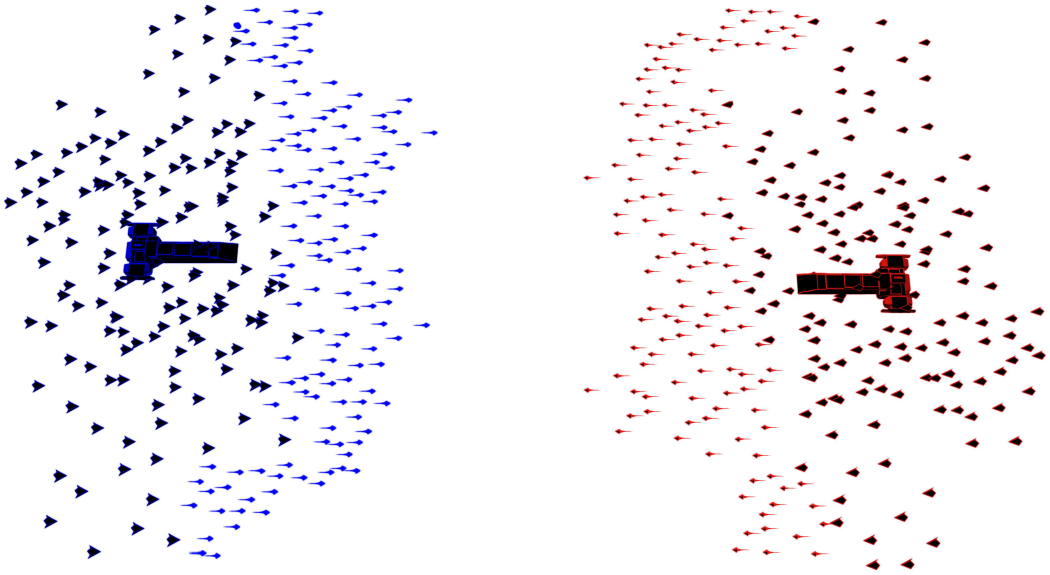
# Distributed Game Engine
# for
# Massively Multiplayer Online Shooting Games



Dat5-project by
- Group d519a -

Aalborg University, January 15th, 2010

AALBORG UNIVERSITET

**Department of Computer Science**
Aalborg University
Selma Lagerlï£¡fs Vej 300
Phone 96 35 80 80, Fax 98 15 97 57
http://www.cs.aau.dk

**Titel:**
Distributed Game Engine for Massively Multiplayer Online Shooting Games

**Theme:**
Distributed Systems

**Project period:**
DAT5, fall 2009

**Project group:**
d519a

**Authors:**
Ron Cohen
Anders Ejlersen
Rasmus Kristensen

**Supervisor:**
Brian Nielsen

**Printcount:** 5

**Nr. of pages:** 61

**Appendixcount and -type:**
3 pages and a CD

**Completed & signed:**
January 15th 2010 at Aalborg University

**Abstract:**

Massive multiplayer games are rising in popularity giving game creators an increasing incitement for building massive multiplayer games which support an ever increasing number of players sharing the same game world. However, continuously increasing the number of simultaneous players will exert extensive strain on the network architecture making it difficult to maintain a acceptable service level. As such, it is difficult to build a network architecture which can scale with the number of many simultaneous players.
This report is the first in a work consisting of two reports. The work pertains to networking in multiplayer games and distribution of work-load to support massive scalability.
This report focuses on existing research done in MMO games and with networking in games in general. It gives a thorough analysis of the state of the art of networking in games of different genres including massive multiplayer online games. A generalization of MMO games is performed to give a generic MMO networking architecture. The report is accompanied by an implementation of a simple game which forms the basis for the ideas developed in the second report.

# Contents

# Preface

This report has been written during the Dat5-project period, by computer science group d519a at Aalborg University. The area of the report is in "Distributed and Embedded Systems" and therefore mainly consists of knowledge from that area. This report is addressed to other students, supervisors and anyone else who might be interested in the subject. To read and understand the report correctly, it is necessary to have knowledge about the most basic computer related terms.

The entire report is written in English and no translation will be accessible, it is therefore required to understand English. Abbreviations and acronyms will at first appearance be written in parenthesis, to avoid breaking the reading stream. Furthermore a list of game terminology is found in Appendix A. Specification of gender in the report is not to be understood as suppression or any other form of political/religious position. The gender is only specified to simplify the process of writing for the authors.

References to sources is marked by [#], where # refers to the related literature in the bibliography at the end of the report.

The appendix to the report is found in the last chapter of the report, and on a compact disc, located on the very last page of the report.

The report is written in LaTeX and is accessible as a PDF-document, which can be read with Adobe Acrobat Reader.

***Signatures:***

| Ron Cohen | Anders Ejlersen | Rasmus Kristensen |
|---|---|---|
| _____ | _____ | _____ |

Chapter 1

# Introduction

The computer has become widely used for interactive entertainment, e.g. games which are played against or with either an artificial or human player. There are different definitions of what a game is, Greg Costikyan[1] (an american game designer and sci-fi author) used the following definition:

*"A form of art in which participants, termed players, make decisions in order to manage resources through game tokens in the pursuit of a goal."*

Thereby a game is a form of challenge between the logic of the game and players to reach the goal of the game. This definition does however not completely cover what a *video game* is. A video game is a game played not by using physical objects, but by using a physical controller and thereby influencing a virtual object or objects on a screen or TV.

Video games can be divided into game genres, such as Role-Playing Games (RPG), First-Person Shooter (FPS), or Mass-Multiplayer Online (MMO) games, which are just some the genres. Each of these genres have some distinct features or characteristics. In MMO games a vast number of players participate in the game together via a computer network.

EVE Online[2] is an example of a MMO game that has all players combined onto one game world. EVE Online is also a RPG game which means it usually has relatively loose requirement to latency (the time it takes for player actions to take effect in the game world), due to the way game mechanics work. In RPG games player interaction is often limited to explicitly select targets. Meaning, the user will click the player she wishes to interact with. In opposition to this, FPS games usually require a lower latency to have an optimal game experience. This is because of the fast pace and requirement to aim at enemies. A high density of players in one area of the game world, called *crowding*, can put immense stress on the game servers[3], increasing latency and thus giving players a poor game experience.

During the history of multiplayer games, cheating has been noticed as a significant problem, that can affect the game experience for other players. Game developers try to prevent cheating, because of the bad reputation a game can get when there are players with an unfair advantage. In the RPG Diablo I[4] cheating was a major problem which developers tried to remove in the sequel Diablo II[5]. The major difference in Diablo I to Diablo II was that the avatar data, information about a virtual character a player controls in the game world, was moved from the unsecured clients to a secure server, which made it more difficult to cheat.

## 1.1 Motivation

MMO games have become increasingly popular. An example of this is the RPG game *World of Warcraft*[6], which had over 10 million subscriptions at January 2008[7]. Due to the large number of players it is important to make the games able to scale with the number of players. Another important aspect is how latency affects the playing experience, and each game genre has its own requirement regarding latency, e.g. RPG games does not have as strict requirements to latency as, e.g. FPS games or shooter games. This is because FPS games are usually fast-paced and require quick dissemination of information due to movement of other players being important for aiming, and so on.

Below is a list of difficulties which should be addressed in a MMO-FPS solution:

- Low latency required for fast-paced game play and a "real-time experience".
- Ability to service a large number of players at close vicinity of each other in the game world (crowding).
- Restrict the potential for cheating and give game creators control over who is playing.

MMO games have different solutions for the problem with many players. World of Warcraft has a limit for the maximum number of players that can play on each shard (An isolated persistent replica of the game world), however this does not address the problem of crowding. In EVE Online one shard solution for all players was created, but has a limit of about 1500-1600 players at one location, which gives problems with high latency for the players. A higher volume of players at one location, experiencing little latency could yield a more intense game experience.

The consequences of cheating in games can have fatal consequences for the game's community, as seen from Diablo I. Therefore a networked multiplayer game (abbreviated: "Multiplayer game") should minimize the potential cheating to ensure a good gaming environment.

## 1.2 Specification

Above we mentioned two MMO games; EVE Online and World of Warcraft. These two games have different approaches to managing their players. World of Warcraft uses a multi-shard architecture, where the shards have a population limit, whereas EVE Online uses a single shard architecture to accommodate all their players. Both, however, require a distributed solution using multiple machines to ensure acceptable performance. Because of the difficulty of creating a huge game world in which every player has a consistent view of the game world, and creating a solution that is scalable, a single shard architecture is interesting. The interesting aspect is how a scalable solution can be created that maintains consistency and a low latency in the game world, how a game world can be divided such that the workload can be distributed, and how a load balancing mechanism can ensure the correct placement of workload for maintaining playability in the game world.

The two MMO games; World of Warcraft and EVE Online both use a client/server architecture, but a peer-to-peer approach could also have its merits. Therefore it would be interesting to see how the two network architectures can be used to ensure a scalable solution, while limiting the possibility of cheating.

Due to the different requirements between RPG games (the genre of both World of Warcraft and EVE Online) and shooter games, it is mainly the latency requirements, that

we seek to fulfil in a MMO solution. The need for very timely consistency in shooter games (due to the aiming and fast-paced movement) requires more frequent updating of game state to maintain the fast-paced gameplay.

## 1.3 Problem Definition

This project will examine techniques for creating scalable MMO-FPS games using a distributed and parallel engine with emphasis on remedying the problems that occur with crowding.

Specifically, the project is tasked with proposing, implementing and evaluating an architecture for MMO-FPS games which addresses the following problems:

- Dividing the game world while maintaining consistency.
- Handling crowding and maintaining low latency.
- Minimizing the potential of cheating and enabling control of who participates.

The techniques will be evaluated through a test-game developed specifically to exert the properties and requirements of a MMO-FPS game. The test-game will both provide a playable version of the game, but also provide a simulation mode in which multiple human players can be simulated.

As this report is the first part of a series of two, the main goal of this part of the project is to analyse different approaches for a MMO game. A test game will also be created to be able to test and compare different approaches. The main approach tested in this part will be a simple solution to establish a baseline for further testing in the second part.

## 1.4 Report Structure

The report contains an analysis chapter, which investigates the different network architectures used in traditional multiplayer games, but also in MMO games. The analysis also describes how interest management techniques work and why the crowding phenomenon is a problem in MMO games.

Next we briefly introduce the goal of the project, a sketch of the solution idea at which we would like to arrive, the methodology used in the project, and a description of the game concepts and mechanics in accordance with the methodology. Subsequently, the design and implementation of the game is presented, this includes decisions made with regard to the network architecture, and so on. The chapter also contains implementational details describing the main aspects of the implementation, how the region splitting works and the interest management scheme that has been implemented, and so on.

Next the implementation is evaluated with the test game. This is done by looking at different metrics, which can affect the performance of the servers.

This report is first of two reports. This report, with the application developed along side, will form the basic platform for further development and refinement in the second report. The first report will analyse, design, and implement a test game, using a basic single server platform. Additionally, we will evolve this solution into a multi-server using well-known concepts and perform performance evaluations of these two solutions. The second report will build on this platform to implement and evaluate a novel approach to scaling and handling crowding in MMO games.

Chapter 2

# Analysis

This chapter presents an analysis of the components needed to create a MMO-FPS game. Firstly, a general description of the inner workings of game engines is given. Then, the requirements that exist with regard to networking in games are explained. Specifically, the impact of latency in multiplayer games is analysed in the context of different game genres. Also, scalability and consistency requirements are analysed. Subsequently, an analysis of the peer-to-peer and client/server architectures are given. In addition to this, techniques to achieve scalability in MMO games and the degree to which problems like crowding impacts the game experience is described.

Different interest management techniques are also described to give an overview of these. This includes an analysis regarding techniques for subdividing game worlds in order to achieve scalability. Subsequently, a generic MMO network architecture is given. Then, the messaging scheme used in multiplayer games is analysed. Furthermore, four different games are analysed to create a basis for a MMO-FPS game, namely Diablo I, Diablo II, Quake III Arena and EVE Online. Finally, an analysis of the special issues that exist when combining the MMO genre with the FPS genre is presented.

## 2.1 Game Engine

When creating a game a certain amount of time is used on designing and implementing a game engine, which handles the tasks like rendering the graphics, input from the user, sound, and so on. This section will cover a general structure for such an engine and the tasks that it needs to take care of.

### 2.1.1 Structure

The structure of a game engine is a set of subcomponents that is used to model the environment and behavioural aspects of the game[8, 9]. In Figure 2.1 (a) an abstract model of a game engine is provided. The game engine exist of the following subcomponents:

**Core:** The core is the coordinator within the system. The main objective is to start the game and all the components needed within the game. It therefore maintains communication between the subcomponents and manages memory, file I/O, and so on.

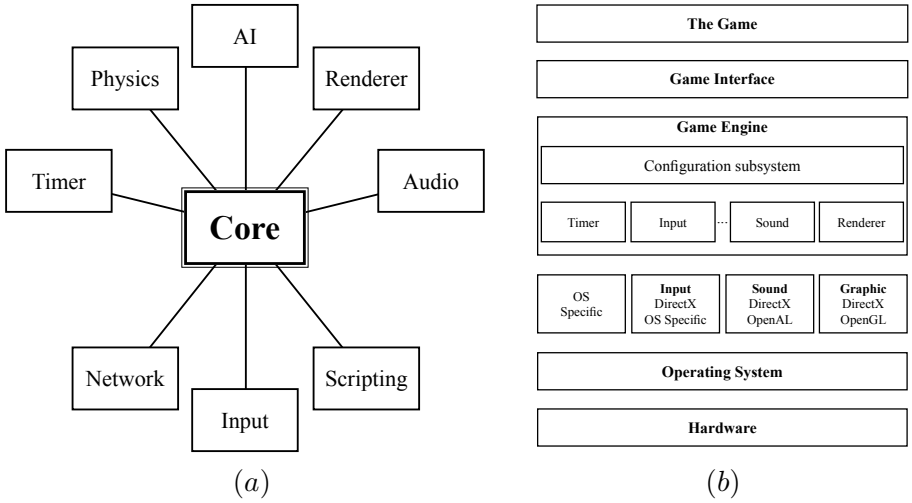



*Figure 2.1: (a) An abstract illustration of different components in a game engine, (b) structure of the layers in a game, game engine, and the underlying OS and hardware layers.*

**Timer:** The timer component is used in the system to keep the game running fluently and consistent on all sorts of hardware. It is used in animation, physics, prediction, game time, and so on. In prediction it is used to predict the next possible step in, e.g. a movement from $A$ to $B$. This is because of the latency from pressing the movement button on the client to receiving the actual move from the server. So one way is to predict it by using a previous position, extrapolation, and thereby find the new position, when the client then receives the correct position from the server it can interpolate from the predicted current position over to the new position.

**Physics:** The physics component maintains the behaviour of the environment. This could be how gravity, friction, movement, and so on works. There are some major middleware solutions that is often used, because physics are often the same in most games, but sometimes physics needs to be developed with the game, because the game physics can be unique for that particular game.

**Artificial Intelligence (AI):** The AI component plays a role if the game universe have Non-Player Character (NPC) playing a part of the game. The AI is often implemented uniquely for each game, since the behaviour of the NPCs often is required to behave in a unique fashion. The AI can also be used in player characters to do path finding and simulation of human players.

**Renderer:** The renderer component maintains all the graphics that needs to be rendered to the screen. It does this by utilising a scene graph consisting of all the objects within the game environment. There are two major graphics APIs used for rendering objects: DirectX and OpenGL. The first is a Windows only graphics API, whereas OpenGL is platform independent.

**Audio:** The audio component maintains the sound within a game. How, when and where it should be played in the game world. There are two major sound APIs: DirectX and OpenAL.

**Network:** The network component manages the communication between, e.g. client and server. This is done by using UDP and/or TCP communication. Section 2.5 goes into details about networking.

**Input:** The input component manages the input events from the user. This can be from mouse, keyboard, or other types of controllers. The APIs for input are OS specific, e.g. Windows uses DirectX for input management.

**Scripting:** The scripting component is used to make game play and content creation easier. This is done by using a scripting engine that can parse scripts to game content.

In Figure 2.1 (b) all the components are placed in the game engine. The game itself is the layer at the top. It uses an interface to the game engine in which it can find all the functionality the game engine has and thereby use it for the game. Beneath the game engine layer there are the different components maintaining sound, graphics, and so on. This is maintained by the operating system, which in turn is running on top of the hardware.

### 2.1.2 Tasks

The game engine has a lot of different tasks, as mentioned in the previous section. The tasks that are most noticeable are: Physics, AI, and lastly the game logic. Even though some of them have been described in the previous section, they are elaborated again here:

**Game Logic:** The game logic are the set of rules that the game objects must obey. This can be how fast a car can move, accelerate, turn, and break. Therefore game logic also defines how the physics is handled. If a tire explodes, should the physics applied to the explosion be realistic giving a more difficult steering or should it be more comic-like, where the entire car flips 360 degrees before driving along as nothing had happen.

**Physics:** The physics is how gravity, friction, and so on is modelled in the game world, but it is also about how objects react if they collide. Therefore in physics there is collision detection and response, that models when objects collide and how they should react on the collision. E.g. a ball should react differently to the pavement, than a vase.

**AI:** The AI uses the game logic and physics to model their reaction on different elements. This can either be done by using state machines, fuzzy logic, or other AI models to model how they should react. When nearing a curve in the road, should they break or should they speed up, and also how to react if they hit the wall.

These three tasks are some of the most important, because they define the game environment, rules, and NPC collaboration or resistance in the game. However AI can be discarded in games, where player characters are enough to emulate the game environment.

### 2.1.3 Game Loop

The game loop is an important part in a game. It makes sure that the game keeps running smoothly, even though the game is not receiving any input from the user or the network. The game loop is often very simple and is often structured as shown in Figure 2.2.

The game loop checks for user input to see if the user wants to do something in the game world. Then it runs the AI, such that the NPCs behave according to the things happening

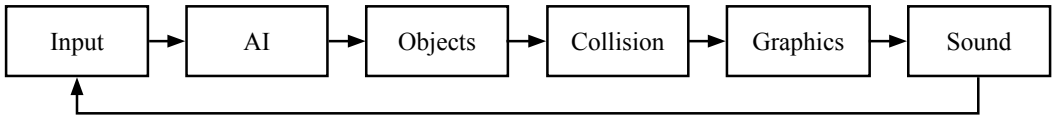



*Figure 2.2: A standard game loop.*

around them. Then it moves the game objects, this can both be the NPCs, player characters, and other moving objects. When all objects have been moved, then there is a need to check for collision and if there is a collision between two objects, then a response must be made, that counteracts the collision. When all this is done, then the game world is drawn on the users screen and the sounds are played.

The game logic is a part of several of the components in Figure 2.2. The game logic is used when a user gives some input to the game, e.g. a key press to accelerate the car. It is also in the AI, because they are also restricted by the rules of the game, so if a NPC wants to deaccelerate, then must be applied according to the game logic. The game logic also has a small part in the collision, such that if a projectile hits the tire of a car, then the car tire should explode or just puncture, this is all evaluated according to game logic. Lastly there is the sound, for instance an exploding tire gives a sound: The game logic specifies the sound to be played.

## 2.2 Requirements

This section describes the requirements which exist in a multiplayer game, particularly the requirements which relate to the network architecture chosen for a given game. These requirements are consistency, latency, and scalability.

### 2.2.1 Consistency

The main issue in developing multiplayer games is to give each player a consistent and timely view of the game world. In order to achieve this, clients must continuously receive updates describing the changes that occur in the game world. These changes include movement of objects, creation of new objects, removal of old objects, interaction between objects, and so on. An example of the difficulty that is involved in this process follows:

A group of players are exploring a cave that exist in a game world. The group of players hopes to find a chest and at some point in their search they find a chest. The chest can contain a gold coin or a bomb. If the players see that the content is a gold coin they will all be interested in picking it up. However a bomb has a negative impact on the players progress in the game. One player opens the chest to see the content. As the content of the chest will decide the further actions of the players, it is important that the information regarding the contents of the chest is available to all players at the same time, such that one player does not have an unfair advantage to pick up the coin. Additionally, two players may decide to pick up the gold coin at the same time. It is important that they are not both allowed to do this, as there is only one coin in the chest.

In practice, simultaneous revelation of information is approximated. It can be approximated to an acceptable level in most cases as long as there is no or very little delay *in*

*addition* to the network delay. Such extra delay may be caused by the need to process game mechanics or graphics rendering at the clients. Thus, in order to achieve the best player experience, it is important to keep the delays incurred to a minimum. Each player in the group must know about the contents of the chest and thus must receive packets describing the contents. The consequence of this problem is that there is an inherent minimum amount of information which should reach all clients, and thus as the group of participants in this cave exploration grows, so does the amount of information which must be transmitted, and arrive in a timely fashion.

Consistency is achieved by having a specific authority on the objects in the game world. In this way, when two players both decide to pick up the coin, they send requests to the authority and only one of the players is allowed to pick up the coin. This adds considerable overhead in that interactions are required to be authorized before the game can continue.

It is important to note that the group of players participating in the cave exploration might just be a single group out of many groups playing simultaneously in the game world. Outside the cave, other groups may be roaming the terrain of the game world. However, as only the players participating in this particular cave exploration need to know about the chest, the information need not be transmitted to players outside the cave. This is called *localised game play*, and it facilitates *localised consistency*. Without some form of localised consistency the limit for the number of players which can participate in the same game world is quickly reached, as it becomes very difficult to maintain a consistent game state across all the participating clients.

### 2.2.2 Latency

Latency is important in multiplayer games, because the time it takes for a message to be communicated amongst sender and recipients can disrupt the experience of real-time gaming. This is because that the difference in time between messages sent from the sender to different recipients can yield an advantage for the recipients with low latency against the ones with higher latency. Latency can be defined in two ways:

**One-way latency:** The time it takes for a packet to travel from the sender to the recipient.

**Round-trip latency:** The time it takes for a packet to travel from the sender to the recipient *and back* again.

When measuring *one-way latency* both sender and receiver are involved. Further more, the accuracy of the measurement depends greatly on the degree to which it is possible to synchronise the clocks of the sender and the receiver. As exact synchronisation of the clocks is not possible, one-way latency can only be approximated. To measure one-way latency, packets are augmented with their departure time from the sender. When they arrive at the recipient, it can approximate the latency by subtracting the departure time from the arrival time of the packet. As the clocks of sender and receiver are synchronised (to some degree of accuracy), this will approximate the latency.

*Round-trip latency* on the other hand, is measured by sending a packet to the receiver and expecting it to return an acknowledgement immediately. In this manner, the sender will augment a packet with its departure time and subtract the time at which it gets the acknowledgement of this packet.

Latency is important in multiplayer games because it defines the time it takes for input from the user to take effect. In order to ensure a fluent and exciting gameplay, it is important

to keep latency at a minimum. If the latency of a game is too high the real-time feeling of the game is obfuscated and the game becomes unenjoyable and possibly unplayable. Thresholds for when games become unenjoyable differs with game genres. An example is fast paced shooter games which requires a low latency to give the desired feeling of the game[10], on the contrary, turn-based games can have a high threshold for latency before the game is no longer enjoyable. The fast paced shooter games is desired to have a low latency in the area of maximum 100-150ms[10]. This limit is set to let the player have an experience where the latency are not noticed and the game seems fluid. The limit is also depended on many other aspects of the game, e.g. prediction and gameplay, which can alter the feeling of the game.

### 2.2.3 Scalability

The scalability of a distributed system is the ability to either use more or less resources given the size of the workload. In a MMO game the workload is often the number of users and the resources is servers or hardware. In MMO games the game world can sustain several thousands of players, as opposed to 2-128 simultaneous players in traditional multiplayer games. Therefore there is a need to make the server structure as scalable as possible to sustain the growing number of users. Increasing the number of players in a game world, will increase the workload on the server. Unless the system is *scalable*, the additional players will cause an increase in the latency experienced by players.

Scalability in network architectures is a much sought-after property, which enable network architectures to scale up the supported workload simply by adding more machines. In order to achieve this, the network architecture must be designed with scalability in mind. MMO games exploit localised game play to facilitate localised consistency and employ *server clustering* and *load-balancing* to achieve scalability. While the details of the implementations are very sparse, there is information about some games and it is fair to assume that similar games employ the similar principles[11].

However there are certain scenarios that a scalable server structure might not cope better in, then a traditional network game. This scenario is called crowding:

**Crowding:** A phenomenon where many players flock to one area of the game world. This phenomenon is often seen in MMO games, where an area in the game world suddenly experiences a rise in population. Crowding constitutes a major issue for game creators. Because of the huge amount of players located in close vicinity of each other, they must all be made aware of each others actions. This leads to increased latency, possibly hindering player interaction.

The crowding phenomenon is a problem in both traditional and in a scalable server solution, because if a certain region held by a server is crowding to its maximum capacity, then the server will not be able to handle the workload and the latency between the server and users will increase.

## 2.3 Network Architectures

Multiplayer games have some requirements for the network architecture, as mentioned in Section 2.2, that it should fulfil. The different game genres have different limits for tolerable latency, e.g. FPS-type games tolerate only very little latency, while RPG-type games are typically playable with considerably more latency. Because of these different requirements,

different network architectures might be applicable to different game-types. However, in general, game designers strive to minimise perceived latency to ensure a good gaming experience.

### 2.3.1 Peer-to-Peer

In peer-to-peer oriented systems every participant is an equal peer in the network. Peers talk directly to each other and they all run the same program and therefore both acts as client and server. A peer-to-peer network topology is illustrated in Figure 2.3 (a).

Peer-to-peer have been used in some of the first multiplayer games. Each computer participating was equal. These games include *Doom* and *Duke Nukem*, which both are FPS games. An approach used in such games was a *lock-step* approach. The lock-step approach meant that the computers involved computed exactly the same set of operations in parallel. Every input and timing was synchronised and with deterministic game logic, every computer had the same perception of the game world[12]. However, the lock-step approach had a number of disadvantages, namely: All players had to start the game at the same time and new players could not join in an already started game. Additionally, the lock-step approach made the game very susceptible to network induced errors, such as dropped packets, making it hard to scale up the number of players. Finally, all players had to run at the same internal framerate in order to be able to synchronise events.

Any peer-to-peer based architecture suffers from well-know problems that are inherent to peer-to-peer based applications today, including firewalls and NAT-traversal problems[13]. This severely complicates any peer-to-peer based solution.

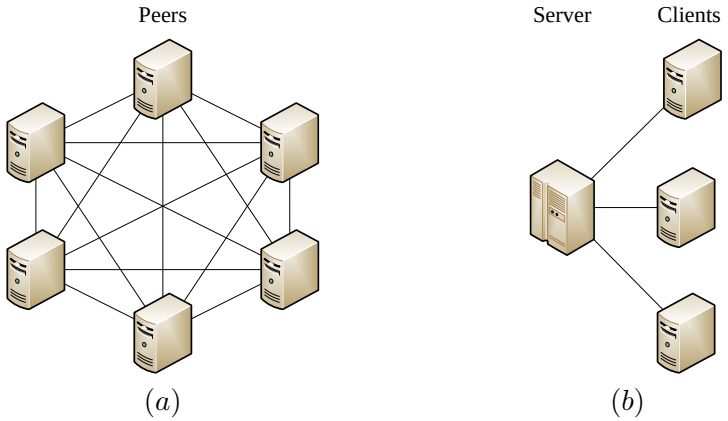



*Figure 2.3: (a) Peer-to-peer network (b) Client/server network*

### 2.3.2 Client/Server

Client/server based applications dedicate a particular set of machines to act as *servers*. The servers run an application in which clients are interested in subscribing to. The clients use a similar application, which can connect to the servers. All communication usually goes through the servers if clients are interested in communicating with each other, as illustrated in Figure 2.3 (b), which illustrates a client/server network topology.

Games like Quake I used a client/server architecture where one machine was designated the *server*, while the clients would send their input to the server and receive a list of objects

to render. The client/server architecture evolved with Quake II, moving prediction and simulation logic into the clients improving the bandwidth usage. This was because that both the server and client would simulate the same environment, but where the server would have all the authority, so when a client made a prediction of another clients movement, then it would do this by using the servers position of that particular client. This made it possible to reduce the number of packages sent between the server and client, because the client could predict the next position of the client and adjust the prediction with the next message from the server.

The client/server approach does not suffer from the NAT traversal issues that are inherent to peer-to-peer based applications. This is due to the fact that in a client/server based solution, the server administrators have control over the NAT devices placed in front of the servers, and they are free to configure them so the servers are possible to reach from the clients.

### 2.3.3 Network Architectures & MMO Games

The two predominant network architectures available are client/server and peer-to-peer. However, it is clear that the client/server approach is preferred in commercial MMO games due to the following requirements:

**Cheat prevention:** All data goes through official servers, which ensures that input from clients conform to the rules of the game world.

**Control:** Due to the fact that many MMO games are subscription based, and that it is desirable to be able to evict players for different reasons, it is important for MMO game creators to maintain some degree of control over who plays the game.

In peer-to-peer based architectures, these requirements are hard to realise as opposed to a client/server architecture. Due to the fact that the responsibility is distributed onto all connected peers, control would become difficult, since there is no centralized mechanism to keep this control. The last thing is cheat prevention, this would also become difficult, since there are no centralized mechanism to check for malicious commands from other peers.

## 2.4 Generic MMO Architecture

In this section we present a generalized network architecture for MMO games obtained from the lessons learned from the analysis of network architectures in Section 2.3.

Figure 2.4 depicts a simplification and generalization of the architecture used in commercial MMO games. The architecture shown is a single-shard architecture, but should be easily changeable to be applicable for multiple shards.

The architecture is included to give a general overview of the server architecture. Any specific implementation will be much more elaborate, however, the general concepts remain the same.

As apparent from the depiction, the generic architecture contains three distinct layers: Client, application, and persistence layer.

**Client layer:** Contains the set of clients currently participating in the game. Clients connect through the internet to the application layer. The responsibility of the clients is to act as the interface between the player and the application layer. Clients will take

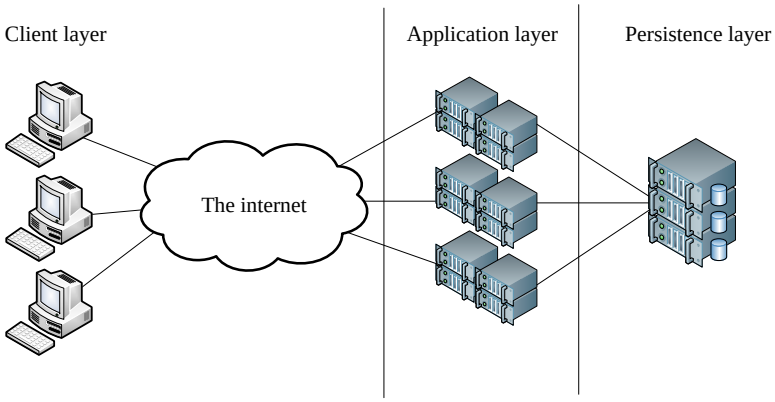



*Figure 2.4: Generic client-server architecture used in MMO games.*

input from the player and forward it over the internet to the application layer - and receive updates from the application layer and present these to the player.

**Application layer:** Responsible for receiving input from clients, processing the input, and sending updates to clients. This layer can be very complex and consist of several sub-layers, such as:

**Game servers:** Calculate the actual game logic, physics, score-keeping, and so on.

**Load-balancers:** Responsible for balancing the load across machines in the application layer.

**Proxies:** Responsible for disseminating the updates back to clients and forwarding updates from clients to the intended machine in the application layer. They also assist in creating a uniform view of the server architecture for clients and hiding complexity from the clients.

**Persistence layer:** Used to allow application layer machines to save information for use later. Often, the persistence layer is presented to the application layer in a coherent and uniform way in order to simplify the application.

Depending on the specifics of the implementation, different processes take place in different layers. In order to address the problems mentioned in previous chapters, server clustering and load balancing is used.

### Server Clustering

Traditional multiplayer games use only a single machine acting as the server, as mentioned in Section 2.3.2. To utilize server clustering, it is necessary to divide the responsibilities of the server such that each server can be assigned a responsibility. The servers then work together to serve clients. A good division of responsibilities is paramount to successfully clustering servers.

Server clustering is tightly coupled to *interest management*. Interest management helps achieve good division of responsibilities. While interest management is highly application specific, there is a number of interesting general concepts as will be described in Section 2.6.

Please take notice that the term *server* is used in this report to mean an instance of a server process. Thus several server processes can run on the same physical machine. This is particularly applicable in multi-core machines.

#### Load Balancing

Load balancing is used to distribute workload across a number of machines. Good load balancing should afford:

**Workload distribution:** Distribute workload, giving less work to overloaded machines and more work to underloaded machines.

**Flexible configuration:** Make it possible to add and remove machines dynamically.

**Complex gameplay:** Do not enforce artificial restrictions to game-play.

There are two basic ways of doing load balancing in a MMO context:

**Player:** Each player is assigned a server (or a cluster of servers) when starting a game session. This assignment is kept for the duration of the game session. This is called a "sticky" session.

**Interaction:** Load balancing is done by analysis of interaction patterns of players.

Player-based load balancing is akin to standard load balancing performed for web-servers and other application delivery services. Inserting a layer Network Address Translation (NAT)-devices between the internet and the application servers is an example of implementing load balancing using "off the shelf" equipment[11]. In practice, players would be assigned to specific application server by a NAT-device in the beginning of the game session, and all communication from a specific client would then be forwarded to a specific application server in a round-robin fashion. More complex solutions could also be devised, e.g. measure load on servers and distribute players to servers with low load.

Interaction-based load balancing is performed by analysis of the interaction patterns of players and is considered to be computationally expensive and complex compared to player-based load balancing.

## 2.5 Messages

This section investigates the message exchange in games and therefore about which has the responsibility in a client/server architecture. Lastly it investigates the scheme used for communicating a message, that yields a lower latency as required by some game genres.

### 2.5.1 Messages in Games

In multiplayer games the players must continuously receive updates about the events occurring in the game world. These events can be triggered by other players, e.g. player movement, or they can be a consequence of the non-player mechanics of the game, e.g. a NPC moves, talks, or acts otherwise in a way that requires adjacent players to be notified.

When a player expresses interest in influencing the game world, e.g. by pressing the "move character forward" button, a message detailing the event is transmitted from the

player's client to the server, such that it can calculate the consequences of this influence. These consequence typically include movement of the player in the game world, movement of objects, and so on. Finally, these changes in the game world are propagated back to all players, and subsequently the players' screens are updated to reflect the new game world state.

Some of the messages that could be sent between the clients and the server are: Movement, spawn, death, chat, shots, and so on.

The responsibility assigned to the client and the server respectively plays a very big role in restricting the possibilities for cheating, which was one of the key factors for using a client/server architecture as mentioned in Section 2.3.3. Specifically, the more responsibilities assigned to the server, the less potential for cheating exists. In one scenario, the client may be responsible for interpreting the user input, moving its own character according to the input, and then sending the new location to the server. In this manner, the server should be able to support a large number of clients simultaneously, because it is not spending time calculating the movement of characters. However, a malicious player may alter his client software, such that his avatar moves faster than allowed by game rules or otherwise break the rules that exist in the game world. In order to ensure least potential for cheating in the game, clients are reduced to accepting input from the user and sending it to the server in some abstract form. For example, instead of "W-key is pressed", it will send "Move character forward". It is then the responsibility of the server to apply this information to the internally represented game world and transmit the updated game world information to clients. The reason for abstracting over key-presses, and so on, is that it gives a lot of flexibility for the client to interpret the user input as the user sees fit. For instance the user may want to use a joystick instead of a keyboard.

In this sense, the client program is merely an input and rendering terminal which sole purpose is to convert the information retrieved by the server to a visual image, and accepting input from the player, transmitting it to the server. This is sometimes referred to as a *thin client* as opposed to *fat clients* who have more responsibility.

### 2.5.2 Communication of Messages

The details regarding messages communication between clients and server can be quite involved. Messages can be sent using a protocol that guarantees reliable delivery, e.g. TCP, or over protocols which do not give such guarantees, e.g. UDP. While TCP guarantees delivery, it comes at the cost of additional overhead of maintaining a connection, ensuring delivery, ordering, and so on. This overhead induces additional latency on the communication.

Delivery guarantees are useful if it is important that a specific message reaches its destination. For instance when logging in to a game server, as mentioned in Section 2.3.3 about control, it is important for the client that the user-credentials will reach the server in order to continue the login process. However, in some cases packet loss is acceptable. For instance, a client may send a message describing that the player wants her character to move forward. The client will continuously send this message with some interval. If a few messages are dropped, e.g. due to network congestion, it is not critical as the message is quickly succeeded by another. This is only possible, because the whole state is transmitted in each message. For example, the state of each button of the clients keyboard is transmitted in every message. If the message only contains changes since last message, e.g. forward-button was toggled, messages can be kept much shorter, but the loss of a message can be critical. The advantage of using a protocol that does not guarantee delivery is that the messages

which *do* reach their destination usually arrive quicker than with protocols guaranteeing delivery. This is very important to maintain low latency.

### Delta-Encoding

Some games which require very low latency take this even further, transmitting all messages over UDP. However, instead of transferring the whole state in each message, they *delta-encode* it. Delta-encoding is applied to packets in order to avoid sending the whole game state for every packet. It does this by calculating the difference $\Delta(s_u, s_v)$ between two game states $s_u$ and $s_v$. Then only the change between the states is transmitted, keeping the packet size small. This might seem like a problem if packets can be dropped, however when delta-encoding is used in cooperation with *acknowledgements* the problem is remedied.

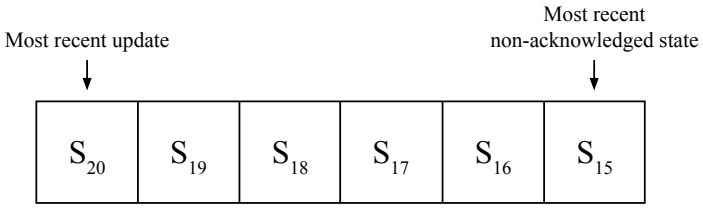



*Figure 2.5: Illustration of the state history maintained at the sender. Sender will continue to send $\Delta(s_{20}, s_{15})$ until $s_{15}$ is acknowledged.*

When delta-encoding the game state and using an unreliable protocol such as UDP, recipients are required to send acknowledgements back to the sender. All packets are augmented with a sequence number and the acknowledgements denote the highest sequence number that the recipient has seen. This way the sender always has some idea of which packet is the most recent packet that the recipient has received. The sender will continue to send packets delta-encoded against the last acknowledged packet. It is common that a sender will send multiple delta-encoded packets before it gets acknowledgement of the first sent packet, thus the sender must maintain a list of packets which have not been acknowledged by all recipients yet. This is done in order to always be able to delta-encode against the game states. Figure 2.5 shows an example of how such a list might look. When all recipients have acknowledged a specific state, any preceding states as well as the state itself can be removed from the list. Acknowledgements in the form of a sequence number can be piggybacked on normal traffic making the transfer cost negligible. There is, however, some overhead in maintaining and storing the list of game states and performing the delta-encoding. When delta-encoding packets in this manner, packet sending becomes *idempotent*. That is, the same packets can be sent numerous times without disrupting the state information at the receiver.

## 2.6 Interest Management

Interest management is used in MMO games to divide responsibilities and limit the amount of data each client is concerned with. Responsibilities include, which servers should handle which objects, which clients should receive which information and which information is relevant to an object or client. Division of responsibilities is necessary to utilize multiple

server machines, and achieve scalability. Interest management is also important to ensure that clients are not overloaded by information.

To determine which information is relevant for a client, an interest management scheme is used. The scheme used is very dependent on specific game mechanics. However, most MMO games model our reality to some degree and thereby a player avatar usually has a range of sight, ways of communicating, and weapons have limited range.

Exploiting the range of sight is a typical example of interest management. The avatar needs to know of events within its range of sight. Anyone within the range of sight of the avatar would be interesting for the player. In this manner, the client representing the avatar will receive game state regarding only the events that take place within its range of sight.

In this section we describe methods to distribute responsibility for objects from a game world onto different servers, particular emphasis will be given to *region division*, as this is used in existing solutions[14, 15]. In addition we describe methods to determine which information is relevant to an object or client, with focus on various aura based approaches as they are also used in existing solutions[16, 14, 15].

### 2.6.1 Region Division

Different approaches can be used to divide the workload of the game world into manageable portions. One such approach could be to assign different objects to different servers. However if objects with a high degree of interaction between them are on different servers a large overhead in communication would occur compared to if they were on the same server. As already mentioned, the usual game world is created so that it models a reality close to ours, meaning objects within sensing range, e.g. they can see or hear each other, need to know of each others actions. In these cases it can be an advantage to divide the game world into regions as illustrated on Figures 2.6 (a) and (b), such that each region handles all the objects in that particular region. However, communication between servers handling regions is still necessary if objects can move or look across region borders. The region divisions must be created, such that this communication is also kept at a minimum.

The main goal of regionalisation is to efficiently exploit localised game play and consistency as described in Section 2.2.1.

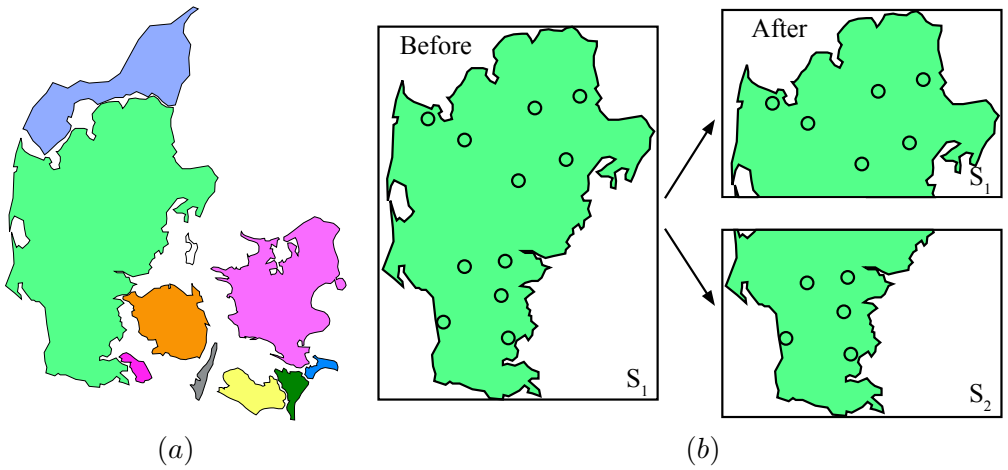



*Figure 2.6: (a) Illustration of static regionalisation of Denmark, (b) dynamic regionalisation of an area where the two subsets are distributed to server $S_1$ and $S_2$.*

When using regionalisation there are two approaches: Statical and dynamic.

In the static approach, regions are created on initiation of the game world. Regions are therefore determined by the developers and cannot be modified at runtime. The static approach must also be designed such that the workload does not exceed the resources that the server handling the region has, such as computational power or I/O capabilities. If the regions are under utilized, more regions could be handled by the same physical server. However if a region becomes overloaded, either more resources must be assigned to the region in form of hardware, or a dynamical approach could be used.

In the dynamical approach regions are also created on initiation, however, their proportions can be modified at runtime. There are different methods to perform the runtime modification of regions which will be discussed in the following section. Modification is usually performed in two cases:

- A server is overloaded. We can assign a subset of the current region to another server, and thus assign some of workload to a new server.
- A server is underloaded. We can assign a larger area of the game world to this server by combining two servers regions to one server.

This flexibility enables all responsibilities to be removed from particular servers, enabling them to be turned off if their resources are not needed and thus save power etc. In times with low usage such as during the night this can prove very power efficient. Additionally, server can be taken offline for maintenance, and so on.

### 2.6.2 Splitting Techniques

Splitting is to subdivide regions in order to distribute regions to separate servers. Dynamic regionalisation can utilize a splitting technique creating two new regions by splitting an existing region and assign a subset of the original region to a new server.

To accommodate splitting of regions during runtime, different techniques can be used. This section covers a variety of splitting techniques used for relieving a server of workload. By relieving servers of workload, a dynamic workload distribution of a game world is possible, this enables for more servers to be added if the workload increases and servers can be removed if the workload decreases. It is desirable to split the regions, such that the longest possible time is elapsed before having to split again due to the overhead involved in reassigning responsibilities. However, it can be extremely difficult to predict when the next splitting will occur, as players can move around as they please. In the following, different techniques to splitting are described.

#### Left Split

The left-split technique splits a region into two equally sized pieces, keeping the left side on the current server and assigning the right part to a new server as illustrated in Figure 2.7 (a). This technique is relatively fast because it is not computationally expensive to determine which region objects belong to. However, when a region is split the right region could be empty and the left full, and thus the overloading of the server is not reduced. In this case left-split will be performed again to ensure the workload on the initial server is reduced.

### Grid Based

Another approach is to divide a region into a grid of $x \times x$ tiles[14, 17], where $x$ is the region split factor. The region split factor is usual a low number of 2 or 3, to have a manageable number of smaller regions after the split, as seen in Figure 2.7 (b), (c), and (d). These smaller regions are then divided between the two servers, such that the workload is evenly distributed. Also when dividing these smaller regions they should be connected in a coherent group, such that the communication is minimised between the two servers handling them, e.g. the pattern should not be a checker pattern as illustrated in Figure 2.7 (e). The checker pattern would generate a lot of communication between servers as avatars move between the smaller regions and thus changes servers more often.

There are different methods to divide these smaller regions, which are described below:

**Left-to-Right (LR):** This method calculates the number of players within each smaller region and uses a scan left to right, row by row method as illustrated in Figure 2.7 (b), where it sums up the number of players in each subregion, until the value is higher than half of the total number of players in the region.

**Top-to-Bottom (TB):** The server uses the same initial method as the LR method, but the summing of the number of players is done by using a top to bottom, column by column method. The method is not depicted below.

**LR TB:** The server uses the same initial method as the two previous, but uses a left to right, top to bottom method as illustrated in Figure 2.7 (c).

**LR TB Diagonal:** This method is almost the same as the LR TB method, however instead of examining the whole set of regions it instead creates a diagonal, as illustrated in Figure 2.7 (d).

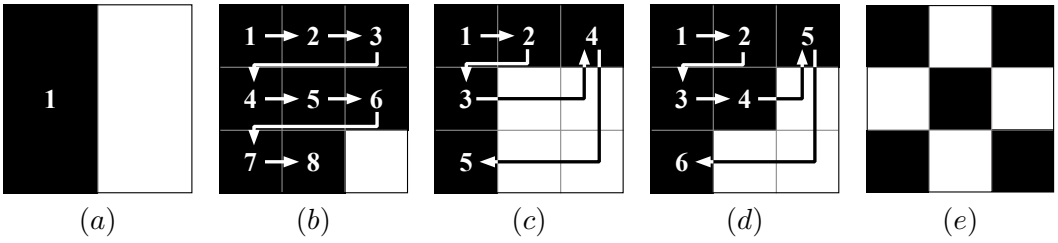



*Figure 2.7: (a) The left-split method, (b) Left-to-Right, row by row method, (c) Left-to-Right, Top-to-Bottom method, and (d) Left-to-Right, Top-to-Bottom with Diagonal region method. (e) Shows a checker pattern which should be avoided.*

When regions are divided and assigned to servers and a server subsequently becomes underloaded, an attempt is made to assign some of the smaller regions to another server. Since a server contains a subset of the regions, then it might not be a perfect square as illustrated in Figure 2.7. It instead pads the empty space with regions with a population of zero, and then uses the splitting techniques on the set. When it has found two subsets, that are evenly balanced, then it discards the added pad-regions and distributes the remaining

two subsets between the old and new server. However, if there is only one region left the region is split again.

The JoHNUM[14, 17] infrastructure uses an algorithm that combines the LR, TB, LR TB, and LR TB Diagonal methods in combination, but instead of using them all individually, they use them in correlation. That is, when using the first method they create a number of unique combinations, when using the next method they only use the combinations that is unique and not already in the set of unique combinations, and so on. This means, if the first method adds a unique combination of: $A = \{r_{00}\}$ and $B = \{r_{01}, r_{10}, r_{11}\}$, where $A$ and $B$ are sets of regions and $r_{ij}$ is a region and $i$ is the row cell element and $j$ is the column cell element. Then when the second method is utilised it does not examine the combination: $A = \{r_{00}\}$ and $B = \{r_{01}, r_{10}, r_{11}\}$, because it already exists in the set of unique combinations. Then the unique methods are examined one by one to determine the most equal division. When a division is found, which is below a certain threshold the division is made. This division is then used to split the region onto the two servers.

#### Behavioural

Another approach is to use the behaviour of the players to decide the partitioning. The behaviour could be decided as the position of the players or as the prediction of a pattern of the players flocking to a location. The behavioural approach can give a good estimate of the partition as it tries to predict the load. However this can be extremely difficult to realise if the game and players can move freely about in the game.

### 2.6.3 Aura-Based

Aura-based Interest Management (AIM) is another interest management scheme. AIM determines which information is relevant for a given object or player by specifying the interactions between them in a game world[16]. The world is defined as the space in which interaction occurs, and present in the world is the objects who performs interaction. An object can both be the observer and the observed object in the world. The key concepts of object interaction are as follows:

**Medium:** The medium is the way objects interact with other objects, these include visual, audio, touch, or text.

**Aura:** An aura is a subspace of the world which is centered around an object. Auras pertain to a specific medium, e.g. visual. Aura can have different sizes given the medium and object they are attached to.

**Focus:** An observer can focus on a given medium. Focus is the consequence of interest of an observer in the given medium.

**Nimbus:** An observed object has a nimbus, which is the objects interest of being seen, heard, or felt.

**Awareness:** Awareness level is the quality of service desired from an object given the medium. It can also be seen as the degree of message exchange between the two objects. The awareness level is negotiated between the two interacting objects.

**Adapters:** The adapter is the modifier for one objects focus, nimbus, and aura.

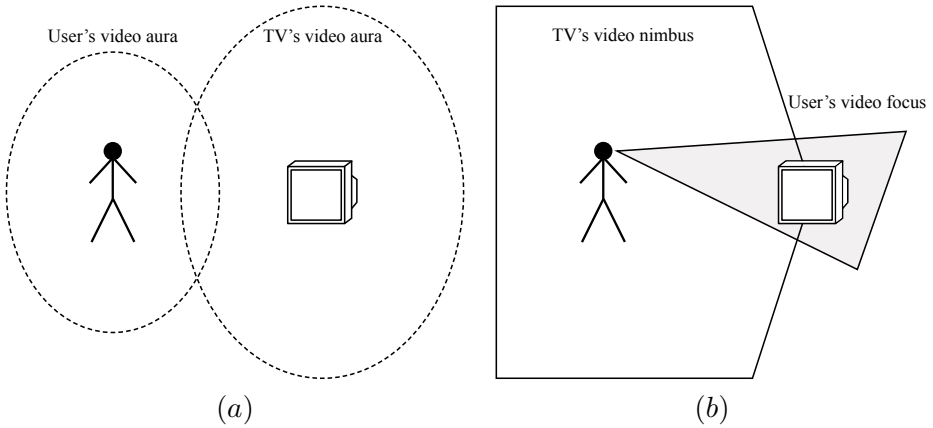



*Figure 2.8: (a) Two auras intersects each other demonstrating an interest for the both to interact, (b) the nimbus from the TV intersects with the focus of the user, giving a high awareness level.*

In AIM each object has an aura, focus, and nimbus. The aura is as illustrated in Figure 2.8 (a) a spherical aura that encompass the object. When two objects' auras collide, the two objects are interested in each other and therefore wants to interact. In Figure 2.8 (a) the auras of the user and TV collide, which therefore enables a visual medium between the two. In Figure 2.8 (b) the users focus is in the TVs direction and the TVs nimbus is towards the user, which results in a high awareness between the two. For instance, as in Figure 2.8 (b), the user and TV are infront of each other, allowing a high degree of visual exchange. If the user was behind the TV, the degree of visual exchange would be lower, since the TVs nimbus is not towards the user anymore. However if the user is standing infront of the TV and uses an adapter to turn off the TV, then the nimbus of the TV would be changed to a much smaller size, since the visual medium no longer is present.

AIM can be used in combination with region division or another method to divide information to a cluster of servers. However when such methods are applied in combination, additional consideration must be given to how information across servers is accessed and used. An example could be if a client is on a region border, the client can see into multiple regions at one time and this complicates interest management as regions could be assigned to distinct servers.

### 2.6.4 Alternative Aura-Based

The AIM method uses a plethora of concepts like aura, focus, nimbus, and so on, which can have a high computational cost. Simpler methods have been used with success[18]. The simplest form of aura interest management is to have a radius which represent the visibility of a player or object to determine which objects are interesting. This simple method is computationally cheaper than using the AIM method.

This simple form can be extended with different other techniques to make it more effective in determining which objects are interesting. Such an extension is the Predictive Interest Management (PIM), which extends the idea of auras to include a Predicted Area of Influence (PAI)[18]. A PAI is an area larger than an aura, which might be interesting for an avatar. The PAI is used to find the frequency of the message passing between avatars, which is increased when two objects get closer to each other.

In the PIM scheme every avatar has a PAI in addition to its aura. The PAI is found by using the current time, $t_{clt}$, and a future time $t_{clt+ft}$, which is used to find the distance that can be travelled with maximum speed for the avatar in the given time difference in any direction, as illustrated in Figure 2.9. The distance that can be travelled is the radius of the PAI. The PAI is the area in which the avatar can possibly exert its influence in the time window $t_{clt}$ to $t_{clt+ft}$. $ft$ is a system-globally defined positive number, thereby all PAI is found with the same time difference.

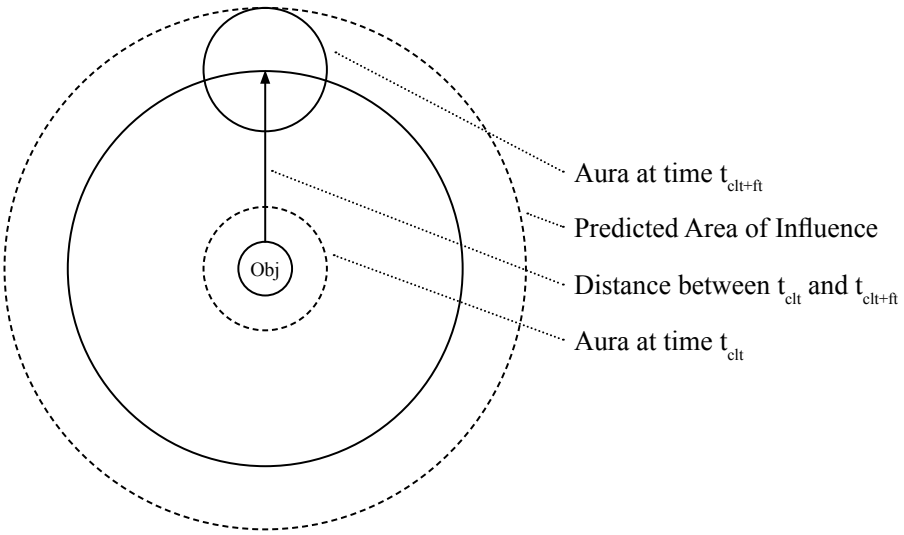



*Figure 2.9: Illustration of the PAI and how it is found.*

Three types of collision can occur in the PIM scheme:

**Aura - Aura:** Two avatars at near proximity that requires a high frequency of message passing to accommodate the increased interest between the two avatars.

**PAI - PAI:** If two PAIs has collided, then the frequency of the message passing is increased, but only as much as the PAIs overlap. Therefore, if a PAI and aura overlaps, then the message passing is increased accordingly.

**None:** Two avatars, which have no PAI collision have a standard low frequency of message passing.

Therefore if a collision between PAIs occur, there is the possibility that two objects may influence each other at some point in the near future.

## 2.7 Existing Games

Four games are taken into consideration as examples, both as technological examples, but also historical. First Diablo I & II from Blizzard Entertainment will be described, next EVE Online from CCP, and last Quake III Arena from ID Software. These four games are studied, because of their history in computer games, so the games will be described in a manner that reflects the objectives, architecture used to create a multiplayer experience, scalability of the architecture, and finally the amount of cheating that the architecture allows

for. Thereby the analysis of the games gives an overview of how popular games have used the technology available at release of the games and how they managed some of the major subjects of interest about creating a multiplayer game. Specifically, the analysis will focus on the network architecture used, how it scales with the number of players, and what the possibilities were for cheating.

### 2.7.1 Diablo I & II

The Diablo series is a game from the RPG genre in which hordes of monsters attack the player as illustrated in Figure 2.10 (a) and (b). The player has various methods of killing these monsters to gain experience points (for this reason it is also sometimes called a hack-and-slash game). As the player gains experience points, her avatar rises through levels, enhancing the powers of the avatar.


(a) (b)

*Figure 2.10: (a) Diablo with Hellfire expansion (b) Diablo II screenshot.*

The games takes place in a dungeon environment, where the monsters spawn in random locations of the dungeon. The dungeons are randomly generated to enhance the replayability of the games.

The games was developed at Blizzard North, which is a part of Blizzard Entertainment. Diablo I was released in 1996, and Diablo II was released in 2000.

#### Objective

The Diablo series games have two major game-modes. The first game mode is Player vs Player (PvP), and the other is Player vs Environment (PvE). In the PvE part of the game the objective is to kill computer controlled monsters (NPCs) while going through the storyline and gain experience points for the player controlled avatar. This part can be played either alone or in cooperation with other players. The PvP mode enables players to battle against other players in addition to monsters.

#### Architecture

The network architecture differs in the two Diablo games:

**Diablo I:** Diablo I uses a peer-to-peer topology for the network. In addition to this, avatar information is stored locally on the players computer. Diablo I supported at most 4-players playing together.

**Diablo II:** In Diablo II an addition was made, such that two modes of game play were added, each utilizing a different type of network architecture; closed and open battle.net (Battle.net is the system used for matchmaking throughout the large Blizzard titles). Open battle.net is used for peer-to-peer games, where the players avatar information is stored on the client akin to the way Diablo I worked. Closed battle.net limited the amount of cheating by using a client/server architecture, in which each server machine would support a number of player groups and the where the avatar information is stored on the servers. Diablo II supports 8-players multiplayer games with PvP and PvE, like in Diablo I.

#### Scalability

Diablo I and II are limited to 4- and 8-players per game respectively. Diablo II saw the addition of the closed battle.net mode, which uses a client/server architecture. This solution is also quite scalable in the sense that additional groups of players could play simultaneously by adding server machines. Each group is placed in their own game world. This technique is akin to sharding which is commonly used in MMO games. Thus, while each game world can only contain a maximum of 8-players, many thousands can play simultaneously.

#### Cheating

The peer-to-peer solution, where avatar information is stored on the client has is problematic because players can alter their avatar as they please and thus create an unfair advantage. This is a problem in Diablo I and Diablo II in open battle.net mode. However this problem was addressed in closed battle.net mode, because a server maintained by the game creators was introduced to store the avatar information, and handle information exchange. This made cheating significantly more difficult[19].

### 2.7.2 Quake III Arena

Quake III Arena takes place in a futuristic surrounding, where players are fighting against each other in an arena, as illustrated in Figure 2.11 and was developed by ID Software. Players have all sorts of devastating weapons, like: Rocket launcher and lightning guns, to try to incapacitate each other. The fighting arenas are often small in size with weapons scattered all around the arena. The physics in the game is not realistic as players can gain speed by jumping, turn in mid air, and jump far distances. Quake III Arena was designed from the start with the major focus on the multiplayer part of the game. The singleplayer part is essentially the gameplay from the multiplayer part with the computer acting as other players avatars, occasionally interspersed with a little background story.

#### Objective

The objective of the game is to eliminate opposing avatars, which are either controlled by a computer or another human player.


(a) (b)

*Figure 2.11: (a) Screenshot from Quake III Arena[20] and (b) screenshot with the heads-up-display.*

### Architecture

Quake III Arena uses a client/server approach, where clients communicate directly with a server. The server updates clients at a fixed time rate, whereas clients can send updates to the server at different time rates.

### Scalability

Quake III Arena supports up to 32-players on one server, but where the player number differs in the different game modes available in the game. As such, the scalability of Quake III Arena is very similar to that of Diablo II, however Quake III Arena allows for more players in the same game world.

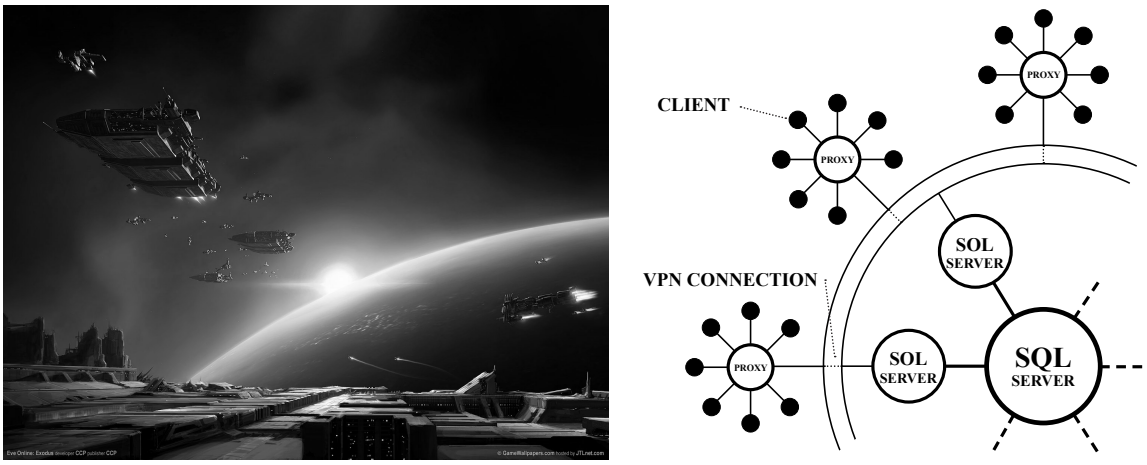
To maintain consistency amongst all players, all information is send to all players. So a whole game state is send to all players giving a global consistency amongst all players. This makes sure that no player knows more about others. Even though there is a latency difference amongst players, the game tries to predict this latency, such that it is consistent on all player machines.

### Cheating

Cheating is possible in Quake III Arena, but additional measures have been taken to make cheating difficult. Cheat prevention measures have been implemented by using "pure servers", where the client is only allowed to use specific information within package files. Package files contains all game relevant data, like graphics, levels, sprites, sounds, and so on. The package files are like .zip files, when unpacked the package file contains other files and folders. These package files are checked for inconsistencies, such that users can not change the data within these packages and therefore cheat. A small independent program called "PunkBuster"[21] is also available for Quake III Arena, which scans the clients computer for programs, memory, and inconsistencies in the game data, that could give the client an advantage.

### 2.7.3 EVE Online

EVE Online is a MMO-RPG game created by CCP[2]. The game takes place in a space environment, where players control a customizable avatar which can learn a number of standard trades, like: Mining, manufacturing, combat, and so on, but also has a number of ships they can customize to fit their needs, as illustrated in Figure 2.12 (a). The game provides a very large universe, where players can roam in. Players can be a part of a *corporation*, which forms an alliance between players. Corporations can give players roles, that exceeds the standard trades. A corporation can, if strong enough, control solar systems, space stations, and so on. This however can lead to wars between corporations or even alliances of corporations, that can finally yield in big battles between the fighting parties.




(*a*) (*b*)

*Figure 2.12: (a) Screenshot from EVE Online (b) The network architecture of EVE Online.*

#### Objective

The objective of the game is to learn new skills and trades to improve ship and equipment. The game also has smaller objectives like being a part of corporations and alliances and thereby help these in battles, trade, mining, and manufacturing.

#### Architecture

EVE Online uses a single shard architecture[3], such that players can interact between all avatars and create a macro economy of user created items. EVE Online is built using a client/server solution where the game universe is distributed onto several servers, as illustrated in Figure 2.12 (b), however there is only one game world. The different solar systems in the game world are distributed on different to *SOL* servers, which are connected by one MS SQL Server machine. Some of the solar systems are more crowded than others, so some solar systems share SOL servers, whereas others have an entire SOL server for themselves. Players tend to move amongst solar systems, so proxy servers keep track of player movement between solar systems.

### Scalability

A SOL server can contain one or several solar systems from the EVE Online game world (which consist of more than 5000 solar systems). The SOL server cluster therefore scales with the size of the game universe, and some solar systems can be specified more power to accommodate the popularity of a given solar system, like the trading hub of Jita, where approximately 1500-1800 players trade at its peak population. The only limit of scalability for EVE Online at the moment is the centralized SQL server, which is not distributed. The SQL server maintains all relevant information in EVE Online, like avatar information, items, and some game logic. The SQL server has ∼2500 transaction calls a second with a size of about 1.4TB of data[3]. The SQL server therefore relies on hard programming rules, software optimization, and hardware.

To maintain consistency in EVE Online, where the world is partitioned onto several servers, CCP has used localised consistency, where all players only know about other players within the solar system, that they currently are a part of. Since there can be thousands of players in one solar system, the interaction amongst the players must be initialised by clicking on other players, so all the information that is relevant for the player is transferred, when not initialised only a small amount of information is transferred. However players can interact in chat between solar systems. This is done by subscribing to either a channel (chat room) or by initialising a private chat session. All solar systems know about their neighbour solar systems, so a player can move between solar systems. This is done by using jump gates, that during the travel process changes server for the player, which creates an illusion of one consistent game world.

### Cheating

In EVE Online the game logic runs on the server, which means that the players cannot manipulate avatar info, like avatars position, money, equipment, and so on. All avatar information and trade of material are stored at the SQL server, which means that the players cannot manipulate this.

## 2.8 Summary

This chapter analysed the structure, tasks, and the game loop of a game engine, which components was needed in the game engine to make a game. We also looked at how a game loop could look like, which order the different components needed to be run in, e.g. that the movement of objects should happen before the collision detection was done.

We also looked at which requirements there is in a MMO. This included consistency, latency, and scalability. We motivated these requirements for a MMO and why they were important to analyse before starting designing a MMO solution for a game.

Next three different network architectures was examined, which were: Peer-to-Peer, Client/Server and lastly a generic MMO architecture. It both detailed the historic usage of the architectures in games and some of the pros and cons of using such architectures, e.g. peer-to-peers problem with NAT-traversal and cheating, but the game could be distributed onto all the player computers. The client/server approach addressed the cheating problem by storing all avatar information on the servers and made all the game logic calculations for the clients, but had the problem that it required a lot of dedicated servers for the task. The generic MMO architecture looked at how a client/server architecture could be used in a

MMO game, how the layers was in such an architecture, e.g. how application and persistence layers should work together to create a consistent view of the game world. It also looked at load balancing and server clustering in such an architecture.

Different types of interest management schemes was examined, which were: Region division, both static and dynamic, aura-based, and an alternative aura-based. The region division covered how a game world could be made into static sized regions, which could be distributed onto several servers, but this approach had a problem with crowding. Dynamic region division has a partial solution for crowding, where an overloaded server can be partitioned into more smaller regions and then distributed onto servers, which is done on runtime.

Then the messages between the clients was discussed, how to send the game state, how to differ amongst new and old messages. The delta-encoding was therefore looked at, such that smaller packages could be send amongst the client and server. The usage of UDP and TCP was examined, what the pros and cons were with these two protocols.

Four games were described, which was Diablo I & II, Quake III Arena, and EVE Online. The Diablo games mainly emphasized the problems of creating a network solution for a game, which was insecure and also the consequences of this. They also described a potential solution to the cheating problem, how a closed realm of servers could remove the manipulation of avatar information. Lastly the EVE Online section describes a scalable solution for a MMO-RPG game, where thousands of players can interact and trade with each other.

Chapter 3

# Towards a Distributed Game Engine for MMOGs with Crowding

This chapter outlines the path of this project. It gives an overview of how the MMO-FPS genre relates to the analysis, a sketch of the solution to scalability and crowding issues in a MMO-FPS game, and describes the methodology used in the project.

## 3.1 MMO-FPS

As this report deals with the design and implementation of a MMO-FPS, the following section summarises the important aspects touched upon in the previous chapters and discusses them in the context of a game which combines the MMO genre with the FPS genre.

Games from the FPS genre, such as Quake III Arena have the strictest requirements to latency from the genres treated, as mentioned in Section 2.3. This is in opposition to for instance the MMO-RPG genre. The MMO-RPG genre is typically point-and-click, that is, a player expresses explicit interest in interacting with another player or object. This greatly simplifies interest-management. Additionally, RPG games typically have game mechanics which do not depend aiming at other players, and only depends loosely on their positions in the game world. The FPS genre on the other hand, requires that players aim for targets, e.g. by pointing a crosshair at them. This requires players to have updated knowledge of other player positions at all times, which in turn, requires information to be dissipated with a low latency. Combining the FPS genre with the MMO genre constitutes a major challenge, because of the additional strain on the network architecture that is inherent to MMO games. Specifically, the combination of the low-latency requirement and the scalability requirement is a major hurdle. Thus, very few true MMO-FPS games have been successfully developed.

A client/server network architecture can help remedy the problem of cheating and is the preferred choice of architecture in modern MMO and FPS games, as mentioned in Section 2.7.1. In a MMO game context, it also ensures the game producers a larger degree of control, e.g who is playing. Thus, a client/server architecture is the preferred architecture as discussed in Section 2.3.2.

The network topology can vary in numerous ways, as mentioned Section 2.4. The topology should attempt to minimize latency by minimizing the number of hops each piece of information must travel, and also afford scalability of the system in its entirety. The appli-

cation server layer may consist of several sub-layers in order to achieve good scalability, e.g. proxies, load-balancers, and so on.

Interest management plays a key role in ensuring scalability for MMO games, as mentioned in Section 2.7. This was because the game world or players can be distributed onto several servers, which handles each subset of the game world themselves. For example, when partitioning the game world into subsets and distributing these subsets onto several servers, then the players would be distributed onto several servers given that they are spread out in the game world. However a phenomenon was mentioned in Section 2.2.3, which was crowding. If all players moved to one specific point in the game world, then it would be the same as a single-server solution, however this is a worst-case scenario. Thus, it is necessary to apply an efficient interest management scheme.

## 3.2 Solution Idea

We would like to arrive at a solution which utilizes and builds upon the ideas and choices described in the analysis chapter in order to cater to the requirements of a MMO-FPS game. This solution will focus on subdividing the game world dynamically which aims to minimize the impact of crowding and enables massive scalability. However since this is the first report in a series of two, then this solution idea is the goal after a single-server and a multi-server (using static geographic regionalisation) solution has been tested.

The special subdivision of the game world will be coupled with novel interest management techniques to further improve scalability and tolerance against crowding.

### 3.2.1 Region Division

The game world will be subdivided by using a Region-Based approach, where the game world is subdivided geographical. However the players avatar will also use an aura to define the interest of the avatar, thereby giving the objects of interest. The usage of the techniques are described below and illustrated in Figure 3.1 (a).

**Geographical:** The game world is subdivided into two types of regions:

- **Region:** A region consists of a part of the game world. This allows for local play between the clients within the region. A region can be further subdivided if the server handling the region is overloaded, and various division methods can be applied.
- **Atomic Region:** A region consists of atomic regions, which are never subdivided. A region can be either bigger or of equal size as a atomic region. All atomic regions are of same size and creates a logical grid of the game world. This logical grid is used as a measurement to find Logical Interest Region (LIR), which are avatar information that the client is interested in and is illustrated in Figure 3.1 (b), by using the aura to create a rough area of interest.

**Aura:** An aura is an area of interest for the client. If the aura spans over several regions, then these regions are of interest for the client. LIR is found by using the aura and the atomic regions. The aura can either be a sphere or a cube, as illustrated in Figure 3.1 (b).

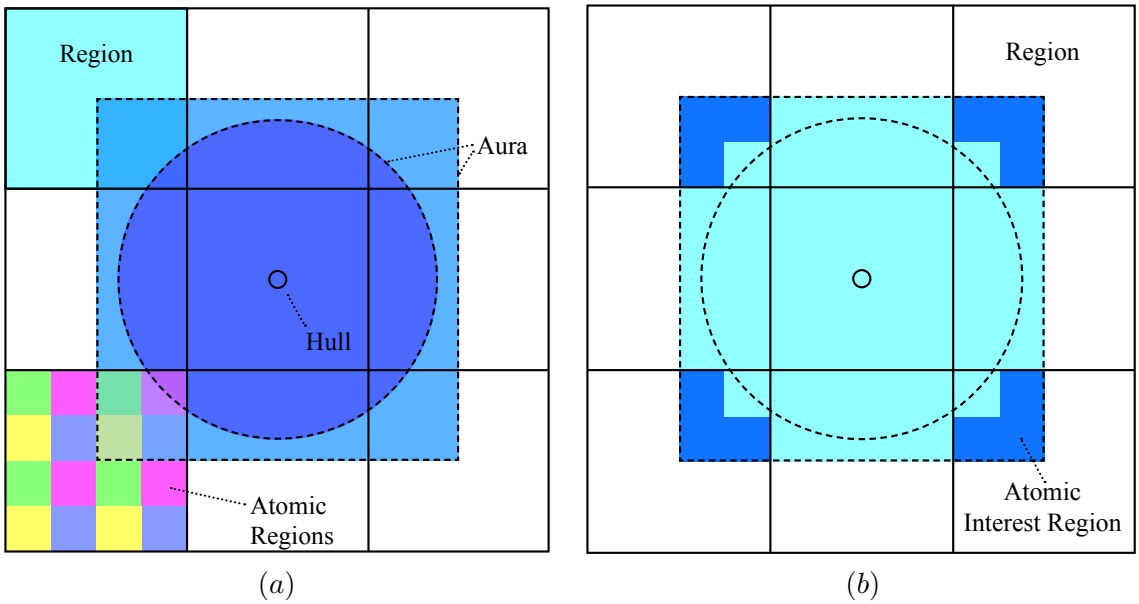



*Figure 3.1: (a) Illustration of the region- and aura-based splitting of the game world (b) Illustration of the message passing from the proxy by using atomic regions from the server.*

The solution will be accompanied by a testbed which enables performance testing of different variations of the solution. This will enable a thorough evaluation of the novel ideas described throughout the project.

## 3.3 Methodology

During this project a basic implementation of a single-server and a multi-server solution will be designed and implemented. The multi-server implementation will have a static region division as opposed to the dynamic region solution sketched in the section above. This was decided on the basis of the time constraint associated with this report. These implementations form the basic platform for implementing the ideas sketched in the above section. A future project will implement these ideas. Additionally, this report contains an evaluation of this basic platform which can be used to compare it to the evolved implementation. The evaluation of the platform is performed using a *test game* developed specifically for this purpose.

## 3.4 Test Game: Rock Pounder

To be able to test the architecture devised, a test game called Rock Pounder was created. The concept, motive, and game mechanics of Rock Pounder are described to give an insight in the game. Then a description of the architecture, how the program execution of the main thread is structured, thereby creating an insight in how the actually game works.

### 3.4.1 Motivation

Rock Pounder is a game created specific for testing the MMO architecture created in this project, however there are already existing games that can be modified to be used for this for

this purpose. These games include amongst others Quake III Arena, which was described in Section 2.7.2. There were two reasons for building a new game for testing, instead of using an already existing game:

1. Creation or modifying time of a new or existing game.

2. Size and content of the game.

The first point speaks both for and against creating a new game. A new game had to be created from scratch, which meant that a game engine needed to be created, however if using an existing game there was a big reliance on existing documentation. If such documentation was lacking, then the process of modifying the game could take a considerable amount of time. There was also the point of the computation the game needed to maintain the game, this is physics, game logic, network, and so on. If a new game was created then there was already knowledge of which elements of the game that was computational heavy, and only needed elements could be implemented creating a minimalistic game. A smaller game with fewer computations could be simulated in greater quantity on a single machine, than e.g. a complex game, and since the game should be tested in a MMO environment, then a huge number of test clients were needed.

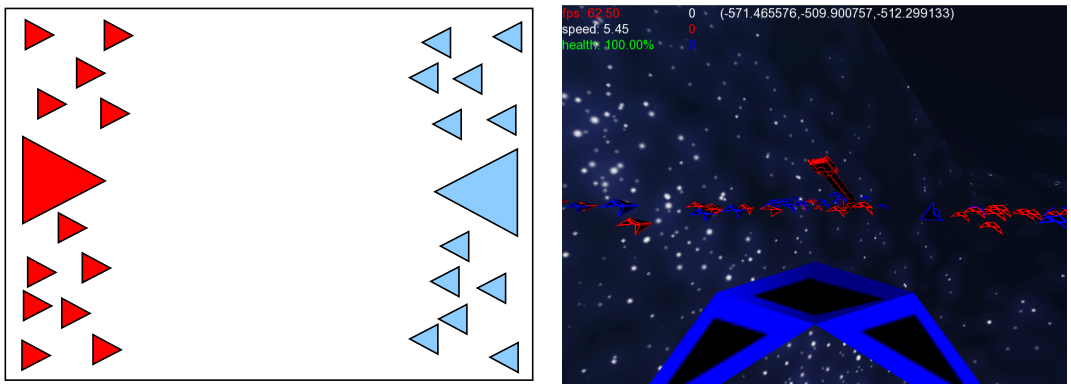
The second point is about the size and content of the game world. Given a game like Quake III Arena a new game world had to be created, which is big enough to have a huge number of clients. It therefore also needed a lot of content (Walls, houses, and so on) for this world to be created and the amount of content reduces the speed of the game. Therefore if a new game was to be created, there would be more control of the requirement for the size and content. The game content could also be created with the amount of detail that would be the minimum requirement. The last part of the second point is the game genre. The game genre has to be FPS, since that was the requirement given in Section 1.3.

Given these two points it was decided to create a new game, which could fit the requirements of being a FPS game, greater understanding of all the key components within the game, minimal computations needed by the game, and lastly the ability to create a big game world with as little content as possible. The creation of the new game would also make it possible to make simulated clients easier, since the knowledge of all the key components was available and therefore components like graphics could be easily disabled. There will therefore be created two versions of the game: A playable version and a simulated version, where hundreds or even thousands of clients can be simulated simultaneously.

### 3.4.2 Game Concept

Rock Pounder is a 3D MMO-FPS which takes place in vast space. The setting is an epic battle between the blue and red team which both have arrived in huge numbers; carriers and small fighters. The fighters are small, fast, and volatile ships, that spawn near the big, robust, and slow carriers. Both teams are trying to take control of the known universe and the only way to resolve the conflict is to eliminate the opposing team, as illustrated in Figure 3.2 (a) and (b).

The goal with Rock Pounder is to create a huge game world with an epic battle, where players controls one of the many ships and have to rely on teamwork and skill to help their team win the battle.




(a) (b)

*Figure 3.2: (a) Concept drawing and (b) screenshot from Rock Pounder.*

### 3.4.3 Game Mechanics

This section briefly describes the game mechanics of Rock Pounder. This includes how the player moves, shoots, scoring, level, Head-Up Display (HUD), and so on.

**Movement:** A player moves her ship by using the mouse and keyboard. The viewport is centred by the mouse pointer which handles pitch and yaw of the ship. The keyboard handles throttle for movement forward, backwards, left, and right and thereby gives the player the feeling of flying in space, where all three dimensions are used in the game world, called the $x$-, $y$-, and $z$-axis in a 3D environment.

**Shooting:** The player shoots by pressing the mouse button. Shooting is bound by the regeneration period between shots. The shots are slow laser beams that travel at a specific speed and a certain distance before disappearing. If a laser beam hits a target, the target will lose 42 Health Point (HP) of the maximum 100 HP an avatar can have. If an avatar reaches 0 HP or less it dies and will respawn at a carrier.

**Score:** There are two types of scoring in game. First, a player scores points by killing opponents yielding one point. Second is a total points for the team, which is all the opponents killed.

**Head-Up Display:** The HUD comprises of a health bar, laser beam regeneration bar, and the score screen. The score screen shows the points the player and the team has achieved.

**Spawning:** When a player has been eliminated, the player will spawn at its carrier and be ready to fight again.

### 3.4.4 Architecture

The architecture of Rock Pounder's engine is a standard game engine structure, which consist of a single main thread, as illustrated in Figure 3.3. The structure of the code that the main thread executes consists of initialization, which initializes the game, input, and network handlers which can handle incoming events. These events are handled concurrently with the game, and stored in a buffer until the main thread can handle these events. The

main thread has a loop, which handles incoming events from network and input. Afterwards in predictions the objects positions are updated to give a fluent feeling of the game between events received. Lastly it renders the graphics on the screen. This loop is executed frequently to get a fluent feeling of the game.

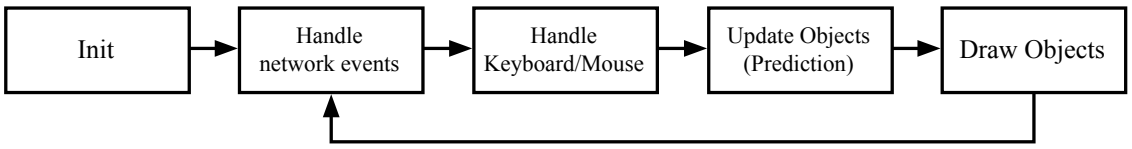



*Figure 3.3: Structure of the main loop on a game client.*

The server has a similar main loop, but it has no input to handle, does not need to draw the objects, and does not predict where objects are. It is however responsible for calculating where the objects actually are in the game, and how they are affected by other game events.

## 3.5 Summary

The chapter covered the motivation for creating a game from scratch, instead of using an already existing game, like Quake III Arena. This was because creating a new game would give a basic knowledge of every component and calculation within the game, but also the ability to minimise the computations needed to maintain the game logic. It also gave the ability to quickly modify the code, since the code now was basic knowledge, and gave the possibility of creating two versions of Rock Pounder: A playable version and a simulation version, where the latter is used to simulate a large number of players in one instance of the application.

The chapter also covered the basic game mechanics and idea of Rock Pounder, which is a MMO shooter in space. It introduced the two main types of units, fighters, and carriers. The small fighters spawn near carriers and are fast and volatile, whereas the carriers are slow and robust with a lot of fire power. In Rock Pounder there are two teams that must eliminate each other in a big game world. Last a brief explanation of the architecture, which is a standard game engine design with one main thread for all the game logic.

Chapter 4

# Design and Implementation

This chapter will detail the design and implementational decisions made in the development process for Rock Pounder. It will focus on the network architecture devised. It will also describe the protocols involved and the interest management scheme chosen.

## 4.1 Architecture

This section covers the design of the client and server structure for handling messages and game logic. The architecture for the servers and the communication is described. Specifically, this section is tasked with describing how the clients communicate to the servers. The first part of this section will pertain to the single-server solution, while the second part will detail the multi server devised. The multi-server section will focus on how the servers handle the distribution of messages and game engine calculations.

The section will not consider persistence of information. That is, it will only pertain to the client and application server layers of the generic MMO architecture.

### 4.1.1 Single-Server

The architecture for the single-server is a standard client/server architecture. A single-server process exists, while clients communicate directly with it. In relation to the generic MMO architecture mentioned in Section 2.4, the application layer only contains a single server. This server is responsible for all objects in the game world, as illustrated in Figure 4.1 (a).

With a single-server approach, many of the concerns that exist regarding division of responsibilities are already solved, since there is only one server to handle all the clients. The design of the single-server is as seen in Section 2.3.2. The advantage of this approach is simplicity.

The restriction of a single-server architecture are directly imposed by the the resources which are available in the machine on which the server process is running. It quickly becomes economically infeasible to increase the machine's resources as the workload grows due to the non-linear performance/cost ratio of hardware.

Additionally, the single machine acting as the server represents a *single point of failure*. If the machine crashes, every player will be interrupted. It may be possible to alleviate the

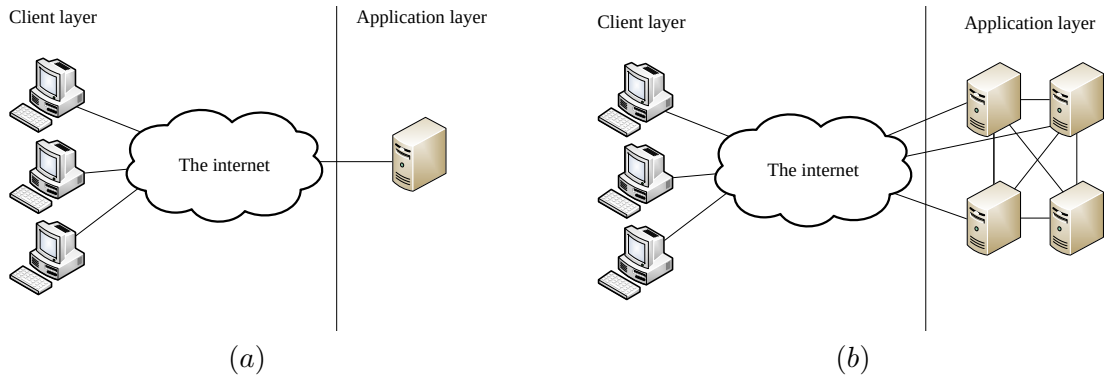



*Figure 4.1: Illustration of the single- and multi-server architecture*

issue by having all data replicated to a second machine and then do a fail-over to the second machine if the first one crashes.

### 4.1.2 Multi-Server

The multi-server architecture uses the same basic techniques as the single-server architecture. However instead of using only a single machine in the server layer, it uses several server processes which can then be distributed on several machines. This section therefore covers the usage of this multi-server architecture with the region division technique mentioned in Section 2.6.1. The game world will be partitioned into pieces of equally sized regions. Each of these regions are statically assigned to a server which handles its region and the game objects residing within its region of the game world. The region division technique is therefore a static geographical partitioning.

Clients connect to a server in the application layer, as illustrated in Figure 4.1 (b). These servers are also connected to each other.

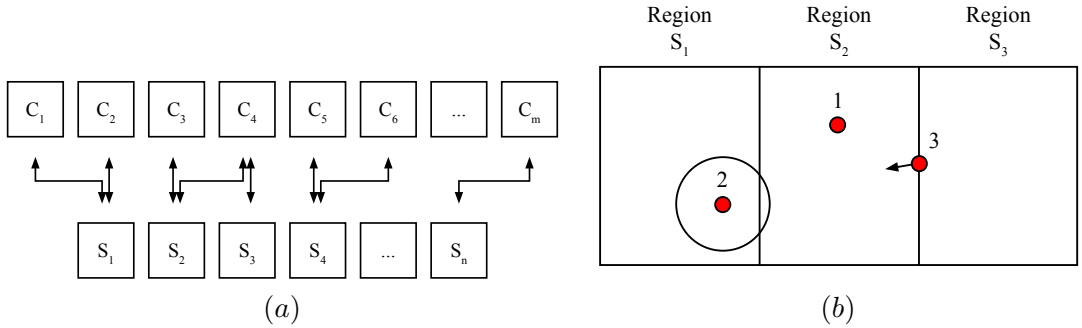



*Figure 4.2: (a) The multi-server architecture and the communication between the clients and server, (b) The static division of the game world and objects within.*

The network architecture is illustrated in Figure 4.2 (a), where there are several servers for the clients to connect to. Client are typically connected to a single server, but it can occur that clients have a connection to several servers. The single server approach can be

considered a special case of the multi-server approach. In this case, there exists a single all-encompassing region, handled by a single server.

In a multi-server scenario, the server has additional responsibilities to the ones mentioned previously:

- The server must notify clients that they are required to connect and receive updates from other servers as their characters's aura overlaps with regions adjacent to the one in which the avatar currently resides. The scenario is illustrated in Figure 4.2 (b), object 2.
- The servers must also coordinate internally when an avatar moves from one region to another. That is, if a client has entered a region managed by another server, the old server will tell the new server, that it is now responsible for the client, as illustrated in Figure 4.2 (b), object 3.

This architecture has the advantage of being able to scale with the number of clients by adding more servers. Also, it is more robust to single machine crashes. The clients connected to the server residing on a crashed machine will be disconnected, but everyone else should be able to continue playing, as long as they do not attempt to enter the region for which the servers on the crashed machine was responsible.

The additional responsibilities that are given to the servers in a multi-server architecture complicate the design and implementation of these considerably. A lot of additional coordination and, in turn, communication is required to have a functioning multi-server solution. The following sections will detail how this is implemented.

## 4.2 Protocol

This section will describe the protocol designed for use in communicating game state between servers in the application server layer as well as between clients and server. The description will include design choices, description of packages, their meaning, and how different events in the game are handled. The first part only considers the single-server architecture and the following part will extend this to support multiple servers.

The protocol is designed such that the server is always the authority on as much as possible, e.g. game logic. In a sense, the clients are designed as *thin-clients* as they are only responsible for accepting user input and rendering objects on the screen. This is done in order to minimize the opportunity for cheating, e.g. cheating by altering the client program such that the player will have an unfair advantage.

Both UDP and TCP is used simultaneously to communicate game state. TCP is used whenever the packet must be guaranteed to arrive. These packets include creation of most objects and removal of objects from the game world. UDP is used for location updates and also for creation of shots. Thus, messages are *not* idempotent, e.g. each time an object is created, exactly one message is sent. Messages sent over UDP are aggregated into packets, minimizing packet overhead. For example, during the main game loop in the server, messages are queued for transmission, aggregated, and sent to recipients at the end of the game loop.

The *qport* is a 4-byte integer which is prepended to every UDP packet sent from clients. This is done in order to circumvent an inconvenience that exists when client are behind a NAT-device. When a UDP packet arrives at the server, it must determine from which client

it originated. The source IP-address can not be used on its own, because two clients may be behind the same NAT, sharing source IP-address. Additionally, the source IP-address and source port of UDP packets cannot be used to uniquely identify clients because NAT-devices can decide to change the source port of packets coming from a client at their own discretion. Thus, the qport is needed to identify from which client a UDP packet originated. It is named "qport", because it was initially conceived in the network protocol of the Quake III Arena game[22].

Table 4.1 gives an overview of the messages used in the protocol.

| MessageType | Description | S → S | S → C | C → S |
|---|---|---|---|---|
| ObjectUpdate | Contains position and orientation for a given object | × | × | |
| ObjectInfo | Contains all info about an object to be created on the client | | × | |
| ObjectDestroy | Contains an object id which has to be removed | × | × | |
| Welcome | Contains users qport and id | × | × | |
| ClientUpdate | Contains information from clients what actions they have done | | | × |
| Login | Identifies a new connection. | × | | × |
| ServerWelcome | Contains information to sync server time when a server has connected | × | | |
| ConnectToNearBy | Contains information of which server the client should connect to | | × | |
| TransferControl | Contains information about an object which moves from one server to another | × | | |

*Table 4.1: Messages used in the protocol. The three right-most columns denote possible senders and recipients of the individual message. 'S' denotes servers and 'C' denotes clients.*

### New Player

Communication is initialized by the client opening a TCP connection to the server. It immediately sends a *Login* message, containing the player name. It subsequently receives a *Welcome* message, containing information, e.g. the team to which the connecting player has been assigned, the object ID number, and the *qport*.

This is followed by information about all the other objects that the new client should know about, as mentioned in Section 2.6, which contains information on how to determine which objects a client should know about.

Finally, other clients are notified about the new player. They receive a message describing the characteristics of the avatar in the game world, this includes score, location, and so on.

### Game Loop

In the context of responsibilities, the single-server and multi-server designs have several things in common. The client renders a visual representation of the game world and handles

input events, which is illustrated in Figure 4.3 (a). The client has two ways of handling communication:

**Receive:** When receiving a message from the server it is sent to the message handler, that translates the message to be used by the game. The translated message is added to the ring buffer, that is emptied at a given frequency and used in the game logic. E.g. the message could be a new position and orientation for the player, which is then handled by the game logic such that the player is positioned and oriented as the servers view of the player.

**Send:** When an event occurs, e.g. pressing the throttle button, the event is added to a message buffer, which is send at a given frequency. Then the buffer is converted into a message, which is then send to the server.

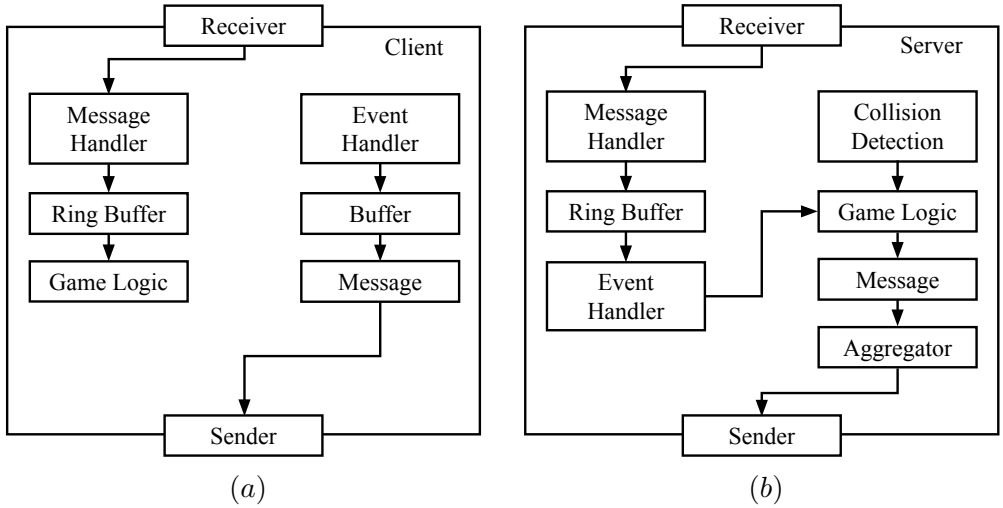



*Figure 4.3: (a) Client structure when receiving/sending messages, (b) server structure when handling messages.*

The server handles all the game logic and collision detection, which are needed by the clients. The server handles the communication as illustrated in Figure 4.3 (b) and as described below:

**Receive:** When receiving messages from the clients the messages are send to the message handler that translates the messages and places the messages in a ring buffer. The ring buffer is emptied at a given frequency and passed on to the event handler. The events from the clients are then send to the game logic, e.g. if a client sends a throttle forward event then the server executes the throttle forward event by using the game logic that handles such an event.

**Send:** The structure also has a collision detection step, as illustrated in Figure 4.3 (b), that uses the location of the different objects. If a collision should occur between the two objects, then a message is added that corresponds to the collision type, e.g. if a shot hits a player, then the player should lose life or die. The messages are then aggregated to form one big message, which is then send to the player.

During regular play, clients continually sends input received from the user to the server. This includes movement and shooting. The input is sent to the server in a bitstring format, e.g. user wishes to "move right", "move forward" and "shoot". The bitstring is interpreted at the server for each iteration of the game loop. *Dead reckoning* is performed both at clients and at the server in order to minimize communication and simultaneously gives for smoother trajectories on players' screens. Dead reckoning works by utilizing the last known information about an objects movement to predict where it is located at the current time. Clients receive updates about object positions at intervals and these updates are timestamped by the server. Clients then calculates a velocity vector from two subsequent updates and applies this velocity vector to the currently known position of every object, until new information about the objects is received. Dead reckoning is also used on the server to make assumptions on where players have moved since an update was received from them. The prediction performed on the server is however authoritative. This is to ensure that clients cannot withhold updates, relying on the server to predict their trajectory and update other clients based on this prediction, and then suddenly send updates which invalidates the server's prediction.

If the server receives a packet signalling a clients intent to shoot, it will transmit an *ObjectInfo* message to clients describing the creation timestamp and velocity of the projectile. Due to dead reckoning, each projectile needs only to be sent to each client once. The client then predicts the trajectory of projectiles.

When a player is hit, she receives a *ObjectDestroy* message to signal that the projectile that she was hit with no longer exists. It is important to emphasize that this is not to notify the client that a ship has been destroyed, but to notify the client that a projectile has hit the ship. The messages includes the ID of the ship that was hit as well as the new health status of the ship. If the projectile does not acquire a target within their lifespan, clients automatically remove them from the game world.

When the health of a player reaches zero, their location in the game world is immediately reset to a designated *spawn point*. Spawn points are where player avatars are also located, when they initially join the game.

Dead reckoning at the server is utilized to determine when it is time to update the clients. Due to the fact that the clients and the server can do dead reckoning in exactly the same way, on the same data, the server can calculate how much the prediction on the clients deviates from the actual position known by the server. As soon as this deviation reaches some specific threshold, the server will transmit updates to the clients. This helps minimize the amount of data transmitted from the server.

#### Player Quits

When a player decides to quit the game, she simply disconnects the TCP connection. This triggers the server to send a *ObjectDestroy* message to all clients signalling that a player has left.

### 4.2.1 Multi-Server

The communication between the client and the server is almost unchanged in the multi-server design. There is, however an additional *ConnectToNearBy* message sent from the server to the clients. This occurs whenever a client's avatar is nearing the boarder of the region in which it currently resides. It instructs the client to connect to the server responsible for the near by region, such that it can receive updates on events taking place within the

adjacent region. This is to ensure that players looking into another region is able to see events taking place in that region. The client is not able to send updates directly to the new server.

As mentioned much coordination must take place between servers in order to keep the game world consistent for their clients. When a client crosses into a new region, the server handling the region from which the client is leaving will notify the server handling the new region of the transfer by using a *TransferControl* message. The reason for this level of coordination between servers is to minimize the potential for cheating. For example, it is important that only one server has the authority over a given object at any given time. Also, if the client sends updates to two distinct servers it may send contradictory updates, creating an inconsistent game world.

## 4.3 Inter-Server Communication

In the implementation of the inter-server communication different problems occurred. One of them was when a client was on the border of two regions. Then the client would keep changing server, which in the end could make the client stop responding. To solve this problem, borders were created as illustrated in Figure 4.4 with the *aura*, *out*, and *in* borders:

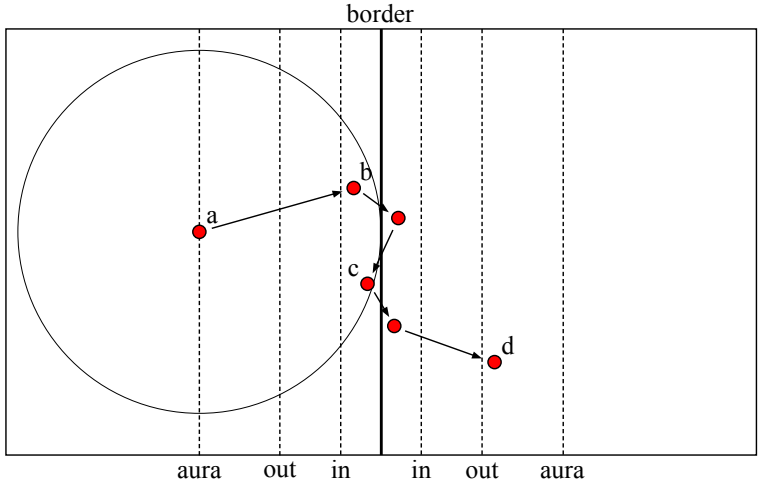



*Figure 4.4: Illustration of the three borders:* Aura*,* out*, and* in*, where a player avatar (a)-(d) moves across the different borders.*

**Aura:** If a clients aura intersects a new region, then it is an indication that the client is interested in the new region, as discussed in Section 4.1.1 and as illustrated in Figure 4.4 (a). The old server will then notify the client which will connect to the new server. The new server will then send updates to the client about the whereabouts of the clients on the new server, but will ignore update messages from the new client, because the client is does not influence the new server.

**In:** When a clients hull passes the *in* border, then it is an indication that it might cross the border between regions, as illustrated in Figure 4.4 (b). Because of this, both servers will test collision against the client with the other clients held by the two servers. If the client crosses the border between the two regions, then the client will change

server, but if it passes over to the other server within a given time frame, then the server change will wait to check if it changes region again, else it will change the clients server, as illustrated in Figure 4.4 (c). The time delay for a new server change is set to 0.5 seconds.

**Out:** When a clients hull passes the *out* border, as illustrated in Figure 4.4 (d), then only the server within the client resides keeps updating the client and checks collision against the client. The other server stops updating, but keeps notifying the client about the clients that is on that particular server.

These three borders makes sure that the client does not change server to often, and reduces the number of messages sent between the client and servers. It also reduces the number of objects that a server has to detect collision between. This was because a server first checks collision for an object if it is within the region, that the server holds, or within the *in*-border.

## 4.4 Implementation

The application has been implemented in C++, utilizing a number of external open-source libraries. These include:

**Protobuf:** Serialisation of C++ objects to a byte sequence suitable for transmission[23].

**SDL:** GUI client for rendering of objects, user input, and so on[24].

**Boost::Asio:** Cross-platform networking code, sending and receiving packets[25].

The implementation consists of three separate programs:

**Server:** Responsible for disseminating information to clients.

**GUI Client:** Graphical interface for players to play the game.

**Test Client:** Developed purely for testing purposes. Acts like the GUI Client, but has no rendering of objects, and simulates player movement.

Collision detection is performed on servers. The current implementation uses *bounding sphere collision detection* but is implemented very naively. It is essentially a simple loop over all the objects in the scene, comparing the distance of each objects centre to every other objects centre. If the distance is less than some threshold there is a collision. Currently, the only collision which can occur are shot/ship collisions. Thus, the algorithm runs in $O(n \cdot m)$, where $n$ is the number of ships and $m$ is the number of shots.

Different types of messages are sent between clients and the server. Some messages are sent over UDP, while others are sent over TCP. The client programs and the server share a common packet reception module. The module helps abstract away the complexity of receiving and deserialising packets. Likewise, sending of packets is encapsulated in a module.

The module has a dispatch method for each message type. These methods are overwritten by the individual application and called with the message as an argument. Receptions

and deserialisation of messages happen on a separate thread - simultaneously with the game loop. When the game loop thread is ready to process the received messages, it will ask the reception module to run the dispatch methods on the main loop thread.

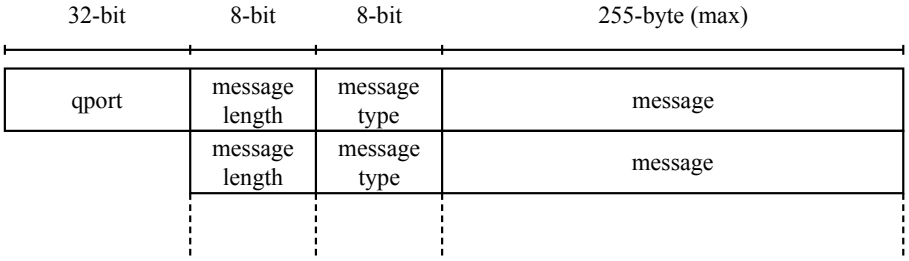



*Figure 4.5: A packet containing multiple messages.*

For serialisation of messages, an open source library called *Protobuf* is used. It allows for object oriented initialisation of messages and serialisation to a format suitable for transmission. Deserialisation is also handled by the Protobuf library. For UDP packets, the *qport* discussed in Section 4.2 is prepended to every packet. Several messages can be contained in the same packet. In order to properly deserialise at the receiving end, it is necessary to prepend each message with its length and its message type. As the messages are shorter than 256 bytes long, only a single byte is used for describing the message length. The message type is a single byte, which is mapped to a specific C++ class. This is similar to Type-length-value (TLV) encoding, however, the length and type are specified in the opposite order compared to TLV encoding. Figure 4.5 depicts a packet sent from the server containing multiple messages. Clients only send one message per packet as they only send single messages at given intervals.

## 4.5 Summary

This chapter described some of the design issues, when developing a single- and multi-server solution for a game. It described the basic topology that is used in both solutions. That is how the messages are received, parsed, used, and then sent again.

Then the single-server design was described. How it uses a standard client/server architecture, where all the game logic and physics are placed on the server, and how the clients sends input commands to the server to initialise a motion within the game logic and physics on the servers. Lastly the weaknesses of such an architecture was mentioned, which was a single point of failure and limited resources.

Next the multi-server design was described, which comprised of several servers in the server layer. This removed the barrier from the single-server solution, which was single point of failure and limited resources. It was also described how the partitioning of the game world was done, which was done by creating equalled sized regions, where a region would be given to a server. A server would know which client objects it held, and would only transfer the control of the client if it moved to another region, which was controlled by another server.

Lastly the protocol was discussed, which was both for the single- and multi-server solution. Which messages should be sent and how. It contained a short discussion of the usage of TCP and UDP for this task and why some packets needed a safety protocol, like TCP, which was because the packets had vital information for the client.

Chapter 5

# Test

This chapter covers the test environment in which the test will occur and which requirements all test cases will be evaluated against to give consistent overview of all the tests. Then the test cases will be described, detailing the scenarios which will be mimicked; both uniform distribution of players, but also crowding. Lastly the test results will be described, ending with a summary of the results.

## 5.1 Test Environment

This section covers the environment in which the different designs for a server architecture will be tested. The section describes both the hardware constraints in the test environment, but also the game world limitations put in place to make all the tests consistent.

### Hardware

To test the two solutions mentioned in Section 4.1 it has been chosen to test them on the same hardware, which is a cluster of computers at Aalborg University. The cluster consists of:

- Two types of machines:
  - 35x 2.8GHz Intel Pentium 4 (Northwood), 2GB memory, and 1Gb Ethernet.
  - 4x 2.4GHz Quad-Core Intel Xeon, 8GB memory, and 1Gb Ethernet.
- All of the computers are running Ubuntu 6.06LTS either 32-bit or 64-bit.
- The network switch is a HP ProCurve 2900 48-G with 1Gb Full Duplex Ethernet.

To keep the data collected as consistent as possible, the server application will only run on the 2.4GHz Quad-Core Intel Xeon machines. The reason for chosing a single type of machine is to keep the test results as consistent as possible. The reason for choosing the 2.4GHz Quad-Core Intel Xeon machines is that these machines have more processing power, than the 2.8GHz Intel Pentium 4 machines. The rest of the machines will be used to simulate the clients, which can be both the 2.4GHz Quad-Core Intel Xeon and the 2.8GHz Intel Pentium 4 machines, if they are available. The number of machines used will vary in each test setup.

### Game World

The game world is limited to be of a certain size. This is to both keep the bots within a certain area, but also to have a game world that is consistent in all tests. The size of the game world is found by looking at the test duration of each test. The duration is set to 15 minutes in all tests to get data that can be used to analyse for anomalies. The 15 minutes have been chosen, because of the magnitude of tests that needs to be done, however, should also be enough because of the size of the game world.

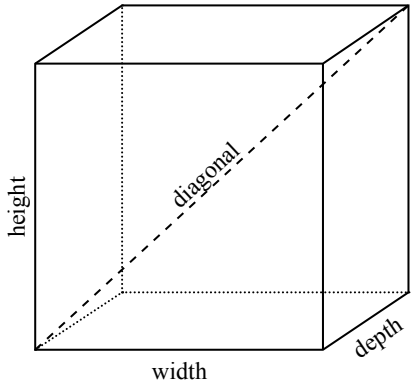



*Figure 5.1: The even sized cubic game world, where the diagonal, width, height, and depth is illustrated.*

The game world is restricted in size by using the maximum travel speed of ships and the time it takes to travel from one corner in the diagonal in an even sized cube to the other corner, as illustrated in Figure 5.1. The travel time is specified to be 30 seconds, so given a maximum speed of $50u/s$ ($u$ is a measurement of distance, units) the travel distance is: $50u/s \cdot 30s = 1.500u$. Since the game world is an even sized cube, then all three dimensions are of equal size. The size of the dimensions can be found by using simple geometry: $1.500u = \sqrt{w^2 + d^2 + h^2}$, where $w$ is width, $h$ is height, $d$ is depth, and $w = h = d$. This yields a size of each of the three dimensions: $866.03u$ or roughly $1000u$, which takes approximately 20 seconds to travel.

The size of the test game world is very small compared to that of EVE Online or World of Warcraft. The size has been chosen mainly because of the crowding aspect, but also because the small area yields more combat between the two teams, giving a more interesting test scenario because more combat should stress different areas of the solution more.

## 5.2 Test Metrics

To measure the difference in resource utilisation and how the single-server solution from Section 4.1 is performing according to the criteria mentioned in Section 1.2. Assuming a sufficiently consistent game world, the most basic criteria is latency. Latency can have numerous causes, but common to them is an exhaustion of resources. Thus, the metrics used in these tests are chosen such that they measure different kinds of resources which can cause latency if they are exhausted. The test metrics are consistent for all test cases. Each test case is going to be evaluated according these metrics:

**Bandwidth:** The bandwidth utilised by the servers. This is both inbound and outbound bandwidth.

- This is to measure the amount of traffic (kbyte/s) the servers are generating at any given point in time. Game creators can purchase additional bandwidth, however this can become unfeasibly expensive. Additionally, players may have limited bandwidth to the internet. This metric will help show if this can become a problem.

**Packets:** The total number of packets sent between the servers, but also between the server and the clients.

- This is to check the amount of packets that needs to be sent/received at any given point in time. Each packet traversing the network incurs individual processing overhead as each packet must be parsed and routed. Thus, transmitting many packages can cause latency by exhausting processing power at computers and routers. Sending many small packets can keep the bandwidth low while still incurring latency due to the sheer number of packets transmitted.

**CPU & Memory:** The resource utilisation of the servers in both CPU and memory utilisation.

- This is done to measure the local resource requirements at the server at a specific point in time. If processing power or memory becomes fully utilised, the server will be unable to perform its tasks clients will experience increased latency.

This is however not enough, since the data must be combined to determine what is affecting the bandwidth, CPU, memory utilisation, and so on. Therefore we are also going to measure:

**Players:** The total number of players currently connected to the game world, but also how many each server is handling.

**Servers:** The total number of servers currently used to manage the game world.

**Regions:** The total number of regions that the game world is partitioned into, but also how many players there are in each region.

Another test metric could be frame-rate, however, the server runs at a fixed frame-rate of 30 frames per second and if the server gets below the fixed frame-rate, the collision detection will begin to fail, thus frame-rate was omitted as a test metric. The only scenario in which frame-rate can go below this fixed frame-rate, is the scenario where CPU utilisation has reached $\approx 100\%$. Therefore it was deemed enough to only measure the CPU utilisation.

These test metrics are a requirement for each test case and are used throughout each test, such that they can be compared against each other to see which solution is best and which weaknesses and strengths each solution has.

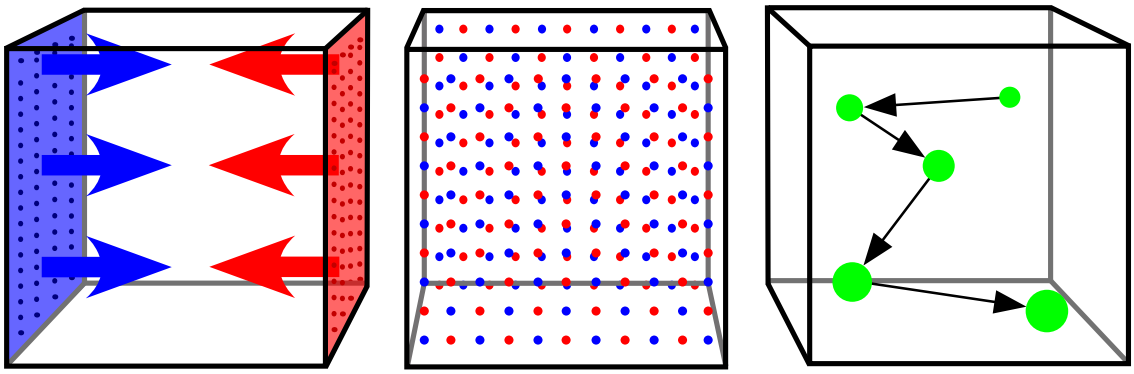



*Figure 5.2: Illustration of three test cases: (a) Red and blue teams are spawned at either end of the game world, (b) uniform distribution of players, and (c) moving combat point of the two teams.*

## 5.3 Test Cases

To test the single-server solution mentioned in Section 4.1, we have created three test cases. Each of the three test cases describes unique situations in Rock Pounder. The three cases are as illustrated in Figure 5.2 with their test case identifier:

**Head-on** ($a$)**:** The red and blue team will start at each side of the game world, where they will spawn at a random location in a spawning box that covers their end of the game world.

**Uniform** ($b$)**:** The red and blue team will spawn uniformly in the game universe with the same distance between each other, e.g. creating a three-dimensional grid, where the spawn areas are at the joints. The spawning on the grid will be blue, red, blue, and so on, such that the neighbours are enemies.

**Crowding** ($c$)**:** The red and blue team will move towards a point in space, called a hotspot or combat point. The hotspot will at given time intervals move to another random location in the game world. Thereby allowing the crowding of a region to move to another region. As long as players are in transit to a new point they will not fire at each other. Only when they reach within the vicinity of the point will they engage one another. The two teams will spawn as they did in Head-on ($a$).

Test case ($b$) is intended to be used, when testing the multi-server solution. This is because when using it with single-server all clients will still only be on one server, whereas in the multi-server the clients will be uniformly distributed amongst the servers participating in the test. Therefore test case ($b$) is not tested in the single-server solution.

## 5.4 Test Results

This section presents the test results from the single-server solution as was described in Section 4.1.1. The single-server solution has been tested with the two test cases: (*a*) and (*c*).

Because of the huge amount of data given at every test case, only a subset of the test data will be evaluated in the next sections. The results have been chosen because of their interesting results and the significance it has to the crowding and performance issues. All test results can be found on the CD in the Appendix folder.

### 5.4.1 Single-Server Test

The tests in the single-server solution mainly concerns the amount of traffic, that the different player limits yields and the difference in the traffic for the amount of players. The CPU usage is also of great concern, since they give an indication of the maximum number of players that a single-server solution can maintain and therefore the limitation of the solution. Therefore each of the test cases was executed with 40 to 260 players with intervals of 20 players for 15 minutes each.

#### Test Case (*a*): Head-On

In test case (*a*) all clients spawned at either end of the game world and would try to eliminate each other, as mentioned in Section 5.3. In this test case the players meet head on, which creates a large front where they engage each other in combat. This results in a large number of projectiles being shot, because a large number of players are engaged in dogfights against each other.

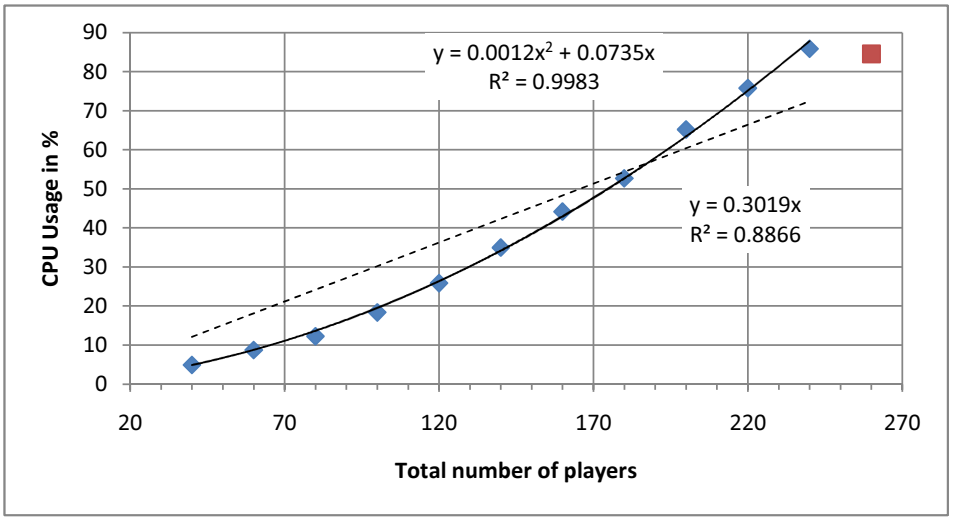



*Figure 5.3: The servers average CPU usage given a number of players from test case (a). The square measurement at* 260 *is an erroneous measurement, where the server cannot cope with the number of players. The dashed line is a linear trendline and the solid is a second degree polynomial trendline.*

The graph in Figure 5.3 illustrates the relation between the number of players connected to the server and CPU usage of the server. From the graph it can be seen that the second degree trendline has a $R^2$ value close to 1, which indicates that the trendlines residuals and the data points are very evenly matched. However the 260 player measurement is erroneous, since the server cannot handle the load as the CPU usage nears 100 percent. This results in the players input is not handled in time, and therefore slows their rate of fire, which results in a decrease of total number of live projectiles within the game world.

The main user of processing power is the collision detection, as described in Section 4.4, and the time complexity of the collision detection is $O(n \cdot m)$, which fits the overall CPU usage of a second degree curve. In this case each player shoots at other players, but often miss which yields a large number of projectiles in the game at any given time.

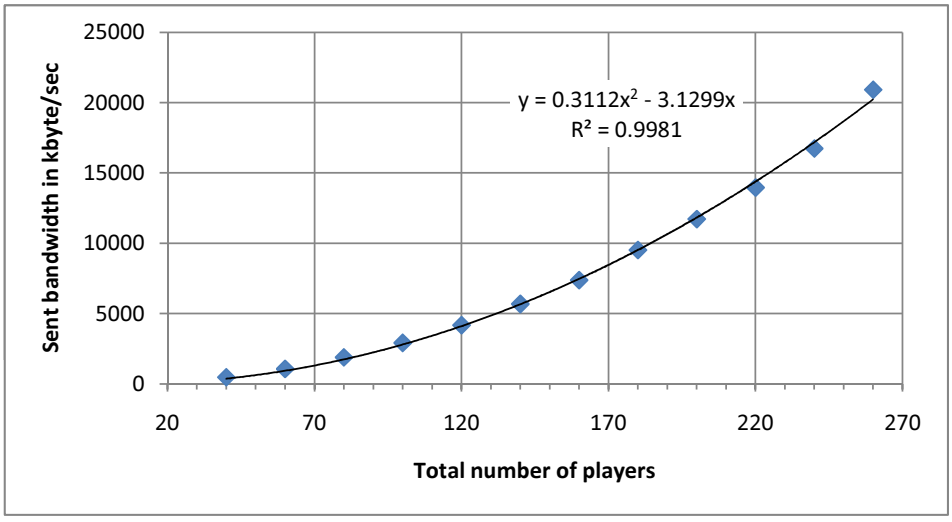



*Figure 5.4: The servers average sent bandwidth given a number of players from test case (a). The line is a second degree polynomial trendline.*

The graph in Figure 5.4 illustrates the relation of the sent bandwidth and the number of players from the server. The data point at 260 players gives a bandwidth usage of $\approx$ 21Mbyte/sec, which is because all players must receive information about all the other players. The total bandwidth from the server is second degree polynomial, as can be seen in the graph of Figure 5.4, which is given from the accumulated bandwidth sent to all the players.

The graph in Figure 5.5 illustrates one players received bandwidth, that has a linear trendline with the number of players for the clients. The trendline is linear, because the information sent per player is constant in relation to the total number of players. The projectiles are neglectable, because the information is only sent when they are created and the maximum number of live projectiles for each player is constant.

The graph in Figure 5.6 illustrates the relation between the received bandwidth and the total number of players for the server. The trendline is linear as the players sends a constant amount of data, which is because they have a fixed update rate. The traffic is however low as it is only around 0.5kbyte per player, so the bandwidth is neglectable compared to the incoming bandwidth, as illustrated by the graph in Figure 5.4. The player graph for sent

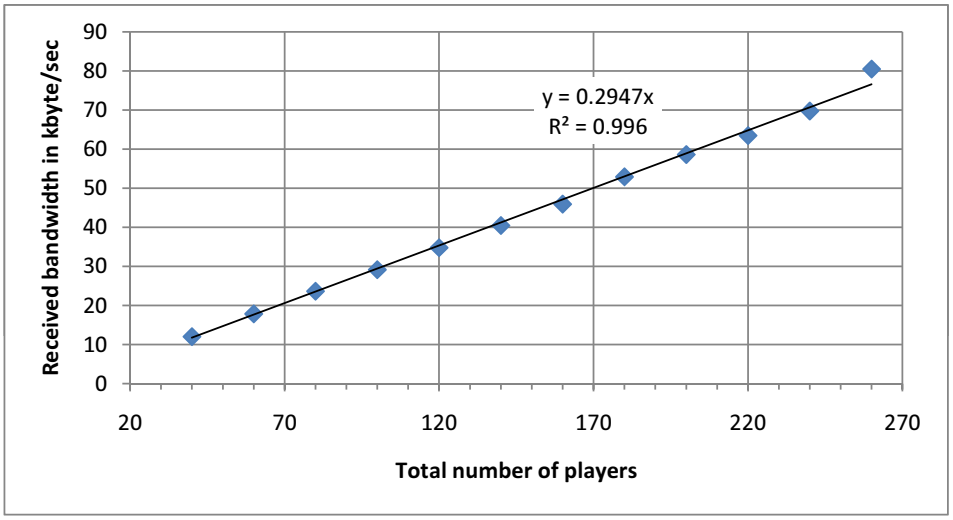



*Figure 5.5: One players average received bandwidth given a number of players from test case (a). The line is a linear trendline.*

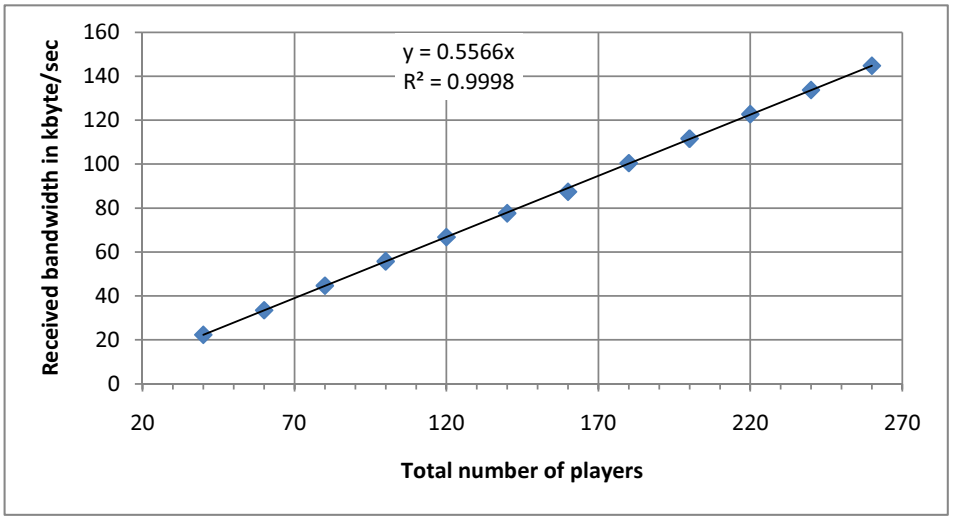



*Figure 5.6: The servers average received bandwidth given the total number of players from test case (a). The line is a linear trendline.*

bandwidth is constant as the player sends the same amount of information no matter the number of players connected.

### Test Case (c): Crowding

In test case (c), all players would move to a randomly chosen point in the game world and first begin to engage in combat, when they were within a given range of the point, as mentioned in Section 5.3, results in lesser projectiles being shot than in test case (a). This

is because there occurs periods of time, where the players needs to travel to a new point before they have permission to engage in combat again. In addition, all players are in close proximity and in a small area when engaging in combat. It is therefore less likely to miss a player when engaging in combat, than in test case $a$ and results in more kills. The random point is moved every 80 seconds.

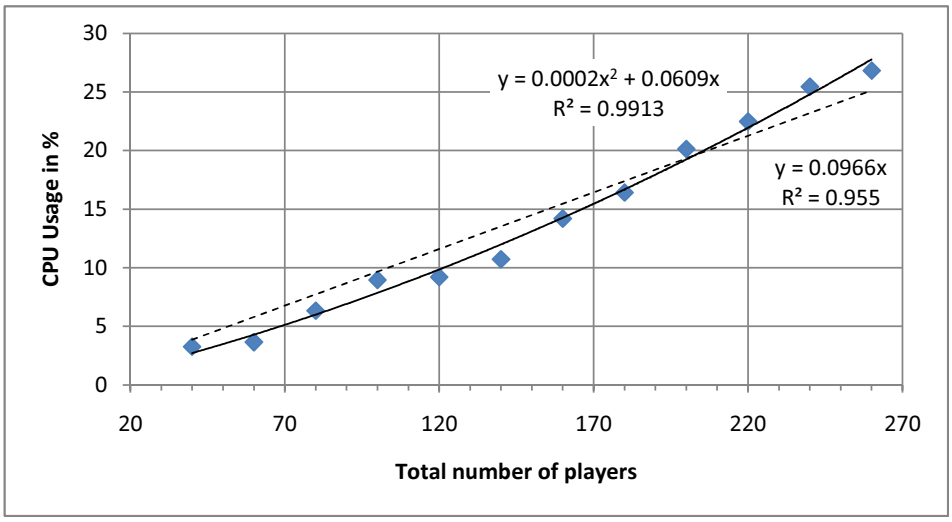



*Figure 5.7: The servers average CPU usage given a number of players from test case (c). The dashed line is a linear trendline and the line is a second degree polynomial trendline.*

The graph in Figure 5.7 illustrates the relation between the number of players connected to the server and CPU usage of the server. As in test case ($a$) the trendline is polynomial, but yields a much lower CPU usage. This is because there are fewer live projectiles in the game world, which is expected as players get eliminated much faster in this test case and therefore fewer projectiles are fired. The cluster that was used for the tests was however not big enough to reach the single-server solutions CPU limit, so the maximum load was approximated by using the existing trendline. To be able to approximate the maximum number of players the server can manage, there have to be determined a threshold for the CPU usage. From test case ($a$) in Figure 5.3 the test begins to fail, when the CPU usage is above 90%. With this assumption an approximation of a maximum number of clients can be made: $0.0002 * (550^2) + 0.0609 * 550 = 93.995\%$. Here 550 clients is approximated to be the maximum capacity of the server in this test case.

The graph in Figure 5.8 illustrates the relationship between the sent bandwidth and the total number of players for the server. The sent bandwidth is also less than in case ($a$), because of the players move more frequently in a straight line and therefore the prediction of the players are more accurate. Thereby the server has to send fewer updates, especially when the clients move to a new random point.

A graph of the sent bandwidth for 200 players can be seen in the graph of Figure 5.9. The graph has some drops every 80 seconds, which is because all players begins to move in a straight line to a new point. The trendline is also a polynomial as in case ($a$), as all players have to be informed of all other players.

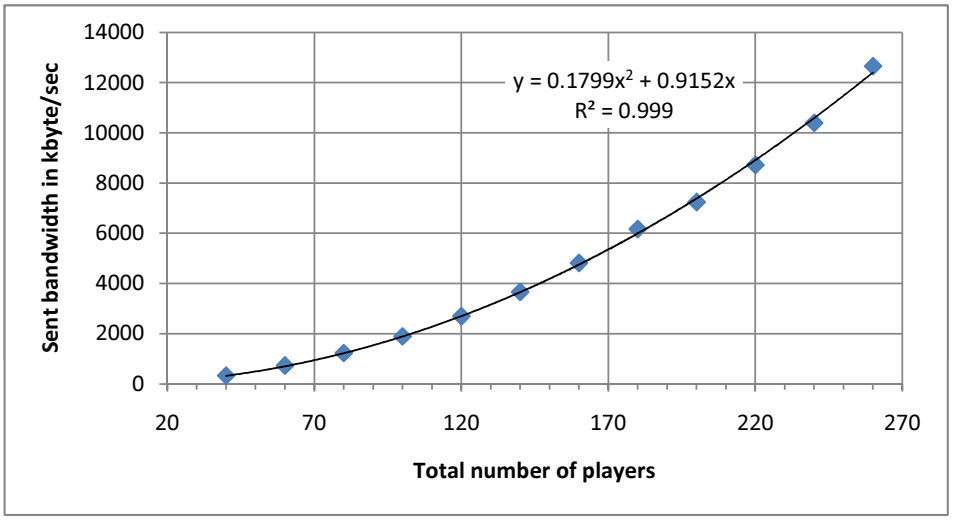



*Figure 5.8: The servers average sent bandwidth given a number of player from test case (c). The line is a second degree polynomial trendline.*

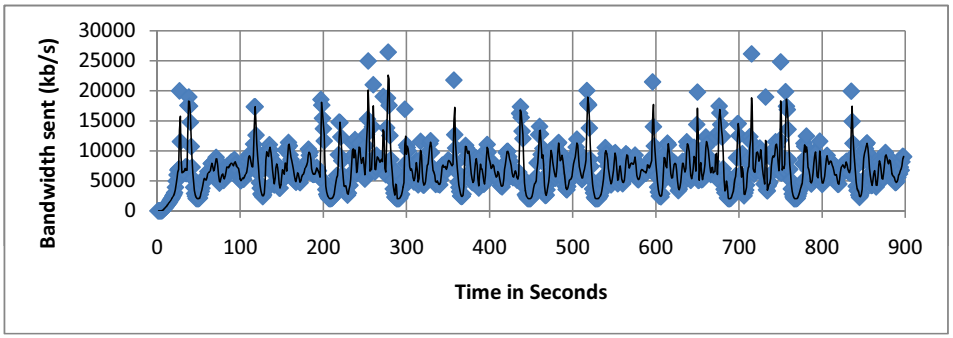



*Figure 5.9: The servers bandwidth over time for a test run with 200 players.*

The graph in Figure 5.10 illustrates the relationship between the received bandwidth and the number of players for the server. The received bandwidth is exactly the same as in case (*a*), because of the players fixed update rate.

## 5.5 Summary

A test environment for testing the solutions developed in this project was described, this included hardware environment as well as software environment. It also included a discussion on the chosen game world size, which was followed by a description of the criteria which were used to evaluate the test results along with a description of the types of data, which were necessary to collect during tests.

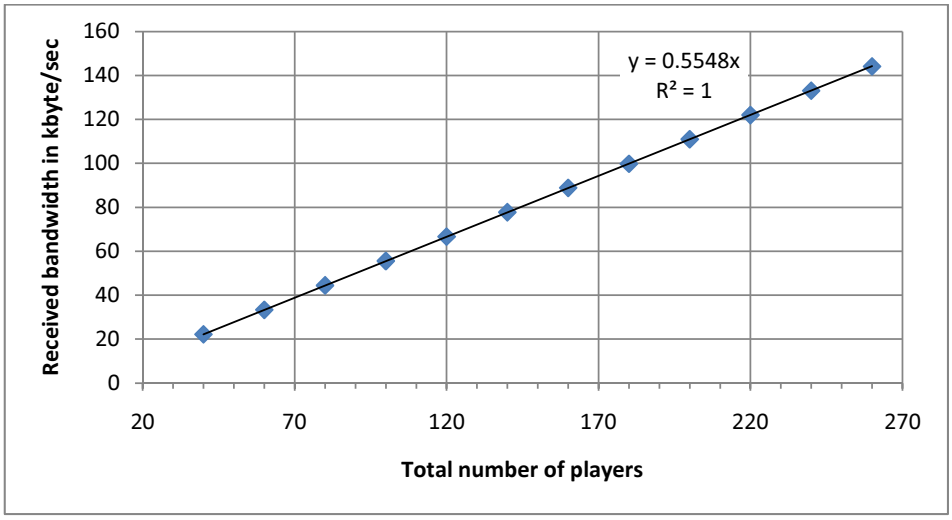



*Figure 5.10: The servers averaged received bandwidth given a number of player from test case (c). The line is a linear trendline.*

Subsequently, the different configurations of the test were outlined and discussed. The tests were performed in two realistic scenarios, which tested different performance characteristics of the solution.

Finally, the test results were presented and discussed. The tests showed that the architecture developed performed well and could support a relatively high number of concurrent players. The maximum was $\approx$ 240 players for test case $(a)$, where the server could not handle more players and still keep the appropriate update rate. This was due to the high number of projectiles live in the game world, which increased the number of objects that needed to be checked for collision. For test case $(c)$ the player limit can be up to $\approx$ 550, which because of the low number of projectiles due to the close proximity of fighting parties.

Chapter 6

# Conclusion

This project dealt with the analysis of networking in MMO games. As part of the analysis, a generalization of the architectures used in MMO games was presented and techniques like static and dynamic regionalisation to distribute the game world onto several machines, were covered.

Further, the report detailed the design and implementation of a client/server based FPS game. The initial implementation was a single-server solution. This implementation was refined to support multiple servers by using static region based interest management. However, due to time constraints only the single-server solution was tested in realistic scenarios played by computer controlled clients.

We conclude that the implementation performs in a sensible manner and providing an acceptable service level, even with many simultaneous players. The single-server test gave an insight in the maximum player limit of 240 for the head-on test case and $\approx$ 550 for the hot-spot test case, and of why these two test cases had such a big difference in the maximum player limit. The test results for the single-server implementation will serve as a good baseline to test future improvements against. Additionally, the testbed developed in this project will be applicable for testing future solutions.

As this report is the result of the first part of a two-part project, the second project will utilize the analysis, basic implementation, and testbed developed in this project to further investigate solutions to the scalability problem with focus on crowding.

## 6.1 Future Work

Future work should include proper testing and benchmarking of the multi-server solution developed during the course of this project.

There are several options to investigate with regard to improving scalability and handling crowding, specifically:

**Dynamic splitting:** It could be interesting to implement different dynamic splitting schemes in order to improve robustness against crowding. There are several schemes available, some of which are detailed in Section 2.6.2 on page 18. Section 3.2.1 on page 30 gave an insight into a novel splitting method (inspired by the Grid Division technique), which could be implemented in future work.

**Delta-encoding:** The current solution uses a mixture of TCP and UDP packets and could be improved by an optimized UDP-only solution. Quake III Arena uses delta-encoding in cooperation with acknowledgements and implementing a similar solution could have a positive impact on bandwidth usage.

**Proxies:** A layer of proxies could be inserted in front of the game servers facing the internet. This layer would aid game servers in transmitting identical data to clients, which could be accomplished as illustrated in Figure 6.1.

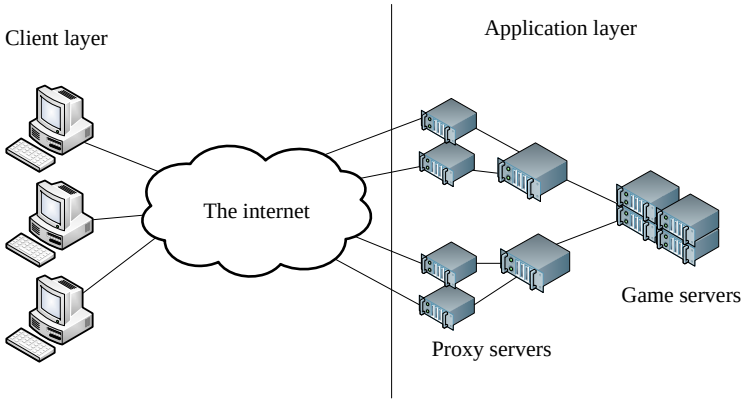



*Figure 6.1: Suggested design of proxy-assisted dissemination of game state*

**Persistence:** Investigating the persistence layer could also be interesting. This is an area we have not touched upon in the report, but the persistence layer serves a crucial role in the MMO architecture. It is quite possible that an "off-the-shelf" solution exists, e.g. an Oracle product or the open source Cassandra database[26].

**Improved collision detection:** An optimized collision detection algorithm used could improve upon the number of supported clients. There are many well documented optimizations available to implement, e.g. *spatial partitioning*[27].

# Bibliography


[1] Greg Costikyan. **I Have No Words & I Must Design**, 1994. http://www.costik.com/nowords.html.

[2] CCP Games. **EVE Online**. http://www.eveonline.com/.

[3] EVE Online Developers. **StacklessIO**, 2008. http://www.eveonline.com/devblog.asp?a=blog&bid=584.

[4] Blizzard Entertaintment. **Diablo I**. http://en.wikipedia.org/wiki/Diablo_%28video_game%29.

[5] Blizzard Entertaintment. **Diablo II**. http://us.blizzard.com/en-gb/games/d2/.

[6] Blizzard Entertaintment. **World of Warcraft**. http://www.worldofwarcraft.com/index.xml.

[7] Bruce Sterling Woodcock. **An Analysis of MMOG Subscription Growth**, 2008. http://www.mmogchart.com/analysis-and-conclusions/.

[8] Luke Hodorowicz. flipcode - **Elements Of A Game Engine**. http://www.flipcode.com/archives/Elements_Of_A_Game_Engine.shtml.

[9] Edward Angel. ***Interactive Computer Graphics***. Addison Wesley, fifth edition, 2008.

[10] Grenville Armitage, Dr. Mark Claypool, Philip Branch. ***Networking and Online Games***, chapter Playability versus Network Conditions and Cheats. John Wiley & Sons, Ltd, 2006.

[11] Lu, Fengyun and Parkin, Simon and Morgan, Graham. **Load balancing for massively multiplayer online games**. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, page 1, New York, NY, USA, 2006. ACM.

[12] **Unreal Networking Architecture**, 2009. http://unreal.epicgames.com/Network.htm.

[13] Ford, Bryan and Srisuresh, Pyda and Kegel, Dan. **Peer-to-peer communication across network address translators**. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 13–13, Berkeley, CA, USA, 2005. USENIX Association.

[14] Umar Farooq & John Glauert. **Managing Scalability and Load Distribution for Large Scale Virtual Worlds**. http://www.uea.ac.uk/polopoly_fs/1.133529!Farooq.pdf.

[15] Rajesh Krishna Balan, Maria Ebling, Paul Castro & Archan Misra. **Matrix: adaptive middleware for distributed multiplayer games**. 2005. http://delivery.acm.org.zorac.aub.aau.dk/10.1145/1520000/1515910/p390-krishna_balan.pdf?key1=1515910&key2=8633687521&coll=Portal&dl=GUIDE&CFID=60823480&CFTOKEN=34352052.

[16] Chris Greenhalgh. **Awareness-based Communication Management in the MASSIVE Systems**. 1998.

[17] Umar Farooq & John Glauert. **ARA: An Aggregate Region Assignment Algorithm for Resource Minimization and Load Distribution in Virtual Worlds**. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5272118&isnumber=5272056.

[18] Graham Morgan and Fengyun Lu. **Predictive Interest Management: An Approach to Managing Message Dissemination for Distributed Virtual Environments**. In *Proceedings of the First International Workshop on Interactive Rich Media Content Production:Architectures, Technologies, Applications, Tools (Richmedia2003)*, 2003.

[19] Andy Kuo. **A (very) brief history of cheating**. 2001. http://www.stanford.edu/group/htgg/cgi-bin/drupal/sites/default/files2/akuo_2001_2.pdf.

[20] **Got Frag**, 2009. http://www.gotfrag.com/.

[21] **PunkBuster - Online Countermeasures**, 2009. http://www.evenbalance.com/.

[22] Book of hook. **Quake 3 Networking**. http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/Quake3Networking.

[23] **Google Protobuf**. http://code.google.com/p/protobuf/.

[24] **Simple DirectMedia Layer**. http://www.libsdl.org/.

[25] **Boost::asio**. http://www.boost.org/doc/libs/release/libs/asio/.

[26] **Cassandra**. http://incubator.apache.org/cassandra/.

[27] **Wikipedia: Collision Detection**. http://en.wikipedia.org/wiki/Collision_detection.

Appendix A

# Terminology

Games have their own terminology, which is not well known in basic computer science. Therefore this section will provide a description of game terminology used in this report.

**Gameplay:** Gameplay refers to the gaming experience and game mechanics of a computer game, where the experience and how the player interacts with the game world is considered.

**Avatar:** An avatar is a player controlled character in a game world. An avatar can both be the main character in a story-driven game, but also a user-created character.

**Multiplayer:** A characteristic of games which support participation of more than one player, with each player utilizing one computer.

**MMO:** MMO describes a game with one or several persistent game worlds, where hundreds or even thousands of player avatars can interact with each other.

**Shard:** A shard is a persistent replica of the game world that has a number of players, also known as duplicate worlds.

**Game genres:** There exist different game genres in computer games, where some of them can be combined with the abbreviation MMO. Three types of game genres are: RPG, where a player takes the role of an avatar and tries to improve skills and abilities to survive in the game world. In FPS games a player takes a first-person perspective of the avatar. Last of the three is Real-Time Strategy (RTS), where a player has a godly role of controlling units and structures in a game world, which often take a third-person perspective of the game world. RPG, FPS, and RTS are just some of the game genres that exist.

**Shooter:** A shooter is a game type, where the player uses a ranged weapon to either score points or eliminate an opponent. A shooter can both take place in a first-person and third-person perspective, and usually requires a high update rate to update the position and orientation of the avatars in the game.

**Hack and slash:** Hack and slash refers to a game, where a players avatar has to slay a lot of opponents with melee weaponry.

**Score:** A score in a game gives a view of the performance of the player and are usually points or in violent video games, kills. The score in a game therefore can reflect the gameplay of the game. That is why not all games have a score system, where in other games a score system is such an intricate part of the gameplay, that it gives the player a goal of scoring better than previous game sessions.

**PvP:** PvP is a part of multiplayer games, where players compete against each other. PvP therefore is the interaction amongst players, and not players and environment, which usually is denoted PvE or Player vs Monster (PvM). PvP is therefore often seen in most multiplayer games, and usually plays an important role in FPS and RTS games.

**HUD:** A visual information aid to the players, that relays information like: Health, ammo, and map to the player. The HUD varies between games, where some has a large visual HUD to help the player or to give an visual experience, where other games do not have a HUD and has used other means to relay information like health to the player.

**Cooperative Gameplay:** Cooperative Gameplay (Co-op) is referred to multiplayer games, where players cooperate to reach a goal. This type of gameplay have been seen in many different game genres, e.g. in FPS games it could be to either eliminate the opposing team or to capture an area.

**Crowding:** A phenomenon in a game, where many players flock to one area of the game world.

Appendix **B**

# CD Content

The CD contains the following parts:

**Code:** The following code projects are included:

- **GUI Client:** The Rock Pounder GUI client written in C++, with both Visual C++ and XCode solutions.
- **Test Client:** The Rock Pounder test client written in C++, with both Visual C++, XCode, and autoreconf solutions.
- **Server:** The Rock Pounder server written in C++, with both Visual C++, XCode, and autoreconf solutions.

**Binaries:** The following binaries are included:

- **Windows:** The Rock Pounder GUI client, test client, and server.
- **OSX:** The Rock Pounder GUI client, test client, and server.
- **Ubuntu:** The Rock Pounder test client and server, but no GUI client has been created for Linux.

**Test results:** The following types of test results are included:

- **MS Excel:** Different test runs have been evaluated and plotted by using MS Excel.
- **Raw:** All test runs raw log files, both for the server and a single client.

**Report:** A PDF version of the report.

**Repository:** The SVN repository.