armed with a fire-weapon with which he/she can shoot and eventually kill other avatars. The main purpose of the game is to demonstrate the possibility of creating a realtime MMOG on Facebook. The game will be simple because of its purpose, therefore no fancy graphics or extensive GUI will be included. The following game specific targets are optional and no requirements:

- Ingame chat
The game will be playable directly on the Facebook website, and not link to some other, external, webpage. The game is also required to provide some mechanic for players to invite each other into the game.

Atleast 3 different approaches for Area of Interest will be implemented and tested using bots. Testing results will include various visual elements such as graphs and tables. Conclusions for the test results should include information about maximum load for each Area of Interest approach.

Since the focus of the project lays on scalability in terms of having many players within the same session/map, the server will not include techniques such as zoning and sharding.

Other than some very basic security, the subject of security is outside the scope of this project.

3 Approach

This section discusses the activities in order to reach the goals and meet the requirements as stated in section 2.

3.1 Facebook

Facebook is a social networking website with over 350 million users. The website contains a rather new game portal with thousands of simple browser based games. Game developers are allowed to create a page for their game where Facebook users can play it.

With this gaming portal, Facebook provides a convenient way of deploying and testing the game created by our product “in the wild” with real people. The game will connect to a game-server (apart from Facebook) and the game itself will be hosted on a separate fileserver.

3.2 Facebook API

The game portal on Facebook differs from a regular game portal. The games on Facebook can take a look into the Facebook-account a player is logged in with, and use this information. This way, for instance makes it possible to see the friend list of the current player. The communication between Facebook and a game is defined in the Facebook API (Application Programming Interface), and this API will be used in order to have ways for players to join the game and invite their friends from within FaceBook.

3.3 Flash applications

The game itself will be created in Adobe's Flash. Flash is a commonly used tool for developing browser based games. Flash is capable to connect to a custom server outside of the Facebook website, which enables it to be used for creating an online multiplayer game.

3.4 Scalability

To make the game playable for 100 simultaneously connected players, the server needs some form of scalability. Scalability is one of the key aspects of the project that need to succeed in order for the
whole project to be a success. Several options for area of interest (AOI) filtering will be evaluated and demonstrated for better scalability. The performance evaluation will be done by repeating the experiments for different area of interest algorithms in the same environment setting. In order to compare the results, they will be presented. Test results include min- and max values, mean, standard deviation and quartiles.

3.5 Risks

There are two crucial factors concerning this project. The first is integration of the game with Facebook, to make it accessible for many people to play. The second crucial factor is scalability of the server to allow more simultaneous active players in comparison to the traditional client-server architecture. Failing any of these two crucial factors means the project will fail. To address this risk, we will allocate extra time for these crucial activities.

The project involves some additional risks. Testing results might show very few differences between different AOI implementations. In this case the results won't show information for good comparison between the AOI implementations, though will show that in our used scenario no specific implementation is preferred over another.

4 Resources & Legal

4.1 Costs

Required commercial software:
2x Adobe Creative Suite Web Premium (student edition): 2 x €350 = €700
   – Adobe Flash Professional
   – Adobe Catalyst
   – Flash Builder
   – Photoshop
   – Illustrator

Required hardware:
2x mainstream dedicated servers (3 months) 2 x €230 = €460
   – File server
   – Game server

4.2 Funding

Students will buy their own Adobe Creative Suite Web Premium (student edition).
The servers will be sponsored by TU-Delft.

4.3 Licensing and ownership

All deliverables will be licensed under the Creative Commons (Attribution-Noncommercial-Share Alike 3.0 Netherlands). Copyright and license holders are the interns Fedor Jutte and Jerry de Swart.
5 Planning

In this chapter a planning for the project is presented. In 5.1 enumerates over the tasks (or substeps) required for successful completion of the project. In 5.2 the tasks are putten in a Pert graph, which shows task dependencies. Finally, in 5.3 a Gantt chart is presented showing the actual planning.

5.1 Tasks

In this paragraph, a breakdown of the project into tasks is presented. The following list is an enumeration (in chronological order) of these tasks. Some tasks are divided into sub-tasks. Each task in the enumeration also shows a time-estimation in man-weeks required to complete a task. Some tasks are considered milestone tasks, and denoted as such. As mentioned in chapter 1, the team consists of two people working in full-time, and the project has an duration of up to 11 weeks. That means up to 22 man-weeks of work is available to complete tasks. The sum of the estimated time-estimations in the enumeration is 21 man weeks.

1. Planning document (2 man weeks)
2. Background on Flash Apps (2 man weeks)
   1. research API
3. Background on Facebook API (1 man weeks)
4. Client + Server (basic) (4 man weeks) Milestone
   1. docs: architecture, message protocol
   2. implementation
5. Client + Server (basic) + Facebook (2 man weeks) Milestone
   1. implementation on Facebook API
   2. testing
6. Client + Server (basic) + bots (1 man weeks)
   1. design bot API
   2. implement 2-3 different AIs
7. Testing (Basic) + bots (2 man weeks)
   1. docs: test plan, testing architecture
   2. implementation
   3. test: produce graphs and tables
8. Client + Server (Area of Interest) + bots (2 man weeks)
   1. docs: architecture, extend message protocol
   2. implementation
   3. test: produce graphs and tables
9. Client + Server (Area of Interest) + bots + Facebook (1 man weeks) Milestone
   1. implementation
   2. test with friends
10. Final report (2 man weeks)
    1. draft
    2. final report
11. Polish (optional)
    1. more tests, debugging, friends/user survey
12. Final presentation (2 man weeks) Milestone
    1. preparation
    2. the actual presentation
5.2 Pert diagram

In this graph, dependencies between tasks are presented. The graph is read as follows: Each node corresponds to a task. An arrow from node A to node B means task B can only start when task A is completed. The number on each arrow corresponds to the man-weeks needed to complete the task the arrow is pointing to. Task 4, 5, 9, and 12 are milestones.
5.3 Gantt chart

In this table the actual planning is presented. The table is read as follows: Each row is a task, and each column corresponds to a week of the project. If there's a “F” in a cell, Fedor will work on that task in that week. Likewise, when there is a “J” in a cell, Jerry will work on that task. A star (“*”) in a cell means the project is expected to reach a milestone at the end of that week.

<table>
<thead>
<tr>
<th>tasks</th>
<th>weeks</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
</tr>
<tr>
<td>7</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
</tr>
<tr>
<td>8</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
</tr>
<tr>
<td>9</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
</tr>
<tr>
<td>10</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
</tr>
<tr>
<td>11</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
</tr>
<tr>
<td>12</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
<td>J</td>
</tr>
</tbody>
</table>

* = Milestone
6 Quality assurance

In this section we give some insight into which measures are taken in order to ensure that the deliverables produced meet the mentor's needs and comply to the requirements stated in section 2.5.

6.1 Product quality

To ensure the product meets the requirements, various tests will be performed on the product formulated in section 2.5. Included tests are code validation tests, various load testing using bots, stress tests, and play-testing by people.

6.2 Process quality

The project will use an agile software development methodology. Agile development means that the project will go through different, incremental, iterations where in each iteration the team will go through a full software development cycle including planning, requirements analysis, design, coding, unit testing and acceptance testing when a working product is demonstrated to stakeholders.

Because of the short iterative steps, documentation will be kept to a minimum to increase productivity.

Since an iterative development method is used for this project, the product will evolve gradually starting from an initial concept. This makes it possible for the mentor to be tightly involved with the evolvement process, by being able to give feedback. The development team and mentor will have contact at least once a week, where the process and progress is discussed.
Attachments

- A Survey of MMOG architectures
Appendix C

Project Progress Card
The Design & Implementation of a Facebook-based Interactive MMOG

Project progress Card
Bachelor thesis (IN3405)

Fedor Jutte
1374141

Jerry de Swart
1392050

Delft University of Technology

Progress

Week 1
Deliverable: Planning Document
This week setup activities of the project and a lot of preliminary research have been done, and most of the conclusions were put into the planning document. BlazeDS has been investigated to use as a base for the server.

Week 2
Deliverable: Working single-player flash-game showing the basics of the game.
In week 2 the first basic flash-game has been made and a lot more preliminary research has been done.
For the server, the use of SSH-keychains and SCP have been evaluated in order to set up automated tests, and some initial design was already started (for week 3). The metrics to measure the performance of various AOI-filtering algorithms and test-setups have been determined and will be evaluated with the mentor in week 3 since the meeting on Friday didn't take place.
This week, also the initial setup of the final report has been made and GNUPlot has been evaluated in order to create graphs for reporting the results later on.

Week 3 and 4
Deliverable milestone: Working basic client-server: gameclients are able to connect to the server and play the basic game versus each other.
Deliverable: The server architecture and message protocol documentation.
Week 3 started with more research. Since the “gameworld” is already written in Flash, having the server simulate this world in another language would mean implementing it twice. Therefore, the option of using ActionScript3 (the language used for Flash) for developing the server has been evaluated. Unfortunately, we had to conclude there was no convenient and easy way to realize this, and therefore will develop the server in Java.
After having investigated BZFlag (before the project started), we did another quick search for a
realtime gameserver this week. Having totally overlooked xpilot in our previous searches, we found that it actually seemed a very useable solution in the first place. However, we soon discovered that the xpilot server would have to be ported to use TCP instead of UDP, and we found implementing the entire xpilot protocol (with a lot of features that exceed our main goals) too risky. Also, the use of a non-object oriented language (on the server) adds up to the risk, and we have therefore decided not to use xpilot and choose the safer option.

In order to get a better overview of what would be needed later on in the project, the integration with facebook has been investigated. Hereby a simple application has been made (on Facebook).

This week, also the first design for the server was made.

The start of week 4 consisted mostly of agile developing the server: continuous re-evaluating the design and implementing the parts. However, even after some initial design in week 3, most of week 4 went into the design too. There was a lot of complexity to the server. A subset of questions we had to answer in our design are: How would the game-state be constructed, how do game-state updates work, how are clients updated about these updates, how do we serialize updates, how are we going to do AOI-filtering, how are we going to do collision checking, how does the server read messages from the client, how does a player control his ship on the server, how is game-state presented client-side, what is the startup/initialization sequence, how do we keep room for performance checking later on when we enter the testing phase.

After quite a lot of time well spent, the design feels very decent and the “flow” of messages and events is now very clear. The logging “system” has been implemented and Unit-tested. It has its own thread which runs at a lower priority, so writing a log-event is a matter of putting the log-event in a queue and never worry about it again.

This week, we also got a server assigned to us. The server is a brand-new dual-quad core 32GB ram headless machine, with which we are obviously very pleased. With the server, some initial scripts could be made in order to automatically deploy and run the server-software on the machine.

Because more time was spent on designing and implementing the server than planned, the deliverables couldn't be made in time and work continues in week 5.

**Week 5**

**Deliverable:** 2-3 implemented bot AIs.

We weren't able to meet the milestones for week 4 because extensive time was needed for the design, so week 5 started as a continuation of week 4. Visible from the design, there is a lot of code repetition. Each Game-method call creates and performs an Action, returns a Result, pushes that result through a (for now) default all-pass filter, create the corresponding message and serializes that message. Therefore we decided to use a code generator, Apache's Velocity. We created templates that generalize the many repetitions, such as for the sequence just described, but also for generating enumerations, interfaces and message handlers.

At the end of week 5 the server was in a working state, and we had a working head-less bot. The bots would fight each other to the death trough the console.

The flash-client implementation started in week 5, but didn't come further than the initial stages.
**Week 6 and 7**

**Deliverables:** Flash-client implemented on Facebook.

**Deliverable milestones:** Graphs and tables of the first test-results without AOI-filtering algorithm.

These weeks, work on the server and flash-client continued. The deliverables of week 5 (2-3 implemented bot Ais) were made and demonstrated to the mentor.

**Week 8**

**Deliverables:** Implementation of the different AOI-filters.

Week 8 is simply an extension of week 7. The original planning now really started to fade, as deliverables from last week(s) were not yet (fully) finished. We were confident that all would still go well in the end, but there would be little time for problems and setbacks. This week, the first writings for the report took place.

**Week 9**

**Deliverables:** Integration with Facebook.

For this week originally the integration with Facebook was planned. However, as noted before, work on the integration had already started in week 6. This gave us the opportunity to work on the deliverables from last week(s). Also work on the test-suite started this week. As can be seen in the planning, this test-suite should already be in use since week 7, but no time was planned for working on it. This ment extra pressure and working on the project in the evenings and weekends now became rule rather than exception. The plan now was to first finish all needed deliverables (server, test-suite, AOI-filtering algorithms), and then run all tests at once at the end. Unfortunately, there is not enough time to do a full usertest.

**Week 10**

**Deliverables:** Final report

Originally, writing the final report was scheduled for this week. However, the final presentation was now scheduled at July 8th (week 12), and the final report to week 11. Therefore some more time came available to work on everything needed to create the gaphs for the final report. This week the 2 extra AOI-filtering algorithms were created, and the test-suite was finished. This Thursday we finally were able to start testing, after a lot of startup-problems with the test-suite. At the end of this week, the first real graphs were produced.

**Week 11**

**Deliverables:** Final presentation

The final presentation is rescheduled to week 12, and this week is all about writing the Final report. Unfortunately, we found out that there was no way to determine the Gamedelta within the view of a player (bot), because we didn't record the position of the player that was taking the snapshot. Also we found that the first measurements of the responsiveness of the system were influenced by the fact that not all bots had yet started. Therefore we quickly rebuilt (and tested) the gamestate-snapshotting parts and built a delay of a few seconds into the measurement-system responsible for measuring responsiveness of the system. While the report has to be delivered at Thursday, we found ourselves rerunning the test-suite at Monday. Fortunately, the results were very promising and therefore our time was well spent.
Week 12

This week is planned for finishing the presentation. The date of the presentation is July 8th.
Planning and Milestones

1. Planning document (2 man weeks)
2. Background on Flash Apps (2 man weeks)
   1. research API
3. Background on Facebook API (1 man weeks)
4. Client + Server (basic) (4 man weeks) Milestone
   1. docs: architecture, message protocol
   2. implementation
5. Client + Server (basic) + Facebook (2 man weeks) Milestone
   1. implementation on Facebook API
   2. testing
6. Client + Server (basic) + bots (1 man weeks)
   1. design bot API
   2. implement 2-3 different AIs
7. Testing (Basic) + bots (2 man weeks)
   1. docs: test plan, testing architecture
   2. implementation
   3. test: produce graphs and tables
8. Client + Server (Area of Interest) + bots (2 man weeks)
   1. docs: architecture, extend message protocol
   2. implementation
   3. test: produce graphs and tables
9. Client + Server (Area of Interest) + bots + Facebook (1 man weeks) Milestone
   1. implementation
   2. test with friends
10. Final report (2 man weeks)
    1. draft
    2. final report
11. Polish (optional)
    1. more tests, debugging, friends/user survey
12. Final presentation (2 man weeks) Milestone
    1. preparation
    2. the actual presentation

Gannt Chart of the Planning

<table>
<thead>
<tr>
<th>tasks</th>
<th>weeks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1 F+J</td>
<td></td>
</tr>
<tr>
<td>2 F+J</td>
<td></td>
</tr>
<tr>
<td>3 F</td>
<td></td>
</tr>
<tr>
<td>4 F+J F+J*</td>
<td></td>
</tr>
<tr>
<td>5 F</td>
<td></td>
</tr>
<tr>
<td>6 J</td>
<td></td>
</tr>
<tr>
<td>7 J</td>
<td></td>
</tr>
<tr>
<td>8 F+J</td>
<td></td>
</tr>
<tr>
<td>9 F+J*</td>
<td></td>
</tr>
<tr>
<td>10 F+J F+J F+J</td>
<td></td>
</tr>
<tr>
<td>11 F+J F+J F+J</td>
<td></td>
</tr>
<tr>
<td>12 F+J F+J*</td>
<td></td>
</tr>
</tbody>
</table>

* = Milestone
Gantt Chart of the Process

Since a lot of changes were made to the schedule and to the deliverables after they were (almost) finished, it is hard to say exactly when the work on certain tasks started and finished. The following chart shows the finishing of the tasks as hard deadlines, but some minor work was still done later.

<table>
<thead>
<tr>
<th>tasks</th>
<th>weeks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 9 10 11 12</td>
</tr>
<tr>
<td>1</td>
<td>F+J</td>
</tr>
<tr>
<td>2</td>
<td>F+J</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F+J</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>J</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>F+J</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>F+J</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

* = Milestone
Appendix D

A Survey of MMOG System Architectures
Abstract

Massively Multiplayer Online Games (MMOGs) continue to grow in user numbers as well as in the size of the virtual worlds. This growth introduces challenges in many directions, such as scalability, responsiveness, and security. A single machine is usually unable to handle the massive load to host a MMOG, so an architecture is required to distribute the load over multiple machines. Many architectures have been proposed that are based on server clusters, peer-to-peer networks, or a hybrid of these two; however there exists no extensive survey of these architectures. In this paper, we present a survey of the various existing MMOG system architectures. First, we introduce a taxonomy of important MMOG design topics. Second, we map existing system architectures to this taxonomy. Third, we suggest directions for future work.

1 Introduction

Massively Multiplayer Online Games (MMOGs) have been a huge commercial success in recent years. Today, millions of people meet and play in virtual worlds with friends and strangers. To reduce costs while still serving the demands of this large-scale population, game developers use one of the many existing MMOG architectures [Iimura et al., 2004] [Bharambe et al., 2006] [Bharambe et al., 2008]. There currently exists no comprehensive survey of MMOG system architectures, which makes the choice of architecture difficult. Previous work has identified many requirements for MMOG architectures [Schiele et al., 2007], but no extensive comparison between different systems has been performed. To address this problem, in this work we introduce a taxonomy of MMOG design topics, and use this taxonomy to compare over a dozen selected MMOG architectures.

A successful example of an MMOG is Blizzards World of Warcraft. The game has over 11 million players globally, and approximately 1 million players online at any one time [WOW Press, 2008]. With so many players, it is readily apparent that MMOGs require a significant amount of computing and networking resources. Regardless of its design, an MMOG architecture should keep the game consistent between multiple users, should scale well in order to serve a high number of players and game objects, should be resilient against hackers and cheaters, should be responsive in order to provide good gaming experience to the players, and, finally, should be resilient to failures to guarantee a high uptime.

Many MMOGs use a client-server architecture, in which a central server maintains the game state. Scalability is achieved by sharding: splitting the global game state in smaller independent chunks, called shards, which are then distributed over a server cluster. This architecture results in a fragmented gaming community with no guarantees on server or network performance, and often leads to poor gameplay experience. Although this architecture scales well with the number of players, it lacks flexibility and the servers have to be over-provisioned to handle peak loads.

A recent development to MMOG architecture design is to apply peer-to-peer networks in MMOGs. Peer-to-peer networking has so far been used to harness idle storage and network bandwidth, including storage systems, content distribution, and instant messaging. Peer-to-peer
Architectures in MMOGs utilize memory and CPU cycles of players to maintain the shared game state. This architecture opens up possibilities to make games scalable without fragmenting the player base. However, peer-to-peer architectures introduce new security issues and other concerns to MMOG architecture design.

In this work our contribution is threefold:

1. We introduce a taxonomy of MMOG design topics in Section 2,
2. We map existing system architectures to this taxonomy in Section 3, and
3. We suggest directions for future work in Section 4.

2 A Taxonomy of MMOG Design Topics

A taxonomy is required to compare different MMOG architectures. The taxonomy we propose for this purpose consists of security, responsiveness, reliability, scalability, and resource ownership. Each of these directions will be elaborated in this section.

2.1 Security

In most multiplayer games security (cheat prevention) is an ongoing issue, and this becomes even more important for MMOG architectures that potentially outsource part of the game state to the clients. This paper only considers in-game security: the ability of individual players to manipulate the game outside of the rules and gain an unfair advantage. We do not consider the problem of hackers attacking the system in order to deliberately shut down part of the system.

2.2 Responsiveness

Because MMOGs support many simultaneously playing players it is unlikely that the full game state can be sent to every player every frame. Players however do not want other players to appear to be teleporting from one place to the other because state updates are sent infrequently. Responsiveness considers the techniques that architectures employ to prevent this kind of teleporting. We have selected three commonly used techniques: prefetching, message forwarding, and entity count limiting. Prefetching tries to predict what objects players will be interested in in the near future, sending object updates right before the player actually needs them. Message forwarding improves responsiveness by allowing players with low upload capacity to forward messages through players with higher upload capacity in order to deliver more updates on time. Finally, player count limiting simply avoids excessive load by limiting the amount of players within a specific part of the world.

2.3 Reliability

A reliable MMOG system architecture is resilient to failures of some of its components. Architectures differ greatly in the amount of components that are allowed to fail and what is done to avoid the complete system to go down. In order to compare systems on reliability, we only consider whether a design contains a single point of failure. An architecture has a single point of failure iff at least one component exists that interrupts the game if only that component fails.

2.4 Scalability

Different architectures use different techniques to allow many players to simultaneously play a MMOG, among these techniques are interest set filtering, sharding, and zoning. By interest set filtering we mean that peers only receives updates (or receives updates more frequently) for entities they are probably interested in. By exploiting a player’s interests, the bandwidth requirements for updates can be greatly reduced. Another technique to support more players to play at the same
time is to divide the game world into zones and having different servers manage different zones. Zones are distributed over servers so that servers only need to communicate with players within their own zone(s). The last technique for scalability is sharding: running multiple copies of the (same part of the) game world without interaction. Each shard is maintained by one server, which is scalable because each server only needs to support a subset of the active players.

2.5 Resource Ownership

Because purchasing and maintaining (or renting) servers costs a lot of money, some designs have devised a way to allow players to host part of the MMOG. Allowing players to host part of the game greatly reduces hosting costs, but it also jeopardizes security. In order to keep the game secure, a middle path can be chosen by using (trusted) community hosted servers which avoids outsourcing control completely to individual players. There are therefore three features to be considered: player owned resources, community owned resources and publisher owned resources.

3 A Survey of MMOG Systems and Architectures

In this section we start with an overview of existing MMOG system architectures: both architecture proposals and architectures that are used in deployed systems. The system architectures are mapped onto the taxonomy that was introduced in Section 2. The overview is followed by a brief description of each system that is considered in the overview.

3.1 An Overview of MMOG System Architectures

To compare different MMOG system architectures, we mapped each architecture’s features to the taxonomy; Table 3.1 shows an overview of this mapping, considering MMOG system architectures sorted by year of publication. In this section we will describe our three most important observations made from this table.

First we observe that security is underrepresented. Only three architectures implement cheat prevention or detection techniques, and those generally perform less well in responsiveness by applying entity count limiting. This observation will be further discussed in Section 4. Second we observe that the EVE Online architecture is the only architecture that implements the combination of reliability mechanisms and publisher owned resources. This is remarkable because EVE Online is also the only architecture in our overview that is implemented by the industry. It seems that is is possible to achieve reliability in architectures similar to the one EVE Online uses, but the techniques to do so are generally not mentioned in papers describing these architectures. Third we observe that sharding has become less popular since 2006. More recent MMOG system architectures use other techniques to achieve scalability, most likely to improve gameplay experience.

The remainder of this section will briefly describe each system we considered in the overview.

3.2 DIVE

In DIVE [Frécon & Stenius, 1998], the game world is considered to be a hierarchical database of entities. An entity persists as long as at least one application is running that interacts with the world. When no more applications are running, the world state is lost. DIVE solves this problem by running a monitor process that periodically saves the state of the world, which can be restored when the system needs to be restarted. DIVE offers tools for an application-level virtual multicast backbone, and it is a hybrid architecture that can both be seen as a client-server architecture and peer-to-peer architecture.
Table 1: Comparison of MMOG system architectures based on the taxonomy introduced in Section 2. Each row represents a single MMOG system architecture. Each column represents a single feature from the taxonomy. Architectures implementing that feature are marked with an ‘x’ in the corresponding cell.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Security</th>
<th>Responsiveness</th>
<th>Reliability</th>
<th>Scalability</th>
<th>Resource ownership</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cheat prevention/detection</td>
<td>Prefetching</td>
<td>Message forwarding</td>
<td>Entity count limiting</td>
<td>No single point of failure</td>
</tr>
<tr>
<td>DIVE (1998)</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mercury (2002)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MiMaze (2002)</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grid (2003)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVE Online (2003)</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zoned model (2004)</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEO (2004)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peer-to-peer (2004)</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM (2006)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IBM Grid (2006)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VON (2006)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Colysce (2006)</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSMMO (2006)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hydra (2007)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Zonal MMOGs (2007)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hybrid (2008)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Donnybrook (2008)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.3 Mercury

Mercury [Bharambe et al., 2002] is a distributed architecture that utilizes a content-based Publish-Subscribe system to notify peers about state changes. For each object in the game world, a set of selection attributes is defined, such as its x, y and z-coordinates. The total range for an attribute value is split over multiple peers, where each peer is responsible for a specific range of attribute values. Other peers can then add a subscription by supplying constraints on these attribute values, and these subscriptions are routed on all peers that are responsible for a matching range of attribute values. When the state of an object changes, a publication message is sent. Mercury routes this message to all responsible peers. These peers will have stored all matching subscriptions, so Mercury can now send notifications about the state change to all corresponding subscribers.

3.4 MiMaze

MiMaze [Gautier & Diot, 1998] is a fully decentralized game based on a multicast communication system. None of the peers keep track of the global game state; instead, each peer computes its own local view of the game world. Messages sent by each peer are synchronized using the bucket synchronization algorithm and a globally synchronized clock: state updates are not interpreted directly as they are received, but delayed so that they can be interpreted at the same time on all peers, avoiding inconsistency between each players local view.

3.5 A Service Platform for Online Games

The architecture proposed by [Saha et al., 2003] is based on a shared, on-demand service platform for hosting online games based on grid technology. A standards-based grid infrastructure promises to provide economies of scale and high availability. The architecture is initially geared toward fast-paced First Person Shooter (FPS) games, but several of the gaming services are also applicable to MMORPGs.

3.6 EVE Online

Unlike the other architectures, the EVE Online architecture [EVE Online, 2003] is actually implemented in the industry. This architecture consists of a central SQL-database server cluster, Solar System (SOL) servers, and proxy servers. These components and their interconnections are shown in Figure 1. The SOL servers are each responsible for certain aspects of the game world and communicate directly with the database server cluster to store and retrieve persistent data.
The players connect to the proxy servers, which keep track of where players are and what they are doing. The proxy servers communicate with the SOL servers about actions that a player performs. When a player performs an action, the proxy server informs only the affected SOL servers.

### 3.7 A Zoned Federation Model for Peer-to-Peer MMOGs

Iimura et al. propose a zoned federation model [Iimura et al., 2004] to employ a peer-to-peer architecture in MMOGs. The model introduces a zoning layer between the virtual world simulation and the peer-to-peer network substrate as shown in Figure 2. The architecture uses a so-called Distributed Hash Table (DHT) to construct the peer-to-peer network, and distributes zones over the DHT. A zone is a part of the game world, hosted by a single peer, called the zone owner. If a peer enters a zone that doesn’t have the address of a zone owner, the peer will subscribe itself to the zone as the new owner. When a zone owner leaves a zone, the zone ownership is transferred to one of the remaining peers in the zone; this peer will then become the authoritative server for that zone. By dividing the game world into several zones, workload can be distributed over a federation of peers. This leads to a scalable MMOG with minimal costs for the game publisher.

### 3.8 NEO

The New-Event Ordering (NEO) architecture [GauthierDickey et al., 2004] is a peer-to-peer architecture, which mainly focuses on cheat prevention. NEO divides game time into time intervals, called rounds. Each peer sends encrypted state changes to every other peer at the start of every round, but only sends the corresponding decryption key at the time the round ends. This ensures that peers cannot send updates for a round based on events that had not occurred at the time the round started. Consistency is ensured by using a majority-voting algorithm: an update message for a round will only be processed if the majority of the peers receive it in time, that is before the end of the round.

### 3.9 The First Peer-to-Peer MMOG

Knutsson et al. [Knutsson et al., 2004] propose the use of peer-to-peer overlays to distribute the time-varying game state of MMOGs. It distinguishes three elements in an MMOG world: avatars, immutable terrain, and mutable objects. The idea is to first divide the world into zones. Peers manage their own avatar, multicasting updates to all the other peers in the zone. Objects get assigned a unique coordinator-peer, which manages the object’s state and multicasts changes to all other peers in the zone. To prevent data loss caused by quitting peers, the algorithm tries to keep alive at least one replica for every object, under all circumstances.

![Figure 2: Zoned federation model](image)
3.10 An On Demand Platform for Online Games (IBM)

The architecture described by Shaikh et al. focuses on on-demand resource allocation for online games. In this architecture [Shaikh et al., 2006] a server pool is used to spawn additional servers during periods of high system load and to shut down servers during periods of low system load. The individual servers work much like in other games: they each provide a separate game world with different players. Because of that, little inter-server communication is necessary.

3.11 IBM Grid

By enhancing online single-server games for running on a grid system, existing games can be changed into MMOGs [Deen et al., 2006]. The game world is run on multiple servers, each serving one or more regions of the world. This architecture lets servers communicate via multiple 'whiteboards' instead of passing messages to each other. Each server only reads and writes to whiteboards relevant to the regions ran by that particular server. This way the whiteboards reduce the amount of traffic between the servers. Additionally, latency is reduced by allowing servers to act asynchronously from each other.

3.12 VON

This peer-to-peer architecture [Hu et al., 2006] uses a players area of interest to reduce bandwidth usage. Every player is considered to be a node in a Voronoi diagram. Each peer maintains the set of peers within its own area-of-interest. Direct connections are formed to the peers within the area-of-interest and to directly connected neighbors in the Voronoi diagram. A peer informs a neighboring peer when an entity may be entering that neighbor's area of interest; this may lead to updates to the Voronoi network and to additional connections being formed.

3.13 Colyseus

Colyseus [Bharambe et al., 2006] is a distributed MMOG architecture. Each game object in Colyseus (such as an NPC) is allocated to exactly one peer that is authoritatively responsible for the objects state and for applying control logic to the object (such as moving NPCs around). These objects can be replicated across several peers. A distributed hash table is used to find objects that peers may be interested in. Objects within the hash table can be found by specifying constraints, for example that their coordinates must be between 10 and 50 units.

3.14 A Public Server MMOG Architecture

Research by Chambers et al. examines the application of the public server architecture to an MMO in order to create a Public Server MMO (PSMMO) [Chambers et al., 2006]. There are three main architectures that can be used to host on-line games: peer-to-peer, client-server, and public server. The only difference between a client-server architecture and a public server architecture is in the provisioning of resources and content. The servers in a client-server architecture are maintained by the game publisher, whereas in a PSMMO the community takes care of the servers; this can minimize hosting costs. Additionally, mods can be created by the community in a PSMMO which reduces the need for publisher content generation.

In order to keep the system secure, an architecture as shown in Figure 3 is used. This figure shows that the publicly hosted servers are not authorized to make all decisions: authentication, billing and the distribution of loot (any persistent advancement a character can achieve [Chambers et al., 2006]) are taken care of by servers hosted by the publisher of the game, which makes it much harder for clients or publicly hosted servers to tamper with the data.
3.15 Hydra

Hydra [Chan et al., 2007] is middleware that converts a client-server MMOG architecture into a peer-to-peer architecture transparently to the developer. Since the client-server model is well-understood by game developers and works well, Hydra can be used without forcing developers to think differently when developing their games. Hydra guarantees state consistency when peers fail, meaning no data will be lost when peers fail. The architecture only shows a small message overhead and is therefore scalable.

3.16 A Zone Based MMOG Architecture

Ahmed et al. [Ahmed et al., 2007] describe two methods for improving the gaming experience in zonal MMOGs. First, it describes how zones using Application Level Multicasting can be made more efficient by creating multiple multicast channels for slow, normal, fast and very fast objects. This ensures that faster objects that are entering and leaving a zone frequently do not affect slower objects, therefore reducing the frequency of structural changes to the zone for slower objects. Second, it proposes a technique to effectively overlap zones. If an object is located on a zone boundary, the objects interest in all objects of the adjacent zones is computed. The zone with the highest total interest is considered to be the zone the player resides in.

3.17 A Hybrid Architecture for Massively Multiplayer Online Games

Jardine and Zappala describe a hybrid between a peer-to-peer and a client-server architecture [Jardine & Zappala, 2008], that makes a distinction between positional messages and state changing messages. Because positional messages are sent frequently, a lot of server bandwidth can be saved by allowing peers to handle these, while security can be maintained by handling state changing messages on the server. This architecture therefore splits the game world into regions. For each region, a single peer is selected to be responsible for broadcasting positional messages to peers within the same region and for forwarding messages sent by the server.

3.18 Donnybrook

Donnybrook [Bharambe et al., 2008] is a system for peer to peer MMOGs that require low response time to player requests. In order to avoid sending avatar state updates to every peer frequently, Donnybrook replaces each avatar by an artificial intelligent (AI) bot that loosely replicates the behavior of the avatar. This loosely replicated behavior is based on guidance messages that are sent only once per second. Each peer only maintains complete and up-to-date information for a relatively small set of players (the interest set); for these, no AI player is used.

4 Recommendations for Future Work

This section describes three directions for future work that we recommend based on the observations made in Section 3. These three directions are: system security, usage of community resources,
and an extension to our taxonomy.

The architectures described in Section 3 mainly focus on the technical aspects of game operations such as responsiveness, reliability, and scalability; security is generally considered to be out of scope. Peer-to-peer architectures typically shift control from publisher-owned servers towards potentially untrustworthy peers, which introduces security risks that do not exist in client-server architectures; for example, players can add items which they are not supposed to have to their inventory. Future work is recommended to determine how to overcome these security problems.

The use of community owned resources in a peer-to-peer based architecture has not yet been studied thoroughly. Table 1 shows that community resources are only used in one architecture [Chambers et al., 2006], which studies the use of community resources in a client-server based architecture. Community resources can potentially be used to overcome resource limitations, such as bandwidth limits. We therefore suggest for future work studying the applicability of community resources in a peer-to-peer based architecture.

In this work, we have compared MMOG system architectures based on security, responsiveness, reliability, scalability, and resource ownership. Although these cover the most important aspects in MMOG architectures, other aspects exist; for example consistency – which differences are condoned to exist in the local view of the game world between different players – and content generation – to what extent can players or communities add game content, changing game objects or the game world. Our taxonomy and survey can be extended with these two aspects, this extension is left for future work.

5 Conclusion

Many MMOG system architectures have been recently proposed to address the exponential growth in MMOG community size, but there exists no comprehensive overview of these system architectures. Since MMOG system architectures are very different by nature, it is difficult to choose which architecture to use. This paper has presented an overview of various MMOG system architectures that either have been proposed by researchers or are used in the industry.

To this end, we have first proposed a taxonomy based on security, responsiveness, reliability, scalability, and resource ownership. Although these cover the most important aspects in MMOG architectures, other aspects exist; for example consistency – which differences are condoned to exist in the local view of the game world between different players – and content generation – to what extent can players or communities add game content, changing game objects or the game world. Our taxonomy and survey can be extended with these two aspects, this extension is left for future work.

References


