CHALMERS | UNIVERSITY OF GOTHENBURG

Optimizing an MMOG Network Layer

Reducing Bandwidth Usage for Player Position Updates in MilMo

Master of Science Thesis

JONAS ABRAHAMSSON
ANDERS MOBERG

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, January 2011

Optimizing an MMOG Network Layer
Reducing Bandwidth Usage for Player Position Updates in MilMo

Jonas Abrahamsson
Anders Moberg

Examiner: Staffan Björk

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden January 2011

Abstract

Massively Multiplayer Online Games give their players the opportunity to play together with thousands of other players logged into the same game server. When the number of players grows so does the network traffic at a quadratic rate. Care has to be taken to be able to support the amount of players wanted.

In this thesis, three different methods were explored and their ability to reduce the network traffic analyzed. The network traffic that was analyzed was generated by the use of a for the purpose developed test client that simulates player movement.

We show that to scale an online world to a large number of concurrent users, one needs to limit the amount of other users for which any one user needs to receive updates about. To achieve this, the game world needs to be partitioned.

Contents

1 Introduction

Online games of today allow for hundreds or even thousands of simultaneously connected players that interact with one another as well as with the game environment, which may have both static and dynamic parts. Supporting massive amounts of players in an online game comes with costs for bandwidth and maintaining servers, which makes the player-per-server count as well as the bandwidth-per-game-session requirements important business factors.

The network protocol of a game, which specifies when, how and in which format updates to the game state are communicated, is one part of a Massive Multiplayer Online Game (MMOG) that needs to be carefully planned and implemented to avoid unnecessary work for the server and client, as well as to provide a good player experience without lag and bugs.

Jonas has studied IT at Chalmers, with the addition and selection of several network oriented courses, after which he has studied the Game Development track of Interaction Design.

Anders has a background in mathematics, but later focusing on computer science and the Game Development track of Interaction Design.

The company Junebud AB (publ), is a newly started game development company. Junebud develops the game *MilMo*, a MMOG action game in which players explore a virtual archipelago defeating various creatures, collecting items, and solving quests. The *MilMo* client is built on the game engine Unity 3D and is played from inside any web browser, needing only an installation of the Unity plug-in software before the game loads.

1.1 Purpose

In *MilMo*, the game that Junebud is developing, the updating of player positions is the task that demands most of the server's computing time and bandwidth. This is mostly because it is the task that recurs most often, but also because it is a task that involves a large number of players, i.e., when one player sends a position update, there are a lot of other players that need to receive that update. In light of this, it was decided that this thesis would look only at optimizing the player position updates.

The purpose of this thesis is to find and compare ways to optimize the player position updates for MilMo in terms of the amount of data sent, without substantially increasing the server's load on cpu and memory.

In this thesis, different approaches to reduce the server-to-client traffic will be explored and their impact on server performance analysed. As a result of this thesis, several working prototypes will be implemented as proofs of concept that each solves or partly solves the problem with position updates of player characters. The prototypes will be tested and the reduction of network traffic will be compared to the original implementation.

1.2 Constraints

While many related papers have analysed the network traffic of already existing games, this thesis is executed during the creation process of the game, and thus comes with different opportunities and limitations for analysing the network traffic. With full insight into the network protocol, only the actual player position updates are considered for and not the network traffic in general.

The methods for optimization of the network traffic will not be tested on games in general, and instead only reasoned about. Furthermore, security issues such as validating the correctness of positions sent by the client and securing the server against denial of service attacks fall outside the scope of this thesis.

Different server architectures will not be analyzed and each game server will be running on exactly one physical machine, a constraint given by Junebud.

While the different optimizations implemented during this thesis will be tested and compared to the original implementation, they will not be proved to be optimal.

1.3 Target Audience

This thesis report is written with game developers in mind, which are not themselves game engine creators. The report emphasizes the performance gains which can be achieved from various network optimizations, and gives directions of how to implement these.

2 Background

The problems that come with growing network traffic when the number of players increases are not unique to *MilMo* and has been handled in different ways in many games before. In this section, existing games are presented along with *MilMo* and Darkstar, the server API used for the *MilMo* game server.

2.1 Existing Network Games

The requirements on a game network layer are heavily dependent on the core gameplay of the game. Existing network layers in a number of genres were studied and the findings are presented here.

2.1.1 FPS

The First Person Shooter (FPS) genre revolves around a fast-paced, usually violent, gameplay which for network play has high demands for low latency. Most of the early networked FPS games, like Maze War [28] and Doom [20], had support for only a few concurrent players. There were exceptions though, like MIDI Maze, which supported up to 16 players. As multiplayer FPS games became popular the demand to be able to play them from home, on a dial-up connection, was raised.

To be able to provide the low latency audio and visual feedback over high latency connections, the idea of predicting movement on the client was born. Client side prediction is the simple idea of letting each client calculate the future game state e.g. where the players are moving. How this prediction works is explained further in the theory chapter.

Quake3
Quake3 [21] is a popular FPS from 1999, much praised for its multiplayer mode. Part of its success can surely be ascribed to its new, quite different, network approach.

The network protocol in Quake 3 differs from many network protocols for games in that it doesn't mediate events, such as player movements, but rather a complete game state. This is advantageous for Quake 3's fast-paced FPS gameplay because it provides low-latency replication of the game state in such a way that differences between clients are kept at a minimum and errors are scarce. To lower the required bandwidth the game state packets are delta compressed using the last known shared game state as a base. This means that the server has to keep one entire game state in memory for each connected player, which is one of the reasons that this solution does not scale well as player numbers reach hundreds.

Unreal Tournament
Unreal Tournament [15], developed by Epic Games and Digital Extremes, was released almost at the same time as Quake 3.

An approach used in Unreal Tournament to reduce bandwidth requirements was to keep a set of relevant actors for each player, that is the set of players and game objects that the player can see, hear or otherwise affect the player. Furthermore, all actors are prioritized and given an amount of bandwidth according to the ratio between the different priorities. When the game state has been updated, only those variables in the actors that changed are sent to the players, and only for actors that are in the relevant set. Furthermore, the data passed between the server and clients can be set to either reliable or unreliable depending if they should be guaranteed to arrive at the receiver. In addition to mentioned approaches to reduce network traffic, to compensate for latency, Unreal Tournament uses a client prediction scheme to smoothen the actors movement [31].

2.1.2 RTS

Real Time Strategy (RTS) games are war games that require the player to make tactical choices in real time, as opposed to turn based war games where the player often has unlimited amounts of time to think for each move. The big challenge in networking an RTS game is how to enable massive amounts of units while still being absolutely correct, i.e. making sure all game events happen in the same way for all players. Real Time Strategy games are not as time-sensitive as FPS games.

Age of Empires
Age of Empires [25] is an RTS game with a historic theme. It was developed by Ensemble Studios and released by Microsoft Game Studios in 1997. Its network architecture was designed with the somewhat ambitious goal of supporting 8 players in multiplayer over 28.8 kb/s modem connections running a smooth simulation even on the minimum machine configuration. To meet that goal, several solutions and optimizations were made [6]. The base of the architecture is that all computers involved run the same simulation, with identical input. For this to work, the clients send their input ahead of time, i.e. in every turn the computer sends the input that is to be executed two turns later. In Age of Empires a turn is typically a fifth of a second long.

Additionally, a speed control system was used to ensure a smooth user experience. The reason for this is that in order to have all computers run the same simulation, the simulation can only run as fast as the slowest computer can run it. To beat the lagging experience that users with faster computers would have while waiting for enough data to end a turn, each client communicates an average frame rate and a worst ping time at the end of each round. The average frame rate sent is an average over the last couple of frames and the worst ping time is the longest average ping time to any of the other clients. Each turn, the host, which is one of the clients chosen at the start of the game to be host, calculates a target frame rate and turn length that is suited for the slowest client and the current network conditions. The host then sends this frame rate and turn length to the other clients.

StarCraft
StarCraft [7] is an RTS game with a science fiction setting and was developed by Blizzard Entertainment. StarCraft, like Age of Empires, uses a peer-to-peer network model for communicating actions [13]. As opposite to the client/server model, while the network traffic still grows quadratically with the number of players, each player in a peer-to-peer network only experiences a linear growth of data sent and received.

2.1.3 MMOG

Massively Multplayer Online Games differs quite a lot in their requirements compared to games in the FPS genre and other network games. The most obvious difference is the number of players playing in the same instance of the game. Where FPS games handle tens of players, MMOGs have support for hundreds or thousands of players.

There are several sub genres of MMOGs such as MMORPG, MMOFPS, MMORTS and virtual worlds with little or no gameplay in them. MMOFPS and MMORTS borrow the gameplay from the FPS and RTS genre respectively, but allow for many more players than what is normal for their original genres. MMORPG or Massively Multiplayer Online Role Playing Game, in turn, borrows its concept from role playing games. Social MMOGs focuses more on player interaction and has less actual gameplay.

World of Warcraft
World of Warcraft [8] is an MMORPG with a fantasy theme, and in which the players combat NPCs (non-player characters) in what is referred to as PvE, or player versus environment, or combat each other in PvP, or player versus player. The combat model differs from that commonly

used in FPSs in that, that the player initiates combat and then the combat actions are based on a timer. Special abilities, or spells, can be cast in addition to the timer based combat behaviour and whether the spells hit is not based on the aiming of the player but rather randomized and depending on the avatar's gear and level. The positioning of the avatar and the distance to the target also affect the spells but in terms of being able to use the spell or to reach the target. As the combat model used in World of Warcraft is less dependent on the exact aiming to hit the target, it is also less dependent on always having the exact position of the target than in an FPS.

The world of World of Warcraft is quite extensive, which helps in spreading the players. This is positive for server performance as there will be less data sent through it caused by player interaction and movement that needs to be propagated to surrounding players. A smaller world would have increased the player density and thus increasing the amount of data sent.

Another concept used in World of Warcraft is instances. An instance is a copy of an area in the game in which a limited amount of players has access to at a time. Multiple copies of the same area can, however, be active simultaneously to allow for several players playing in the same place without interacting with each other. Much of the end game content use instances such as dungeons, battlegrounds and arenas.

Measurements made in Svoboda et al. [30] states that the median bandwidth outgoing from the server is 6.9kbit/s per player, which can be compared to the estimated 53.6kbit/s – 154.4kbit/s from only position updates in the existing MilMo implementation for the wanted number of players per island.

EVE Online

EVE Online [10] is a science fiction MMOG set in space. The players can gather resources, manufacture items, explore the galaxy and combat each other in large scale battles. Eve Online differs from many other MMORPGs in that, that there is only a single instance of the game, which all players inhabit [14]. EVE Online has on occasions had more than 40000 players logged into the same instance.

The world of EVE Online consists of thousands of solar systems, which each is run on one of many servers in the server cluster. Several low populated solar systems can run on the same server while highly populated solar systems each needs their own server. Despite having the most powerful server cluster for a single game instance in the gaming industry, as players are free to go wherever they want, popular solar systems can experience severe performance issues during peak hours [14].

Combat in EVE is in some ways fairly similar to that of World of Warcraft. The player locks on to an enemy spacecraft and sets the weapons to fire and the ship will fire automatically with the given weapons. Other gadgets can also be used to cripple the enemy ships and in such works a bit like spells in World of Warcraft.

2.2 MilMo

MilMo, the game that this thesis revolves around, is a social Massively Multiplayer Online Game (MMOG) with an action adventure setting where the players can socialize with each other, solve quests, fight creatures and explore a virtual archipelago. *MilMo* is based on the Korean business model free to play (F2P). The players can access all the content in the game for free and may, if they wish, pay extra for character customization such as new clothes and hairstyles.

The game consists of multiple islands which the players can travel between. It is the wish of Junebud that a single server should allow for 100 players on each island at the same time and that each server should hold ten of these islands. To hold this amount of players, in the existing implementation a bandwidth usage of 18.4 Tb – 46.7 Tb (depending on packet congestion) per month and server is estimated solely from position updates, which is also deemed the largest individual portion of all network traffic. The servers, however, are limited to 15 Tb monthly data sent per server and with more than just position updates to send, the traffic needs to be reduced with about a factor of three merely to handle the worst case scenario for position updates.

2.3 Project Darkstar

The MilMo game server is built upon the Project Darkstar API. Darkstar has several features to help the game developer, most important are the parallelization features.

In parallel computing the biggest challenge is to have parallel processes share memory, in other words, to communicate. The operating system hides most of the problems regarding sharing of resources, network adapters and such, to the programmer by limiting the hardware access and exposing an API. Synchronizing memory sharing and creating multiple processes, or rather threads, is, however, up to the application programmer, but there are libraries available that can make the task easier.

When developing an application to run in parallel threads, there are two models to use, data parallelism and task parallelism. The different models are applicable to different problems but for game development data parallelism, where the same instructions are carried out to different units of data, has little application outside of graphics where it is used on the GPU. In task parallelism different instructions are carried out on either the same or different data, making it more prone to suffer from memory sharing and synchronization issues.

Project Darktar solves the memory sharing issues by providing a data storage from where threads, in Darkstar called tasks, can access shared data either for reading or for both reading and writing. If a task request permission on data that another task has accessed for writing to, both tasks are aborted, their changes to the data storage rolled back, and they are rescheduled to be run at a later time. This behaviour allows the developer to spend less time managing memory sharing and communication between processes and instead focusing on writing the task in a manner similar to when developing a serial, single threaded program. At the same time, the developer needs to be alert of the possible contention that can occur between tasks that share data, when at least one task tries to modify the shared data.

The data in the storage is automatically persisted using the Berkley DB, and for that reason all data submitted to the data storage has to be serializable. Even the tasks are submitted to the data storage so that if the game server crashes, the next time it starts it will be in the same state as before the crash. The operating system will have closed its network connections though, so clients need to reconnect.

More than just hiding threading and task scheduling issues, Project Darkstar also includes features for load balancing over multiple nodes in a server cluster. Since the tasks and all data they need are serializable, they can be computed at any node in the cluster irrespective of which node it was scheduled at. The load balancing implementation used in Darkstar has the advantage over zone based approaches in that, that the servers handling sparsely populated zones could be running at less than their capacity while at the same time servers running densly populated zones could have a hard time coping with the amount of clients connected to them. Instead Darkstar spreads the tasks over multiple nodes evening out the workload of the nodes in the server cluster. At the time of this writing, the system is, however, not fully implemented.

Darkstar also has an interface to handle the network connection and communication with the game clients. There are two different ways to communicate in Darkstar: either the communication is done over a session to a single client, or through a channel. A channel groups together multiple client and let the server send to all clients that are members of that channel.

As of 2010-02-02 Darkstar is no longer being developed by Sun Labs, there is a community fork with the name RedDwarf where the development continues.

3 Theory

Several papers written in the area of network traffic in games focuses on analysing existing games and their respective network architectures' and protocols' impact on game performance [11, 17, 13]. Some focus on the server performance as a function of the players' behaviour[12, 23, 32], while other tries to make a simulation of the players and test concepts for reducing the server load [27].

In Steed and Abou-Haidar [27] pedestrians in the inner parts of London are simulated and the world partitioned according to different partitioning schemes. How player behaviour and environment, in terms of moving or standing still in densely and sparsely populated areas, relates to the network traffic sent was studied in Chen and Lei [12], Kinicki and Claypool [23] and in Szabó et al. [32].

3.1 Network Protocols

Different games and game types use different ways for communicating the server's game state to the clients: passing events or passing the entire game state or change thereof. When passing events it is important that all events are received by the client so that the game states of the client and server do not diverge. It is also important for the client to receive the events in the same order as they occurred or the game states might diverge even though all events were received.

When sending the game state, this is often done so at a high frequency. If a network packet is lost, the next received packet will be used instead and if two packets arrive in the wrong order, only the most recent packet is used. This removes the need of the delivery guarantee and the order guarantee that the sending of events requires.

The two ways of communicating the server's state to the client maps well to the two network protocols used on the internet, TCP and UDP [9]. Sending events is done using TCP and sending states is done using UDP.

3.1.1 UDP

When sending network packets with the User Datagram Protocol no session is established between the sender and receiver and on a protocol level, there is no way for either part of knowing if all packets are received, and if they are received in the correct order. If delivery and order guarantees are wanted it can be achieved by implementing them in the application layer protocol [9].

UDP is the favoured protocol for several games in the FPS genre, such as Quake 3, Call of Duty, Battlefield 1942 and Half Life [1, 2]. If a protocol with a delivery guarantee had been used, the game would have stalled while waiting for retransmission of lost packets while the server at the same time would have more recent states to transmit to the client. Instead, if a packet is lost, the game state is estimated with client side prediction techniques and the next received packet is used.

3.1.2 TCP

TCP, as opposed to UDP, is built for correctness, and all TCP packets are acknowledged by the receiver sending an ACK packet, to guarantee the sender that the packet has been received. If no ACK is sent from the receiver within a given time, the packet is retransmitted by the sender. As TCP also guarantees that the packets are received in order, a packet that is not received will block later packets until the lost packet has been retransmitted successfully.

Nagle's algorithm [26] was created to enhance the TCP/IP protocol when working with several small packets, where the TCP header would make up a considerable part of the total data sent. Packets are coalesced by waiting for more data to be sent as long as there are packets sent, which have not yet been ACKed or that the send buffer has reached a certain threshold.

Delayed ACKs [9] is another optimization to increase performance while using TCP. The assumption is that when the application receives a packet, it will soon send a response. By waiting for 100-200ms, if the application sends a packet in that time frame, the ACK can be sent together with that packet instead of sending two packets which of one will have no payload. Also, every second packet will be ACKed.

Using both of these techniques together can, however, lead to extensive delays [29]. If only a small packet is sent, Nagle's Algorithm will cause a delay of 200ms before the actual transmission is done. Before an ACK is received, no more data will be sent, but at the same time the receiver will not send an ACK until it has either another packet to ACK, a packet of its own to send or has waited for 100–200ms.

What has been observed in other studies is that the packet streams have been very thin with only three to five packets per second [11, 17, 30], which means that an ACK will often be sent due to a timeout. Griwodz and Halvorsen [17] analyses how various methods to ACK affect retransmission times, and concludes that changing ACK-policy can lead to performance gains. Furthermore, it was found in Svoboda et al. [30] that in World of Warcraft most packets, had the PSH flag set, which will override Nagle's algorithm.

When using TCP for MMORPGs, a large portion of all packets sent are acknowledgements [11][30]. Furthermore, Chen et al. [11] states that almost three fourths (73%) of all transmitted data sent from the client to the server is due to TCP/IP headers and that 30% is from ACK.

3.2 Area of Interest

A concept described in Griwodz and Halvorsen [17] as well as in Steed and Abou-Haidar [27] is Area of Interest (AOI). The AOI is the area in which a player is interested in knowing the actions of other players. The same idea is mentioned by Huang et al. [18], which also states that most of the time, there are few other players in a player's view scope. Different parameters can affect the AOI such as distance to the other players and whether or not those players are visible to the observing player.

The problem the AOI tries to handle is the quadratic behavior in server and client load that comes from every client communicating and interacting with every other client through the server. To manage the AOI on a client-to-client basis will however have the same quadratic behaviour as the original problem.

A way to handle the AOI is to create a spatial partitioning of the game world. There are several ways of how a partitioning could be created. The partitioning can, for instance, be made as a regular grid or have tree-like structure, the structure can be static, calculated from data of the average player distribution in the world, or be updated dynamically as the players move around in the world. By handling all the players inside a partition in the same manner, the cost of the AOI management is reduced at the expense of not being optimal, that is, clients will receive more information than they need to know about.

In a server cluster, each partition, or a group of partitions, can be assigned to a specific server, reducing both the network traffic and the server load. A single server environment can still benefit from implementing AOI but as the network load is decreased, the load on the server itself can become a problem when the number of players increases.

3.3 Client Side Prediction

A method for getting a smooth avatar movement in network games is to use client side prediction. This means that the client is estimating where the other avatars will be given their previously known positions, velocities and input. Client side prediction can be done by letting each client progress the game state with the same procedure as the server, i.e. reading input, simulating physics, etc. It can also be done by simply extrapolating ahead of time from previously known positions, so called Dead Reckoning [22]. Given a correct initial state, dead reckoning can be used to reduce the network traffic by only sending updates when the input changes and let the clients simulate the movement of the other avatars. Not to deviate too much from the server's position, the simulated position is adjusted over time towards the correct position given with the updates.

An alternative to client side prediction for smoothing avatar movement is using interpolation between known positions. This introduces extra latency but gives a more correct result and is easier to implement.

3.4 Huffman Coding

Huffman Coding [19] is a general compression technique that is used in existing games [3], and it is based on the frequency of the symbols in the data to be compressed. Normally, in computers each symbol, or byte, is represented with eight bits, but when applying Huffman Coding to the data, the most frequent symbols are represented with fewer bits while the less frequent are represented with more. Given the frequency of the symbols a so called Huffman tree can be computed and used to compress the data. If the frequency, or an approximate thereof, is known beforehand it can be used to precompute the Huffman tree, otherwise the tree has to be computed on the fly and passed along with the data.

3.5 Player Behaviour

How players behave has a large impact on how much network traffic that is generated. In Chen and Lei [12], the player behaviour is studied in terms of player distribution over the world and how the players interacts with each other. It was found that in ShenZhou Online the 30% of the players are located in 1% of the world. Furthermore, 40% of all players had been in the vicinity of at least four other players during their game session and 20% had been in the vicinity of more than 100 other players. More than 10% of the players spent their entire session in the most popular place in the game world.

The effects on the network traffic from moving and standing in differently populated areas was analysed in Kinicki and Claypool [23] as well as in Szabó et al. [32]. In Second Life [24], the largest amount of traffic is generated by moving in densely populated area, followed by standing in densely populated areas, moving in sparsely populated areas and standing in sparsely populated areas. It was also found that the number of object in the areas had a large impact on how much traffic that was generated.

The ratio of how much players tend to be standing and moving in densely and sparsely populated areas was presented in Szabó et al. [32]. In World of Warcraft players tend to be standing still in cities (or any other densely populated area) for an average of 42% of the time for sessions longer than one hour. For sessions shorter than one hour the ratio was found to be 35%. In other MMORGPs the same values were 29% and 35% respectively. Together with standing still outside of cities, players were standing still for roughly half the time logged in.

4 Methodology and Planning

As the thesis work took place in the development phase of the game *MilMo*, there was a great likelihood that issues not related to the thesis could arise, but that still would affect its work flow. Such unpredictable interruptions lead to the use of ideas from agile development [5] with short iterations and frequent re-planning to early address problematic areas. Only a rough plan was created initially and just the upcoming week would be planned in more detail.

To gain better knowledge of what needed to be done, two different tasks were found: researching existing literature on the subject and to familiarize with the existing code base. The need of a tool to analyse the network traffic was evident, and as at the time several of the network messages split up over multiple files, it was decided that one of the initial tasks would be a protocol generator, as it would help with the understanding of the network model in *MilMo*. It was also important that the protocol generator was done first in the project as the network optimizations would be using the generated protocol. The analysis tool would be written as a plug-in for Wireshark [33] and would be based on the generated protocols as well.

As the exact impact of the optimizations would be hard to predict, the implementation process was split up in three phases, with each phase being three weeks. One week was assigned to research and design, one week of the actual implementation and one week of testing and evaluation. The process would then repeat taking the knowledge gained in the previous phase as input to the next. Depending on the state of the implementation, the coming phase could either be used to iterate on the existing optimization or to start with a new one. Although borrowing ideas from, it is not a strictly iterative process as possibly three different optimizations could be implemented with no connection to each other.

Table 1: Initial plan

Task	Week
Protocol generator / further research and inventory	32
Protocol generator / further research and inventory	33
Protocol generator / further research and inventory	34
Wireshark Plugin	35
Integration of the new protocols into the existing code	36
Research and design (Phase 1)	37
Implementation (Phase 1)	38
Evaluation (Phase 1)	39
Research and design (Phase 2)	40
Implementation (Phase 2)	41
Evaluation (Phase 2)	42
Research och design (Phase 3)	43
Implementation (Phase 3)	44
Evaluation (Phase 3)	45
Report	47
Report	48
Report	49
Report	50
Report	51

5 Execution

The planning described more in detail in the previous chapter led to an initial priority order of all the subtasks that were needed for this thesis. The priority was based on which tasks that had dependencies on other tasks and which tasks that could be executed in parallel by two persons. Halfway through the execution of the thesis, both authors started working part time for Junebud, which drastically changed the time frame of the thesis.

5.1 Literature Review

Literature was searched from the ACM Digital Library, CiteSeer^X, Gamasutra and Springer among others. When a few papers related to the subject had been found, papers referred to or papers referring to the ones first found were then followed up.

A few distinguishable categories of papers were found: [11, 17, 30] analyse the characteristics of MMORPG network traffic, [16, 27] discuss and implement ways of partitioning virtual environments and [12, 23, 32] look at what impact the players' behaviour has on the network traffic.

Before the literature review started an idea about a spatial partitioning of the world already existed, and the literature found solidified that idea. Ideas were also borrowed from the field of computer graphics to have a different level of details on the position updates that the clients send when the players move around in the world depending on the distance between the players.

An assumption made about the players' behaviour is that they will be standing still for a considerable amount of time both while playing and when leaving their characters online while being away from the keyboard. According to the study performed in [11], in ShenZhou Online, most players had been idle for periods of time, and some players were idle most of the time online. The authors of [32] finds that in MMORPGs players are standing still for about 50% of their time logged in and the majority of this time the players are in crowded areas.

The papers on the characteristics of the network traffic emphasized the existing optimizations and their impacts on the games studied. One of the reasons for reducing the amount of data that was sent in each individual network packet, even though the packet header often was much larger that the payload, was that Nagle's Algorithm would coalesce many of these packets together making the reduction have a larger impact. This was also one of the reasons that Darkstar's channels were not used initially, as they induced additional overhead.

5.2 Protocol Generator

At the time the protocol used in the game was not gathered in a single place but rather spread over multiple files and the messages sent over the network was read not from a single place in the code but rather passed around and read partially in different places. Additions to the protocol were also done on a regular basis, so to be able to write a tool to analyse the network traffic from the game, the need of a generated protocol was identified.

The requirements on the protocol generator was found to be:

- The protocol should be described in a single place and should be easy to read.
- The messages should have a clean interface and not be passed around and read partially.
- The game logic, in response to received messages, should be executed in message handlers, in which the type of message would be given.

A few major subsystem were identified:

- Parsing the protocol description.
- Generating an abstract syntax tree (AST) from the parsed protocol.

- Generate the code from the tree.

XML being fairly descriptive, easily readable to humans and as java is shipped with an easy-to-use tool for creating XML parsers, it was decided that the generator should be written in java, parsing XML files describing the protocol. As the game server was written in java and the game client written in C#, which are fairly similar languages, the same AST could be used for both the server and the client. The AST and the code generators was created using the visitor pattern, letting the two visitors, one for each language, accumulate the output code while traversing the trees. This design also made it possible to work with the two visitors in parallel.

When the process of creating the generator started, the idea was to send the data structures already existing within the game, letting the user fill in how the data structures would be parsed. This idea got as far as to when the generated code was going to be integrated into the existing server code. There were, however, many flaws in the system which of the biggest was that not all data structures easily could be made into messages sent over the network. This led to an almost complete overhaul of several subsystems, keeping merely the visitors and the structures of the abstract syntax tree.

Instead of the previous approach, both data structures and messages were automatically generated using basically the same system, where the messages are special cases of the data structures. The only thing that a user of the system would have to implement is the actual handling of received messages.

As the rewrite of the generator dragged on for two weeks longer than the initial planning, the Wireshark plug-in was postponed as it was not going to be used until the analysis and as the partitioning solution was deemed more important.

5.3 Dead Reckoning

It was suggested from our mentor at Junebud that we first look into Dead Reckoning as a method of client side prediction. Client side prediction is traditionally used to conceal latency. If implemented, this would allow for a lower message rate, thus reducing bandwidth usage, while still maintaining a good end user experience.

However, early tests showed significant problems with synchronization i.e. merging the result of the prediction with the later arriving actual state from the server. Because of these problems, Dead Reckoning was considered unable to allow enough lowering of the message rate to significantly lower bandwidth usage and was thus dropped in favour of other methods.

5.4 Partitioning Phase One

To get an idea of how much the network traffic will grow with the number of players, consider the following in figure 1: players A, B and C are connected to the server and player D connects. For every update of player D's position that the other players need to know about, the server will have to send D's position to A, B and C and vice versa. In general, let n_i denote the number of unordered pairs of unique players among i players. Adding one more player would add i more pairs as the new player can be paired with any other player, thus the number of pairs for $n_{i+1} = n_i + i$. With no pairs existing for only one player and expanding this formula recursively, this can be expressed as

$$n_{i+1} = \sum_{i=0}^{n} i = \frac{\sum_{i=0}^{n} i + \sum_{i=0}^{n} (n-i)}{2} = \frac{\sum_{i=0}^{n} i + (n-i)}{2} = \frac{\sum_{i=0}^{n} n}{2} = \frac{(n+1) \times n}{2}$$

The network traffic is said to have a quadratic behaviour in regard to the number of connected players.

In order to escape this quadratic behaviour a spatial partitioning is applied, so that a player's position is only sent to those players that are in the vicinity of it and thus need to know about it.

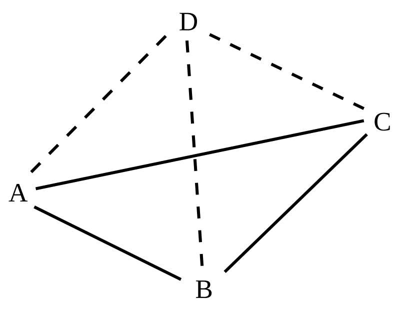


Figure 1: To add the fourth player, D, the position of D needs to be broadcast to all the other players, A, B and C.

Although this does not remove the quadratic behaviour with regard to the number of connected players, it should reduce it.

As spatial partitioning was the initial idea of network traffic reduction, it was the first optimization method explored when dead reckoning had been dismissed. A player is located inside one partition, and listens only to messages from partitions nearby. Furthermore, in addition to distance, messages could also be refrained from being sent if the recipient is inside a partition occluded by geometry.

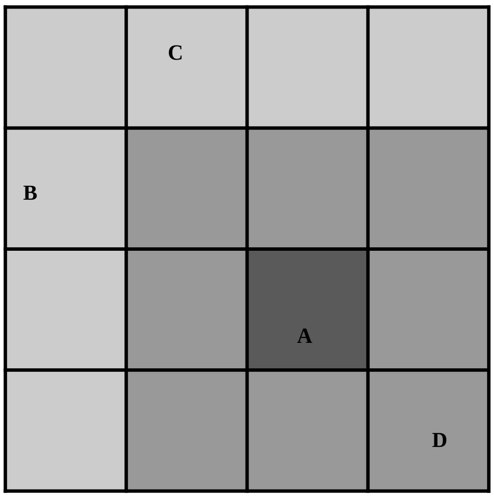
It was a request from Junebud that even though all players might not be rendered, either from being occluded by geometry or from being far away, they should have a marking in the world so that other players can find their position. This lead to the introduction of a level of detail (LOD) system for the position updates. When a player sends a position update to the server, depending on the distance between the partitions, the server only passes on the update to a subset of all partitions. Players within the same partition and players within neighbouring partitions would get all messages, while players in partitions further away get every second or every third message and so on. The system is visualized in Figure 2.

When deciding on how to send messages to clients in a given partition, two different approaches existed. Either could each client listen to exactly one partition and the server send to all neighbouring partitions when an update was sent from the client, or each client could listen to multiple partitions and the server send solely to one partition. The former method requires less overhead when changing partition for a client while the latter method only requires the server to send to one partition and which partitions to listen to can be computed client side. When using Darkstar's channels no good solution could be found for using LOD on the sent messages, which weighed in favour of the former.

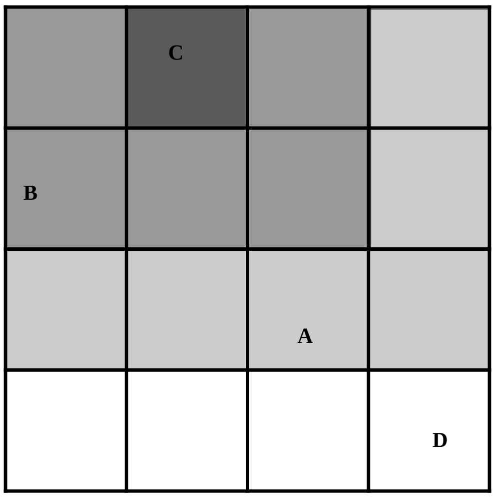
Even though advised against by Chen and Lei [12], the initial partitioning was a fixed-size and static partitioning of the world. Given by Junebud, however, was that MilMo should run on a single server machine, which would greatly reduce the usefulness of a dynamic solution as the server would have to run all computations, nonetheless, plus the extra overhead of updating the partitions. If at a later time the architecture would change to a server cluster, Darkstar's multi node system will do the load balancing automatically.

The partitioning of the game world was done as a regular grid of squares where each partition held data about which players that were inside it and which other partitions that were its neighbours and how often messages should be propagated to their players. Other types of regular grids, such as triangles and hexagons, were also considered but for the first prototype a square grid was chosen to try out the concept of partitioning.

Different sizes of the partitions were tested and where the larger partitions had problems with not reducing the network traffic enough alternatively made the movement look jerky depending on the fall off of the send ratio with the distance between the partitions, where the smaller partitions had a far greater negative impact on the server performance. From this an idea sprung up to use a quadtree for the partitions instead of a regular grid. This would allow for smaller partitions in densely populated areas and larger partitions in sparsely populated areas. It would



(a) Player A gets every position update from player D and every second position update from players B and C.



(b) Player C gets every position update from player B, every second update from player A and every third or no updates from player D depending on settings.

Figure 2: A partitioning of the world with the Level of Detail system.

also make occlusion based culling easier to fit to the partitioning system as a smaller occluded area could be partitioned further to fit into a single or a few partitions. The new system replaced the old as regular grids of squares is a subset of quadtrees.

During this phase, experiments with overlapping partitions were executed to reduce the frequency at which players moving across a partition border would have to update which partition they were located in. The overlapping partitions were, however, abandoned when moving to the quadtree based solution. Also a decision was made to avoid Project Darkstar's channels due to the extra channel data sent with each message and the extra overhead from the channel's join and leave messages. That another developer in the Darkstar community[1] had also come across performance issues when trying to use a large quantity of channels for partitioning the game world helped when deciding against the use of channels.

In Project Darkstar all game logic is executed inside tasks, which are also transactions. For each task, the data needed for the task is read from disk, thus for tasks that are run frequently it is important that the data that needs to be accessed is not too large. Furthermore, if a task tries to modify data shared with another running task, the tasks will be aborted and retried later, making contention an important factor of how well the server will perform. Tasks are also aborted if run for more than a given period of time making it possible to write tasks too long to ever be completed.

The earlier implementations of the partitioning system suffered from both reading and writing a substantial amount of data in each task and frequently modifying data shared between multiple tasks. At this time, however, the reasons were only partially known. At the same time, other new game features slowed down the server to such a state that it was hard to run on an ordinary workstation. Some effort was put into solving the issues with the server but as an upcoming Darkstar version would make a solution to a few of the issues possible, other methods for decreasing the amount of network traffic was implemented in the meantime not to fall too much behind schedule.

In the initial planning three weeks had been put aside for each method, one week of planning and designing, one week of implementing and one week of testing. The work was already at least two weeks behind schedule from the protocol generator and the Partitioning Optimization was still not implemented in fully nor had any major testing taken place and the three designated

[1]The Darkstar forum has been brought down since Oracle's acquisition of Sun, which is why there is no citation.

weeks was up. The existing implementation did not allow for more than just a few players so all further testing was postponed until all the methods had been implemented. Moving all the tests to one point would also allow for more consistency in the testing, which was a positive effect that helped the decision making. To catch up with the schedule and not to get too far behind, focus was switched to the two other methods of reducing the network traffic, which got, and only needed, two weeks in total to be implemented.

5.5 Standing Still

As players have shown tendencies for staying idle for extended periods of time in other MMORGPs this will most likely happen also in MilMo. To benefit from this the standing still optimization was implemented.

Originally, the frequency at which position updates were sent was three per second no matter the player's behaviour. When sending position updates with a lower frequency while standing still, not only less data will be transmitted to other clients, but less work needs to be done by the server. If not sending any messages while standing still and 50% of the population is standing still at any given time, this method obviously has the potential of halving the network traffic.

The implementation of not sending messages while standing still was easily implemented by checking how far the player had moved since the last message was sent and comparing that to a threshold value. If the difference surpasses the threshold, then a message is sent. However, the implementation of interpolating remote player avatars, which hides the jerkiness of the network updates, was dependent on a fixed position update interval and needed a rewrite to work with the standing still optimization. This rewrite prolonged the implementation time of this optimization to about a week.

5.6 Message Optimization

The size of each position update stands in direct relation to the total amount of data transferred. Initially, each position update had a size of 26 bytes. A number of methods to reduce this amount was explored.

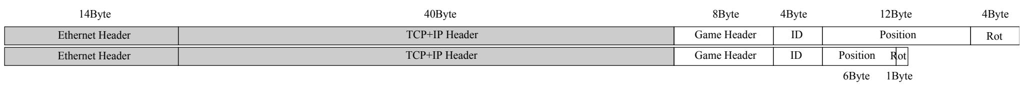
Delta encoding is a compression technique used in a variety of settings including networking, backup systems, source code control and video compression. The idea is to store or send a description of the change from the latest state instead of the current state. Although proven efficient in existing games [3] it was dismissed as it would increase the server's computational and memory load too much. Also, the efficiency of delta encoding in TCP streams is somewhat limited compared to UDP.

Another technique for data compression is Huffman Coding, which is described more in detail in the theory chapter. The symbol frequencies of the player position updates are, however, not known in advance thus the Huffman tree has to be sent with each packet. For small packets like the player position updates, the size of the tree can outweigh the reduction of data size, which is why Huffman Coding was dismissed.

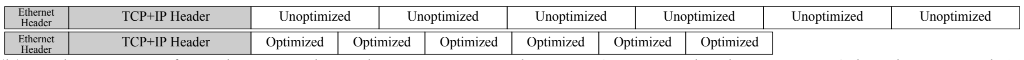
The method finally chosen was moving from a 32-bit floating point representation of the world coordinates to a 16-bit fixed point representation, and reducing the rotation around the y-axis from a 32-bit floating point number to an 8-bit fixed point number. Testing showed that, after some minor tweaking of the fixed point position on the vertical axis, the loss of precision did not affect the gameplay in any noticeable way. The implementation took no more than a day to finish due to its simplicity.

5.7 Test Client

As the number of test players at this time still was very low, it was decided that a test client would be developed to gather the data needed for the analysis of the different methods. Two different behaviours of the client was discussed. In one, the behaviour of the available testers would be recorded and then replayed in multiple instances. In the second, the test client would



(a) 54 bytes out of 66 bytes and 57 bytes respectively are TCP, IP, and Ethernet MAC headers.



(b) 54 bytes out of 222 bytes and 168 bytes respectively are TCP, IP, and Ethernet MAC headers. Nagle's algorithm increases the benefit of reducing the payload size.

Figure 3: TCP overhead when sending a single message compared to sending multiple messages with Nagle's algorithm.

move pseudo randomly within the test area according to a set pattern. The former method was, however, deemed too time consuming to construct and the behaviour of a few players would not necessarily be similar to that of hundreds of players, and thereby perform better than the latter.

A benefit of having a test client for all the tests, instead of running tests with actual players, is that the result is highly reproducible and that all the test cases will have the same setup. Furthermore, it is possible to run hundreds of clients from a single workstation, something that would have been impossible with the ordinary game client.

The test client was implemented in the same language as the regular client, that is in C#, which meant that the existing network client code could be reused and that allowed the test client to be completed in about a week.

Figure 4: The test client in action.

5.8 Partitioning Phase Two

When the Standing Still and Message Optimization were completed, the second phase of the partitioning started. At that time, the new Darkstar version was still not released, however a server machine had been acquired on which the game server could successfully run. A system for profiling the server had also been found, which greatly helped in the identification of the performance issues with the partitioning solution.

The profiler showed that there was much contention between the tasks updating the position of the players. Further analysis showed that the contention occurred when a player changing partition modified the data structures accessed by all players during moving. To reduce the lookup times the current partition of every player was stored inside a map from the player identifier to the partition, and for every position update the partition object was queried if the new position still was inside, and otherwise remove the player from the partition and put the player in the new partition.

To be able to update the position without having to save the current position, the original regular grid was reintroduced, and thus the partition in which the player is located could be calculated without having to access any partition data.

The next problem found by the profiler was that the queue of tasks was constantly growing, slow at first but when a certain limit was reached, the queue size grew rapidly with the halting of the server as a result. This lead to the conclusion that the position update tasks took longer to complete than the time frame of a transaction used in Darkstar. As the tasks never got completed, they would be retried over and over, growing the queue more and more, eventually leading to a halt of the server.

The work performed by the task was to loop through all the neighbouring partitions, and for each partition loop through all the players located inside the partition and send the data to that player. Instead the task was split up in several subtasks, one for each partition and one for each player in their respective partition.

Even though the queue did not grow as fast as before the problem still existed, and given enough players the server still came to a halt. As the tasks themselves performed very little work while each task took more than one millisecond to execute the focus went to the serialization and deserialization of the tasks. For every task, a list of neighbouring partitions had to be deserialized and for each partition a list of players located in it had to be serialized. To speed up task loading these lists were instead put in a static member in a manager of the partitions.

The moving to a static member of the manager greatly reduced the time for each task but the overhead of queuing multiple tasks for every position update now became the dominating factor. Going back to loop through all the neighbouring partitions in a single task was not an option as that solution would not scale very well.

The server was being readied for the upcoming tests simultaneously with the later parts of the partitioning system. As the other server issues still had not been fixed, these systems were plainly disabled. Furthermore, the tasks that were sending messages to all other players via the channels did run faster by several magnitudes, and it was decided to use the channels even though they initially had been dismissed.

Using channels, the partitioning system was able to run successfully with about forty players for long enough time to perform the tests. Some issues with the partitioning system still existed, but as the deadline for the partitioning system was reached, and small tests could be executed, implementation of the partitioning system was ended.

5.9 Data Gathering

A test bed was set up where a dedicated server ran the server application, and two workstations were used to run test clients. The server also ran the tcpdump [4] utility with arguments to capture all data on the application's TCP port.

6 Result

The main result is the gathered data. In addition, the implementations created during the thesis are described, as they are also results.

6.1 Protocol Generator / Dissector

The protocol generator was built for several reasons. Most important was to be able to analyse the network data easily even when the network protocol changes, to keep the protocol specification in one place, and to get a clean interface to the messages. The user of the generated protocols should only have to bother about what to do with the messages, not how to read them from the network stream. In its current state, the protocol generator has all this.

The protocol is specified in an XML file, which is parsed by the generator, and the protocol code for the server and client is generated. In the generated code every message is its own class and its members is accessed through getter functions, rather than previously when a generic message object was passed around and partially read in several different places. When a message is received, it is read in its entirety and its data is stored in the members of the class. After that its message handler is called. The message handlers are template classes for the developer to fill in containing only one method, `handle`, which is overloaded to take as parameters the received message and either the session or the channel that the message arrived on. To send a message, the constructor is called on the message with all the data that should be sent, and then the message is passed to a send function on either a channel or a session.

The protocol specification is made up from messages and data structures, where the former is a special case of the latter. The data structures can consist of built in types, like integers, floating point numbers and strings, other data structures and lists of built in types or other data structures. Furthermore, the data structures also allow for limited inheritance.

A small example of the generated code can be found in Listing 1. For a more extensive example, please refer to Appendix A.

Listing 1: Parts of a constructor created with the protocol generator.

```
this.maxHealth = reader.readInt32();
this.damageSusceptability = new TemplateReference(reader);
this.damageSound = reader.readString();
this.noDamageSound = reader.readString();
this.deathEffectsPhase1 = new ArrayList<String>();
short deathEffectsPhase1Size = reader.readInt16();
for (short i0 = 0; i0 < deathEffectsPhase1Size; i0++)
{
  deathEffectsPhase1.add(reader.readString());
}
this.deathEffectsPhase2 = new ArrayList<String>();
short deathEffectsPhase2Size = reader.readInt16();
for (short i0 = 0; i0 < deathEffectsPhase2Size; i0++)
{
  deathEffectsPhase2.add(reader.readString());
}
```

The design of the code generator is similar to that of a compiler, with a front end that parses the code and performs syntax and type checking, and a back end that generates code. The parser was generated by the java utility XBC, the XML Binding Compiler, which given an XML schema produces all the necessary classes to parse and create an abstract syntax tree (AST). That tree is then type checked and a second AST is created, which is in a format the back end can handle.

The back end and the second AST uses the visitor pattern where the code generator visits the AST. The code generator for each language implements the visitor interface and thus one AST can be used for multiple languages. However, while java and C# are fairly similar languages and could use the same AST, the dissector had to be written in a very specific manner and in C and therefore had its own AST, but still uses the same parser.

By splitting the protocol generator in this way the same front end, with some modifications, can be used for multiple back ends to reduce the amount of work that needs to be done.

Due to time constraints the dissector was left in a working but not fully implemented state. Missing is the parsing of the inherited data structures in the packet stream, but as no messages containing such data structures were of interest in this study it did not cause any problems.

6.2 Partitioning

Many of the planned features had to be put aside due to problems that were encountered while implementing them in a Project Darkstar environment. Most notably the quadtree based partitioning had to be abandoned for a simpler version, and the occlusion based culling of messages was not used, however the system supports it with the use of message LOD.

For the final version of the partitioning, a regular grid of squares is used, and each partition holds data about how often the position updates should be passed on to the neighbouring partitions. How frequently messages are sent between players in different partitions is determined based on the minimum distance between the partitions. Each partition uses its own Darkstar channel, which lets the server send the message to all the players inside it in a single function call.

Because of the performance issues discovered when using Darkstar, all read only data of the partitions is put in a static hash map outside of Darkstar context, where lookups can be performed fast without the need of having to deserialize the data.

When a position update is received at the server, the partition corresponding to that position is calculated and compared to the player's current position. If the partition changed since last update, the player is added to the channel corresponding to the new partition and removed from the channel corresponding to the old partition. Each player object also contains a LOD value which is incremented by one. From the partition in which the player now is located all neighbouring partitions and their respective LOD-values are fetched and for each partition, if the player's LOD value is congruent with 0 modulo the partition's LOD value the message is sent on that channel.

In the final version of the partitioning system made during this thesis, a few performance issues exist that eventually will make the server unresponsive. This topic will be handled further in the discussion section of the report.

In addition to the partitioning system, a small utility for partitioning the world as a quadtree was partly developed, but when the quadtree approach was dropped, so was the utility.

6.3 Standing Still

Based on the idea of players standing still a substantial amount of time, the standing still optimization sends messages less frequent when the players are not moving. Although tweakable, the client sends a message every ten seconds while standing and three times per second while moving.

Whenever a player sends a position update to the server, that position and time are saved client side. The next time a position update is about to be sent, the client computes the distance from the last sent position to the position about to be sent. Only if the distance is above a given threshold is the position update is sent to the server. However, if the time since the last update is greater than another given threshold, the update is sent even though the player has not moved.

6.4 Message Optimization

The message optimization was used to minimize the amount of data sent with each message. Instead of sending the player position as three 32-bit floating point numbers, the position is sent as three 16-bit integers using fixed point precision. The rotation around the Y-axis is sent as a 8-bit discretization of the angle instead of a 32-bit floating point number describing the angle.

This resulted in a message size of 17 bytes plus 54 bytes TCP, IP, and Ethhernet headers. 35% ($\frac{17}{26} = 0.65$) reduction of the position update, which reduces the whole message (including headers) with 11% ($\frac{73}{82} = 0.89$). However, Nagle's algorithm is used to coalesce several small packets into fewer large ones, so that the initial gain of 11% less data sent tends to a 35% gain as the number of packets coalesced tends to infinity. The actual gain measured can be found in the data analysis.

6.5 Test Client

As the test client is based upon the existing client code from the game it uses the same generated protocol. No content from the game is, however, loaded and the test client discards all messages from the server except for the messages needed to log into the world. This way the cost of running the test client is very low.

Through a command line interface when starting the client, the behaviour of the client can be specified, from send rates to how long the test players should be standing still and the boundaries of the level they are playing.

While running, the test client selects a position within the test area with a uniform probability and moves towards that position with a constant given velocity. When arriving at the position, the procedure is repeated. On top of this behaviour, the client can be set to move or stand still for a given amount of time, and also at which rate position updates will be sent for the different modes. An obvious problem with this behaviour is that ordinary players might not spread over the map in this specific pattern, but rather have areas with dense population and sparsely populated areas. Moreover, it is possible that the standing-or-moving behaviour differs between the densly and sparsely populated areas.

6.6 Data

A couple of test cases were set up, listed below, formed by combinations of different bandwidth-saving measures.

- Reference
- Message Optimization
- Standing Still
- Message Optimization + Standing Still
- Partitioning
- Partitioning + Message Optimization
- Partitioning + Standing Still
- Partitioning + Message Optimization + Standing Still

For each of the cases three tests on each level of 10, 20, 30 and 40 users were run, each test lasting at least five minutes. For the cases that did not include Partitioning, additional tests were run with 100 and 150 users but the data from those tests were deemed unreliable due to network and server performance issues when running those tests as well as large deviation between test runs in the captured data.

In the tests, the clients were set to send three messages per second to the server while moving, and one message every ten seconds while standing still, and for the tests on Standing Still Optimization alternating between moving and standing in 30 seconds intervals.

During the tests, all other game logic was turned off better to see the impact the network optimizations had on the general system performance, and also due to that the server at the time was not optimized enough to take the amount of clients the tests demanded.

Following the tests, the data from each run was trimmed to 300 seconds for easier comparison, the specifications of the resulting data files are presented in tables 2, 3, 4, 5, 6, 7, 8 and 9.

Table 2: Reference

(a) 10 users

	First run	Second run	Third run
Number of packets	35506	35894	35827
Data Size (bytes)	4619824	4614402	4614490

(b) 20 users

	First run	Second run	Third run
Number of packets	70896	71692	71404
Data Size (bytes)	14146854	14115254	14124518

(c) 30 users

	First run	Second run	Third run
Number of packets	107173	107851	104854
Data Size (bytes)	28561304	28460984	28569542

(d) 40 users

	First run	Second run	Third run
Number of packets	143114	144312	137455
Data Size (bytes)	47886986	47638714	48365460

Table 3: Message Optimization

(a) 10 users

	First run	Second run	Third run
Number of packets	30330[a]	36309	35843
Data Size (bytes)	3287116[a]	3858204	3837490

(b) 20 users

	First run	Second run	Third run
Number of packets	71599	70544	71084
Data Size (bytes)	10962364	10998802	10946932

(c) 30 users

	First run	Second run	Third run
Number of packets	107533	107414	106860
Data Size (bytes)	21465532	21467109	21408997

(d) 40 users

	First run	Second run	Third run
Number of packets	143038	145278	142386
Data Size (bytes)	48365460	35377148	35178487

[a]While the other runs were 300 seconds or longer, this run was cut short at 258 seconds.

Table 4: Standing Still

(a) 10 users

	First run	Second run	Third run
Number of packets	24169	24172	26753
Data Size (bytes)	2727350	2726858	2886418

(b) 20 users

	First run	Second run	Third run
Number of packets	56204	56079	56025
Data Size (bytes)	8389008	8396590	8378660

(c) 30 users

	First run	Second run	Third run
Number of packets	93875	93436	89323
Data Size (bytes)	16962032	16890510	17011224

(d) 40 users

	First run	Second run	Third run
Number of packets	127132	127274	122217
Data Size (bytes)	27927750	27825372	27706152

Table 5: Message Optimization + Standing Still

(a) 10 users

	First run	Second run	Third run
Number of packets	25455	24208	21347
Data Size (bytes)	2389485	2315489	2113811

(b) 20 users

	First run	Second run	Third run
Number of packets	56452	56910	56146
Data Size (bytes)	6878508	6863986	6855544

(c) 30 users

	First run	Second run	Third run
Number of packets	94888	94776	93148
Data Size (bytes)	13519284	13348633	13281474

(d) 40 users

	First run	Second run	Third run
Number of packets	122712	127077	126252
Data Size (bytes)	21260590	21357942	21287299

Table 6: Partitioning

(a) 10 users

	First run	Second run	Third run
Number of packets	28984	29251	29190
Data Size (bytes)	2131563	2170764	2139206

(b) 20 users

	First run	Second run	Third run
Number of packets	61318	62391	60756
Data Size (bytes)	4588766	4689667	4505330

(c) 30 users

	First run	Second run	Third run
Number of packets	96285	94677	96284
Data Size (bytes)	7384247	7357303	7482321

(d) 40 users

	First run	Second run	Third run
Number of packets	132264	132981	131327
Data Size (bytes)	10490926	10576142	10359337

Table 7: Partitioning + Message Optimization

(a) 10 users

	First run	Second run	Third run
Number of packets	28821	29280	29348
Data Size (bytes)	2030890	2048409	2041758

(b) 20 users

	First run	Second run	Third run
Number of packets	61574	61740	62013
Data Size (bytes)	4385396	4370814	4406663

(c) 30 users

	First run	Second run	Third run
Number of packets	96499	96262	97760
Data Size (bytes)	6980789	6886015	7036508

(d) 40 users

	First run	Second run	Third run
Number of packets	131410	132593	120457
Data Size (bytes)	9761056	9808229	9092798

Table 8: Partitioning + Standing Still

(a) 10 users

	First run	Second run	Third run
Number of packets	15416	15931	15515
Data Size (bytes)	1131156	1241131	1139901

(b) 20 users

	First run	Second run	Third run
Number of packets	33967	35617	33552
Data Size (bytes)	2555943	2665292	2475296

(c) 30 users

	First run	Second run	Third run
Number of packets	54927	58214	56374
Data Size (bytes)	4159487	4396537	4222582

(d) 40 users

	First run	Second run	Third run
Number of packets	78526	84583	83483
Data Size (bytes)	6284290	6394621	6328266

Table 9: Partitioning + Message Optimization + Standing Still

(a) 10 users

	First run	Second run	Third run
Number of packets	15221	16792	14888
Data Size (bytes)	1063900	1183834	1032317

(b) 20 users

	First run	Second run	Third run
Number of packets	31494	34819	33641
Data Size (bytes)	2202318	2451061	2350928

(c) 30 users

	First run	Second run	Third run
Number of packets	57073	57004	56126
Data Size (bytes)	4007998	4013001	3963713

(d) 40 users

	First run	Second run	Third run
Number of packets	83335	83603	84662
Data Size (bytes)	5993756	5918438	5986166

7 Analysis

The data was analysed with Wireshark and the average datarate for each level of users, listed in table 11, was plotted in figure 5. The mean deviation, σ, was calculated for each combination of optimization methods and level of users and turned out to be low for most of the tests. Also, the average packet size for each level of users was plotted in figure 8.

Additionally, the data was cut into two pieces, incoming and outgoing traffic, which were analysed separately and their average data rate, listed in tables and , were plotted in figure 6 and 7 respectively. Figures 5 and 7 are quite similar because the main body of the traffic is going out from the server.

Looking at the reduction of the mean byte rates in figure 5 it is clear that of all the different methods, the partitioning optimization is the one method that reduced the outgoing traffic the most. The standing still optimization is the second best single method, but even standing still along with message optimization doesn't reach the reduction of the partitioning optimization alone. In comparison with the other methods, the partitioning is the only method that significantly reduced the quadratic coefficient of the quadratic behaviour when the number of users increases, which can be seen in table 10.

Table 10: Interpolation of the mean byte rate (in kb/s) fitted to a quadratic polynomial $y = a + b \times x + c \times x^2$

Method	a	b	c
Reference	0.03489	0.30055	0.08036
Message Optimization	0.00288	0.31552	0.05406
Standing Still	-0.12644	0.24462	0.04260
MO + SS	-0.08681	0.24061	0.02928
Partitioning	-0.02848	0.27155	0.00494
PT + MO	-0.03205	0.25533	0.00384
PT + SS	0.03918	0.13083	0.00358
PT + MO + SS	0.04171	0.10328	0.00342

As for the incoming data, the test cases in figure 6 can be divided into three distinct groups; combinations with both the partitioning and the standing still optimization, combinations with the standing still optimization without the partitioning optimization and the rest of the combinations. Inside these groups, the combinations have roughly the same results in the tests. The standing still optimization obviously had the largest impact on the client to server traffic because it makes the client send updates less often. The reason that the partitioning optimization lowers the traffic so much in cooperation with the standing still optimization, but hardly affects the traffic amount without it, is that the partitioning optimization lowers the number of packets sent to each client. This lower amount of packets going from the server to the client means that when standing still the client needs to send less ACK packets. However, this doesn't significantly affect the combinations without the standing still optimization, since in those combinations the client is sending packets at such a rate that most ACK's can be sent in packets also containing a position update.

Table 11: Total Traffic

(a) Average Byte Rate for 10 Users (kbyte/s)

Method	First run	Second run	Third run	Mean	σ
Reference	15.0397	15.0226	15.0231	15.0285	0.0079
Message Optimization (MO)	12.4503	12.5599	12.4936	12.5013	0.0451
Standing Still (SS)	8.8793	8.8771	9.4341	9.0635	0.2620
MO + SS	7.7787	7.5386	7.9755	7.7643	0.1786
Partitioning (PT)	6.9392	7.0683	6.9637	6.9904	0.0560
PT + MO	6.6114	6.6696	6.6482	6.6431	0.0241
PT + SS	3.6824	4.1642	3.7108	3.8525	0.2207
PT + MO + SS	3.4645	3.8540	3.3605	3.5597	0.2124

(b) Average Byte Rate for 20 Users (kbyte/s)

Method	First run	Second run	Third run	Mean	σ
Reference	46.0526	45.9538	45.9847	45.9970	0.0413
Message Optimization (MO)	35.6874	35.8044	35.6392	35.7103	0.0694
Standing Still (SS)	27.5758	27.5664	27.6332	27.5918	0.0295
MO + SS	22.3919	22.3447	22.3171	22.3512	0.0309
Partitioning (PT)	14.9381	15.2664	14.6690	14.9578	0.2443
PT + MO	14.2769	14.2286	14.3453	14.2836	0.0479
PT + SS	8.3205	8.6767	8.1670	8.3881	0.2135
PT + MO + SS	7.2552	7.9794	7.6529	7.6292	0.2961

(c) Average Byte Rate for 30 Users (kbyte/s)

Method	First run	Second run	Third run	Mean	σ
Reference	92.9820	92.6498	93.0052	92.8790	0.1624
Message Optimization (MO)	69.8782	69.8830	69.6932	69.8181	0.0884
Standing Still (SS)	55.6713	55.4958	55.9629	55.7100	0.1926
MO + SS	44.0099	43.5385	43.2724	43.6069	0.3049
Partitioning (PT)	24.0387	23.9501	24.3572	24.1153	0.1748
PT + MO	22.7240	22.4160	22.9061	22.6820	0.2023
PT + SS	13.5708	14.3473	13.7756	13.8979	0.3286
PT + MO + SS	13.1934	13.1722	12.9365	13.1007	0.1165

(d) Average Byte Rate for 40 Users (kbyte/s)

Method	First run	Second run	Third run	Mean	σ
Reference	155.8887	155.0781	157.4466	156.1378	0.9828
Message Optimization (MO)	114.7014	115.1700	114.5189	114.7968	0.2742
Standing Still (SS)	91.3401	90.5864	90.3396	90.7554	0.4256
MO + SS	69.2454	69.5931	69.3667	69.4018	0.1441
Partitioning (PT)	34.1502	34.4277	33.7221	34.1000	0.2903
PT + MO	31.7760	31.9311	30.5091	31.4054	0.6370
PT + SS	20.4905	20.9057	20.6775	20.6912	0.1698
PT + MO + SS	19.5282	19.3266	19.4894	19.4481	0.0873

Table 12: Incoming Traffic

(a) Average Byte Rate for 10 Users (kbyte/s)

Method	First run	Second run	Third run	Mean	σ
Reference	3.8693	3.9033	3.9138	3.8955	0.0190
Message Optimization (MO)	3.8600	3.9817	3.9462	3.9293	0.0511
Standing Still (SS)	2.5879	2.5806	2.8658	2.6781	0.1328
MO + SS	2.7153	2.5835	2.6981	2.6656	0.0585
Partitioning (PT)	3.8266	3.8214	3.8367	3.8282	0.0064
PT + MO	3.6990	3.8406	3.8961	3.8119	0.0830
PT + SS	2.0067	2.1597	2.0338	2.0667	0.0667
PT + MO + SS	1.9714	2.1188	2.0326	2.0409	0.0605

(b) Average Byte Rate for 20 Users (kbyte/s)

Method	First run	Second run	Third run	Mean	σ
Reference	7.7134	7.7975	7.7822	7.7643	0.0366
Message Optimization (MO)	7.8127	7.7703	7.7489	7.7773	0.0265
Standing Still (SS)	5.9782	5.9412	5.9750	5.9648	0.0168
MO + SS	5.9792	6.0542	5.9290	5.9875	0.0515
Partitioning (PT)	7.6604	7.7534	7.6764	7.6968	0.0406
PT + MO	7.6094	7.6842	7.6481	7.6472	0.0306
PT + SS	4.2215	4.4449	4.2514	4.3059	0.0990
PT + MO + SS	4.0647	4.3327	4.2393	4.2122	0.1111

(c) Average Byte Rate for 30 Users (kbyte/s)

Method	First run	Second run	Third run	Mean	σ
Reference	11.6530	11.7203	11.4566	11.6099	0.1119
Message Optimization (MO)	11.7120	11.7270	11.6408	11.6933	0.0376
Standing Still (SS)	9.8920	9.8274	9.4645	9.7280	0.1881
MO + SS	9.9476	9.9433	9.7305	9.8738	0.1014
Partitioning (PT)	11.5375	11.0675	11.5411	11.3821	0.2224
PT + MO	11.4131	11.5791	11.7082	11.5668	0.1208
PT + SS	6.5665	6.9811	6.8029	6.7835	0.1698
PT + MO + SS	6.9512	6.9815	6.7982	6.9103	0.0802

(d) Average Byte Rate for 40 Users (kbyte/s)

Method	First run	Second run	Third run	Mean	σ
Reference	15.5645	15.6732	15.1086	15.4487	0.2446
Message Optimization (MO)	15.5839	15.9190	15.5107	15.6712	0.1778
Standing Still (SS)	13.3493	13.2496	12.7582	13.1191	0.2584
MO + SS	12.9085	13.2615	13.2060	13.1253	0.1550
Partitioning (PT)	15.4427	15.4840	15.3808	15.4358	0.0424
PT + MO	15.2872	15.4300	14.6162	15.1111	0.3548
PT + SS	9.2675	9.9204	9.7687	9.6522	0.2790
PT + MO + SS	9.6568	9.8241	9.8688	9.7832	0.0912

Table 13: Outgoing Traffic

(a) Average Byte Rate for 10 Users (kbyte/s)

Method	First run	Second run	Third run	Mean	σ
Reference	11.1706	11.1193	11.1094	11.1331	0.0268
Message Optimization (MO)	8.5903	8.5782	8.5474	8.5720	0.0181
Standing Still (SS)	6.2914	6.2965	6.5721	6.3867	0.1311
MO + SS	5.0634	4.9552	5.2775	5.0987	0.1339
Partitioning (PT)	3.1130	3.2469	3.1271	3.1623	0.0601
PT + MO	2.9125	2.8294	2.7523	2.8314	0.0654
PT + SS	1.6758	2.0045	1.6772	1.7858	0.1546
PT + MO + SS	1.4936	1.7352	1.3281	1.5189	0.1672

(b) Average Byte Rate for 20 Users (kbyte/s)

Method	First run	Second run	Third run	Mean	σ
Reference	38.3392	38.1564	38.2025	38.2327	0.0777
Message Optimization (MO)	27.8747	28.0342	27.8904	27.9331	0.0718
Standing Still (SS)	21.5976	21.6265	21.6609	21.6283	0.0259
MO + SS	16.4127	16.2905	16.3881	16.3638	0.0528
Partitioning (PT)	7.2777	7.5131	6.9926	7.2611	0.2128
PT + MO	6.6675	6.5444	6.6972	6.6364	0.0661
PT + SS	4.0990	4.2319	3.9159	4.0823	0.1296
PT + MO + SS	3.1907	3.6468	3.4135	3.4170	0.1862

(c) Average Byte Rate for 30 Users (kbyte/s)

Method	First run	Second run	Third run	Mean	σ
Reference	81.3292	80.9295	81.5486	81.2691	0.2563
Message Optimization (MO)	58.1662	58.1560	58.0524	58.1249	0.0514
Standing Still (SS)	45.7824	45.6707	46.5046	45.9859	0.3696
MO + SS	34.0623	33.5952	33.5419	33.7331	0.2338
Partitioning (PT)	12.5012	12.8829	12.8161	12.7334	0.1665
PT + MO	11.3118	10.8378	11.1979	11.1158	0.2020
PT + SS	7.0045	7.3662	6.9728	7.1145	0.1784
PT + MO + SS	6.2432	6.1910	6.1383	6.1908	0.0428

(d) Average Byte Rate for 40 Users (kbyte/s)

Method	First run	Second run	Third run	Mean	σ
Reference	140.3243	139.4049	142.3380	140.6891	1.2249
Message Optimization (MO)	99.1175	99.2510	99.0082	99.1256	0.0993
Standing Still (SS)	77.9908	77.3368	77.5814	77.6363	0.2698
MO + SS	56.3381	56.3317	56.1608	56.2768	0.0821
Partitioning (PT)	18.7075	18.9438	18.3413	18.6642	0.2479
PT + MO	16.4891	16.5018	15.8929	16.2946	0.2841
PT + SS	11.2230	10.9854	10.9094	11.0393	0.1336
PT + MO + SS	9.8714	9.5025	9.6206	9.6649	0.1538

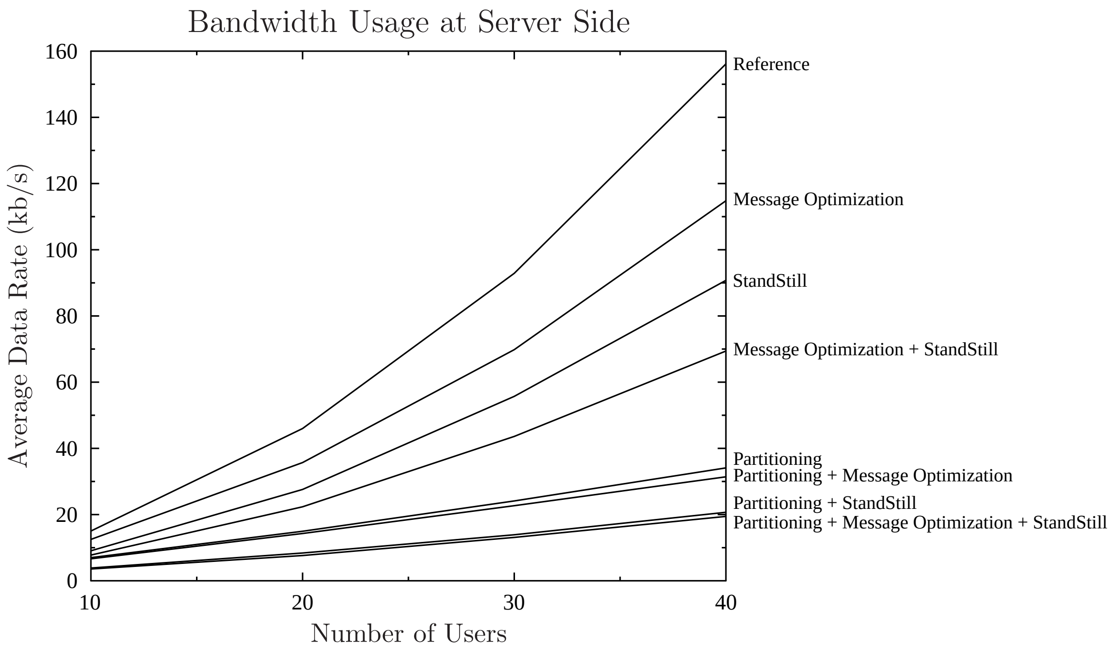


Figure 5: Mean data rate for different test cases.

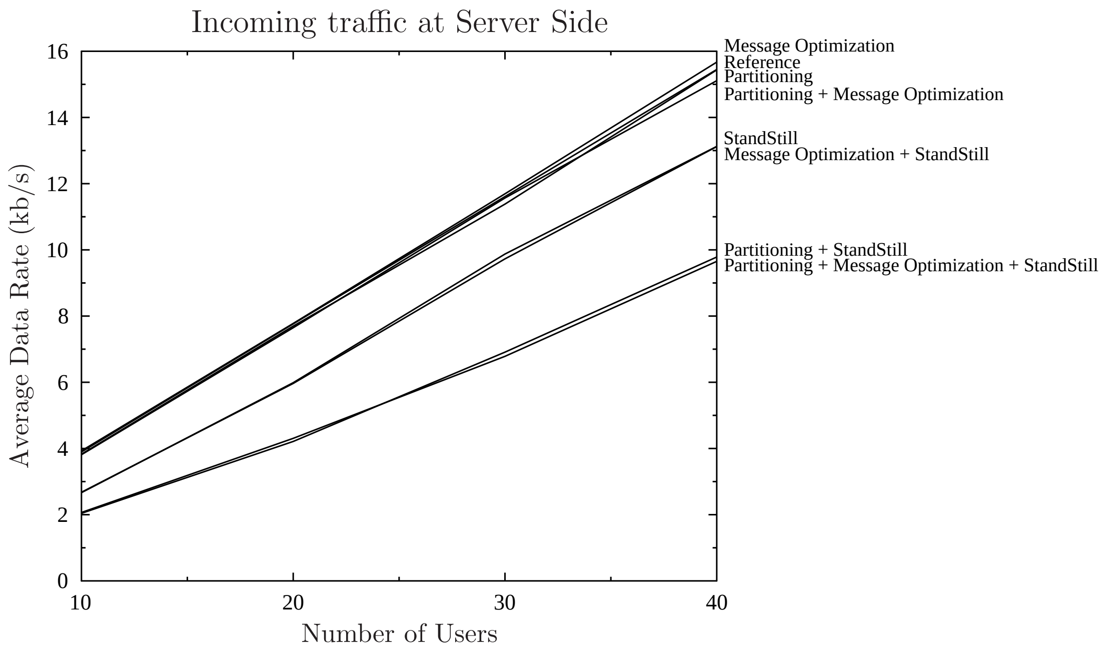


Figure 6: Mean data rate that the server is receiving at for different test cases.

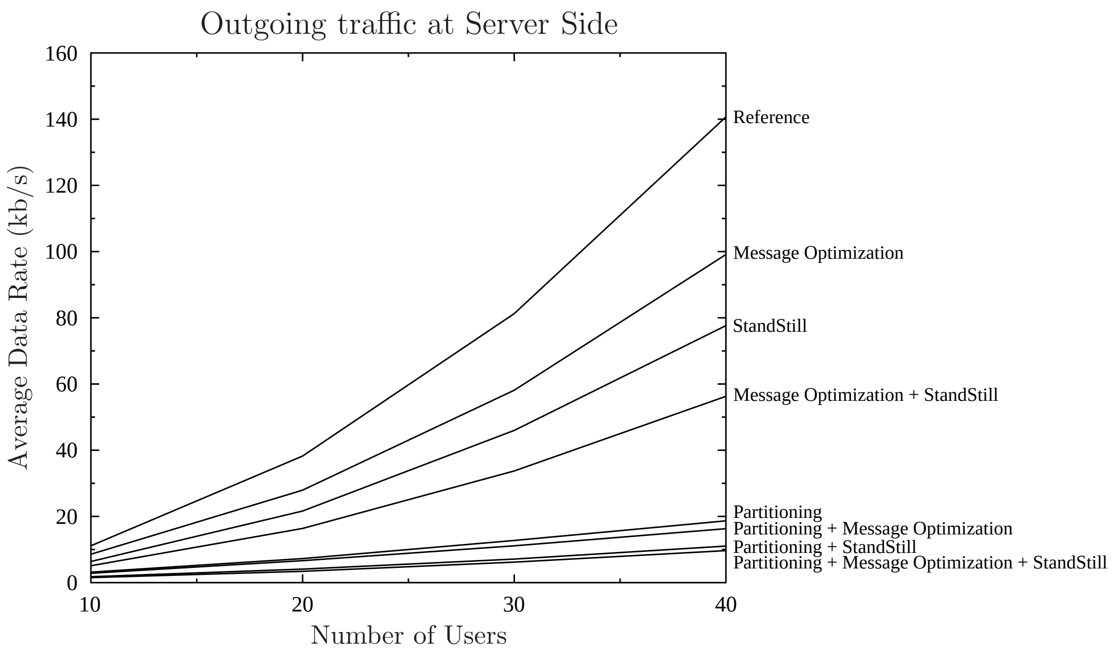


Figure 7: Mean data rate that the server is sending at for different test cases.

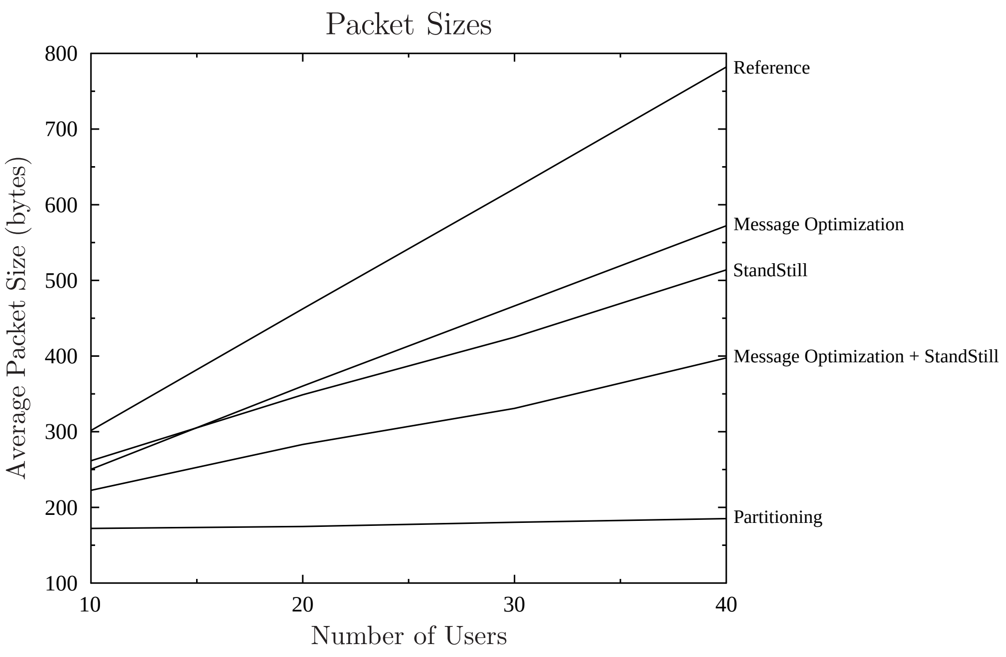


Figure 8: Mean packet size for different test cases.

8 Discussion

The purpose of this thesis was to explore different methods of reducing the server-to-client traffic in the game MilMo. Three different methods were implemented and evaluated, a spatial partitioning of the world, a method where fewer messages were sent when standing still, and a method where the precision of the players' positions had been reduced.

All the implementations reduced the network traffic but with how much differed widely between the methods and with the spatial partitioning greatly surpassing the other methods.

8.1 Results

As the data gathered during the thesis was produced by the use of test clients, the data was assumed to be very predictable and that it would deviate only little from the mean. This was also true with the exception for when the *Standing Still Optimization* was used, which slightly increased the standard deviation, especially in the tests with few clients. As the test client alternated between moving and standing still, a small change in the time frame of the test runs could either increase or decrease the average bitrate depending on the current state of the clients.

A setback in the data gathering phase was that the larger tests could not be used. Inefficiencies in the server code lead to that larger tests could not be executed at all for the *Partitioning Optimization* and for the other optimizanions, the large deviations made the tests unreliable. The large deviation can be ascribed both to the server being overloaded, but also that the clients was connected to the server via a wireless connection.

Also, as stated earlier, the behaviour of the test clients differs from that of human players in many aspects and thus the data gathered might differ greatly from the network traffic generated with the same amount of human players.

8.1.1 Partitioning Optimization

The Partitioning optimization is the only method of those tested that significantly reduced the quadratic coefficient of the network traffic as a function of the number of players. This is especially important for an MMOG, since the aim of any MMOG is to have many players. However, it is also the optimization that took most time to implement, partially due to the difficulties experienced with Project Darkstar but also because it is the optimization with the highest complexity. Some games have spatial partitioning built into the game by the designers, as levels or areas, and may therefore not even need to look at further partitioning.

8.1.2 Standing Still Optimization

The standing still optimization was unique among the optimization methods in that, that it did not only reduce the network traffic but also reduced the load on the server in general. That it did not reduce the network traffic with as much as 50% could be ascribed to a few different factors. First of all, the client did send messages while standing still, but at a lower frequency, and secondly the packet header is not reduced in size. From the test data, one can also see that the higher the number of users, the greater is the reduction, which is a result from the headers being a lesser part of the total data sent.

8.1.3 Message Optimization

There are many ways messages can be decreased in size. While this thesis explored only reduction of precision, message sizes can also be reduced by delta compression, Huffman coding, and other techniques. Lossless compression techniques, like Huffman coding, can be applied directly. However, such application needs to be evaluated to make sure that the compression doesn't increase the message size instead because of added headers or other metadata. As for lossy compression, such as reducing the precision of numbers, positions, etc., the precision can only be reduced to a certain level before it starts to affect the gameplay.

Irrespective of how message sizes are reduced, a message on the Internet is still a minimum of 54 bytes when using TCP or 42 bytes when using UDP[2], whether it has a payload or not. Because of this, reducing the message sizes only plays a limited role in reducing the network traffic, thus for many games the number of messages sent are more important. However, if a coalescing scheme is used, either Nagle's algorithm or a game specific implementation, the payload size can outweigh the header size and thus play a greater role in the overall traffic size.

8.1.4 Combinations of Solutions

The greatest weakness of the Partitioning optimization is that if several players are located close to each other, so that they are in the same partition, no reduction in traffic will occur. Under the assumption that players close to each other are more prone to be standing still interacting than running around, the Standing Still optimization is a great complement to the Partitioning optimization. This is, however, related to player behaviour and is outside the scope of this thesis.

How well the *Message Optimization* works is highly dependent on how well Nagle's algorithm can coalesce the messages. Sending fewer messages due to the *Partitioning Optimization* or the *Standing Still Optimization* can hence reduce the effectiveness of the *Message Optimization*. Disabling Nagle's algorithm and implementing a packing routine that is better suited would counteract that reduction in efficiency.

8.1.5 Protocol Generator

The protocol generator was created early during the start up phase of the thesis, and helped a lot with the understanding of the system in place. When the thesis started the plan was to run the tests with a fairly large set of players thus making the protocol generator and the Wireshark plug-in that was developed with it more useful to separate the position updates than it ended up being with the test clients running and every other system deactivated. The system could, however, easily be used together with other games that use the Project Darkstar game server and C# as the game client. Without too much effort the generator would be able to use languages similar to Java or C# for the game client, and with a bit more effort, other languages as well. As how the protocol is sent over the network is already given, the same Wireshark plug-in could be used to analyse the network traffic.

8.2 Methodology and Planning

The method of working with short iterations and re-planning frequently was advantageous in this project due to its ability to counteract the influence of external issues on the project as well as issues that arose due to the limited knowledge about the software utilized in the project. It allowed focus to be put on the things that were found to be in need of more time, such as the protocol generator and the partitioning optimization.

There were drawbacks though, mainly that each re-planning had a tendency to reduce the amount of time for later tasks, such as the writing, in favour of the tasks at hand. After re-planning a few times, the time for writing was substantially reduced.

To put all the testing in the end of the thesis was advantageous as the overhead of changing focus was reduced and that the different tests were more consistent than they would have been otherwise. The postponement of the testing also allowed for the splitting of the partitioning implementation into two phases, thus being able to work with the two other implementations in between. As a consequence of this, however, little knowledge was carried over between the different implementations and after the tests, there was no time to change the implementations based on the result.

[2]Including the Ethernet MAC header, IP header, and TCP/UDP header. The IP and TCP headers may vary in size depending on options, the values presented here are the minimum sizes.

As the number of players during the thesis was low at all times, a test client was developed to simulate the players. The thought behind the initial planning was to gather data on the beta servers during the weekends following the implementation of a method. This proved to be unfeasible due to the small number of active players at the time, which forced the decision to use a test client. The test client moved the schedule even further but allowed the tests to be executed in a much more controlled manner and thus giving reliable data faster. Although the data could be more reliably gathered with the use of the test client, the behaviour of the test client would not correspond to that of real players in some regards, mainly the amount of movement and the locality of the movement.

About halfway into the thesis work the authors were recruited by Junebud to work two days a week on tasks not directly related to the thesis. This made the already compressed time plan inoperable and thus the deadline for the thesis was advanced.

8.3 Project Darkstar

Project Darkstar was one of the factors that affected the work process and decision making the most during this thesis. The choice of protocol was one decision based on what was at hand in Darkstar. To use UDP with Darkstar would have required to delve into the internal API of Darkstar, which was deemed time consuming, and take time from more important tasks. TCP is, however, used in several existing MMOGs so the decision to stick with TCP would only be limiting in what would be tested.

The transactional tasks of Darkstar come with three issues that needs to be handled for a game to perform well. The contention needs to be minimized, and a good step towards this is to whenever applicable split up tasks so that they only access the data needed to perform a sub task of a larger task. The lesser the amount of data accessed or modified in a single task the smaller the chance is of another task modifying the same data at the same time. This means that looping through large lists of objects and modifying these should be avoided as far as possible. Looping through large lists is also connected to the other two issues that come with Darkstar. To achieve a short average response time, if a task is not executed before a given timeout, it is aborted and retried later, to allow other tasks to run. Looping through large lists usually means that a lot of work will be executed, which could have the result of tasks that will never commit as they will always take longer time to run than the timeout is set to. Finally, as the data is saved to disk between each task, having large lists will increase the time to load and save the data used in the tasks, which can hurt performance.

What has been discovered during the work at Junebud since the end of the implementations for the thesis is that sending on the same channel from multiple tasks can also be the source of a lot of contention, to such a degree that for 50 players sending three updates per second only a tenth of the tasks are successful.

8.4 Usefulness of the Implementations

The concepts used in this thesis to reduce the network load would not only be beneficial for *MilMo* but can also be applied to other games as well. If not all players in a game world need to know about every other player, a partitioning of the world will be effective. The LOD system for messages could, however, be inappropriate for the FPS genre, as such gameplay tends to have a higher demand on always having the correct game state for the parts of the game state that is known at all.

Many games already have systems for culling objects that cannot be seen by the player, and a similar system, such as the one implemented in this thesis, could be used for network messages. Indoor environments with lots of walls and furniture could greatly benefit from such a system, as large parts of the environment would be invisible to the player. In this case, the LOD system for messages could be used to produce sound effects in the world as sound is hard for humans to pinpoint exactly, especially in those cases when surround sound is not used.

How much of the players' movement that can be culled from the network traffic is also dependent on the how the player can view the game world. A first person or third person view would allow the player to see over large distances, so to reduce the traffic a maximum viewing distance has to be established. In comparison to a game using a top down view, what a player can see is limited to what can be fitted onto the screen.

As the tests were performed using the test client, both the result of the spatial partitioning and the standing still optimization will most likely differ from the results given real players, in that, that real players behave differently both in terms of their spatial location and when and where they will be standing still. Furthermore, the tests were not executed with as many clients as the servers are intended to handle.

8.5 Recommendations for *MilMo*

Extrapolating from the test data, for 100 players per level and with 10 levels the monthly outgoing bandwidth sums up to 20.1 TB without any optimizations. The best combination without Partitioning is the combination of the Message and Standing Still optimizations, this combination has an estimated monthly bandwidth usage of 7.65 TB for the same scenario. With the *Partitioning Optimization* in place along with Standing Still and *Message Optimization*, the estimated bandwidth usage drops as low as 1.1 TB.

What should not be forgotten is what can be achieved from instancing of the different levels, allowing a lower number of players per instance while having more levels. With 40 players per level, the maximum of players used in the tests, and 25 levels or instances there of, the three numbers drop to 8.5 TB, 3.4 TB and 0.6 TB respectively.

The reduction of the bandwidth needed for the player position updates is reduced by a factor 18.7 when using all optimization schemes explored in this thesis, and when adding level instancing, the bandwidth is reduced by a factor 34.5. We would therefore recommend that Junebud implements a partitioning scheme for their levels and, if it would not reduce the gaming experience, reduce the number of players per level and add multiple instances of each level.

As the values presented above are extrapolated from the gathered test data, their accuracy is unknown.

8.6 Future Work

What was not extensively studied in this thesis is how the different optimizations affected the visual representations in the game world. The less information that is sent from the server to the clients, the less accurate is the client's game state. This is especially true when using the LOD system in the partitioning optimization, as sending only a subset of the messages could make the movement look jerky.

Occlusion based LOD of the network messages is something that would be a worthy candidate for future work as it could help reduce the network traffic even more without reducing the quality of the movement observed in the game.

Different data structures for the partitioning, such as a quadtree or a grid, were tested in this thesis but most had to be abandoned due to implementation difficulties together with the Darkstar API. Given more time it would be interesting to see how well the quadtree would perform compared to the grid based partitioning, and also if there will be much difference between a hexagonal grid and a quadratic. All the methods studied in this thesis were static in order not to take up too much server resources, however a dynamically updating partitioning could theoretically reduce the network traffic even more.

All the tests were run with a test client, which will not behave as a human player. Therefore, it would be of interest to analyse the gain of the different methods when real players are playing the game. A difficulty that has to be conquered in the case of real players is to get reliable data as the reproducibility would be less than when using the test clients developed in this thesis.

9 Conclusion

The purpose of this thesis was to find and implement different approaches to reduce the network traffic generated by the player position updates in the game *MilMo*. The reduction in network traffic was then to be compared to the original implementation.

Although the data that was gathered is not as extensive as planned with regard to the number of users, it clearly shows one most important thing, which is that no matter what other approaches one takes to reduce server-to-client traffic one has to reduce the number of clients that needs to communicate with each other if one wants to support a large number of players. This can be achieved by the use of space partitioning. For some games, the partitioning can be done as levels, or even worlds, and that will be sufficient for the required number of players while other games may need a fine-grained, well tuned partitioning scheme. *MilMo* is one such game. At the moment, *MilMo* is working fine being partitioned only in different levels, but to allow for the action type gameplay to evolve and the number of users per level to rise, a more fine-grained partitioning is needed, such as the one explored in the *Partitioning Optimization*.

In general, if one wants to decrease the network traffic for an MMOG, one should not only consider to optimize the size of the data structures sent on the network, but also how often these packets are sent, as evident by the fact that the *Standing Still* had a larger impact than the *Message Optimization*.

References

[1] http://portforward.com/.

[2] List of TCP and UDP port numbers. http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers.

[3] Quake 3 Networking. http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/Quake3Networking.

[4] Tcpdump. http://www.tcpdump.org.

[5] BECK, K. AND OTHERS. Manifesto for agile software development. http://www.agilemanifesto.org/.

[6] BETTNER, P., AND TERRANO, M. 1500 archers on a 28.8: Network programming in age of empires and beyond. http://www.gamasutra.com/view/feature/3094/1500_archers_on_a_288_network_.php.

[7] BLIZZARD ENTERTAINMENT. StarCraft. http://us.blizzard.com/en-us/games/sc/index.html?rhtml=y.

[8] BLIZZARD ENTERTAINMENT. World of Warcraft. http://www.worldofwarcraft.com.

[9] BRADEN, R. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), Oct. 1989. Updated by RFCs 1349, 4379.

[10] CCP. EVE Online. http://www.eveonline.com.

[11] CHEN, K.-T., HUANG, P., HUANG, C.-Y., AND LEI, C.-L. Game traffic analysis: an mmorpg perspective. In *NOSSDAV '05: Proceedings of the international workshop on Network and operating systems support for digital audio and video* (New York, NY, USA, 2005), ACM, pp. 19–24.

[12] CHEN, K.-T., AND LEI, C.-L. Network game design: hints and implications of player interaction. In *NetGames '06: Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games* (New York, NY, USA, 2006), ACM, p. 17.

[13] DAINOTTI, A., PESCAPE, A., AND VENTRE, G. A packet-level traffic model of starcraft. In *HOT-P2P '05: Proceedings of the Second International Workshop on Hot Topics in Peer-to-Peer Systems* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 33–42.

[14] DRAIN, B. EVE Evolved: EVE Online's server model. http://www.massively.com/2008/09/28/eve-evolved-eve-onlines-server-model/.

[15] EPIC GAMES AND DIGITAL EXTREMES. Unreal Tournament. http://www.unrealtournament.com.

[16] FUNKHOUSER, T. A. Ring: a client-server system for multi-user virtual environments. In *SI3D '95: Proceedings of the 1995 symposium on Interactive 3D graphics* (New York, NY, USA, 1995), ACM, pp. 85–ff.

[17] GRIWODZ, C., AND HALVORSEN, P. The fun of using tcp for an mmorpg. In *NOSSDAV '06: Proceedings of the 2006 international workshop on Network and operating systems support for digital audio and video* (New York, NY, USA, 2006), ACM, pp. 1–7.

[18] HUANG, G., YE, M., AND CHENG, L. Modeling system performance in mmorpg. *GlobeCom Workshops 2004. IEEE Global Telecommunications Conference Workshops, 2004.* (Dec. 2004), 512.

[19] HUFFMAN, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE 40*, 9 (January 2007), 1098–1101.

[20] ID SOFTWARE. DOOM. http://www.idsoftware.com/games/doom/doom-ultimate/, 1983.

[21] ID SOFTWARE. Quake 3 Arena. http://www.idsoftware.com/games/quake/quake3-arena/, 1999.

[22] JESSE ARONSON. Dead Reckoning: Latency Hiding for Networked Games. http://www.gamasutra.com/view/feature/3230/dead_reckoning_latency_hiding_for_.php.

[23] KINICKI, J., AND CLAYPOOL, M. Traffic analysis of avatars in second life. In *NOSSDAV '08: Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video* (New York, NY, USA, 2008), ACM, pp. 69–74.

[24] LINDEN RESEARCH INC. Second Life. http://secondlife.com/, 2003.

[25] MICROSOFT. Age of Empires. http://www.microsoft.com/games/empires/, 1997.

[26] NAGLE, J. Congestion control in IP/TCP internetworks. RFC 896, Jan. 1984.

[27] STEED, A., AND ABOU-HAIDAR, R. Partitioning crowded virtual environments. In *VRST '03: Proceedings of the ACM symposium on Virtual reality software and technology* (New York, NY, USA, 2003), ACM, pp. 7–14.

[28] STEVE COLLEY. Maze War, 1974.

[29] STUART CHESHIRE. TCP Performance problems caused by interaction between Nagle's Algorithm and Delayed ACK. http://www.stuartcheshire.org/papers/NagleDelayedAck/.

[30] SVOBODA, P., KARNER, W., AND RUPP, M. Traffic analysis and modeling for world of warcraft. In *Communications, 2007. ICC '07. IEEE International Conference on* (2007), pp. 1612–1617.

[31] SWEENEY, T. Unreal Networking Architecture. http://unreal.epicgames.com/Network.htm.

[32] SZABÓ, G., VERES, A., AND MOLNÁR, S. On the impacts of human interactions in mmorpg traffic. *Multimedia Tools Appl. 45*, 1-3 (2009), 133–161.

[33] WIRESHARK FOUNDATION. Wireshark. http://www.wireshark.org.

A Protocol Generator

A.1 The specification of the AvatarCreated message

```
<message>
  <name>ServerAvatarCreated</name>
  <sender>server</sender>
  <var>
    <name>avatar</name>
    <type>Avatar</type>
  </var>
  <var>
    <name>inventory</name>
    <generic>
      <type>list</type>
      <params>
        <type>InventoryEntry</type>
      </params>
    </generic>
  </var>
</message>
```

A.2 The generated code for the AvatarCreated message

```java
package network.messages;

import network.MessageReader;
import network.MessageWriter;
import network.Message;
import network.MessageFactory;
import network.StringSize;
import network.types.InventoryEntry;
import network.types.Avatar;
import java.util.ArrayList;
import java.util.List;
import java.nio.ByteBuffer;

public class ServerAvatarCreated implements Message
{
  public static class Factory implements MessageFactory
  {
    public Message createMessage(MessageReader reader)
    {
      return new ServerAvatarCreated(reader);
    }
  }

  private static int OPCODE = 24;

  private Avatar avatar;
  private List<InventoryEntry> inventory;

  private ServerAvatarCreated(MessageReader reader)
  {
    this.avatar = new Avatar(reader);
    this.inventory = new ArrayList<InventoryEntry>();
    short inventorySize = reader.readInt16();
    for (short i0 = 0; i0 < inventorySize; i0++)
```

```
      {
        inventory.add(new InventoryEntry(reader));
      }
    }

    public ServerAvatarCreated(Avatar avatar, List<InventoryEntry> inventory)
    {
      this.avatar = avatar;
      this.inventory = inventory;
    }

    public Avatar getAvatar()
    {
      return avatar;
    }

    public List<InventoryEntry> getInventory()
    {
      return inventory;
    }

    public ByteBuffer getData()
    {
      // OP-Code: 1
      // inventory: 2 (list size)
      short size = (short)3;
      size += avatar.size();
      for (InventoryEntry inventoryElem : inventory)
      {
        size += inventoryElem.size();
      }

      MessageWriter writer = new MessageWriter(size);

      writer.writeOpCode(OPCODE);

      avatar.write(writer);
      writer.writeInt16((short)inventory.size());
      for (InventoryEntry inventoryElem : inventory)
      {
        inventoryElem.write(writer);
      }
      return (ByteBuffer)writer.getData().flip();
    }

  }
```

B UML Diagrams

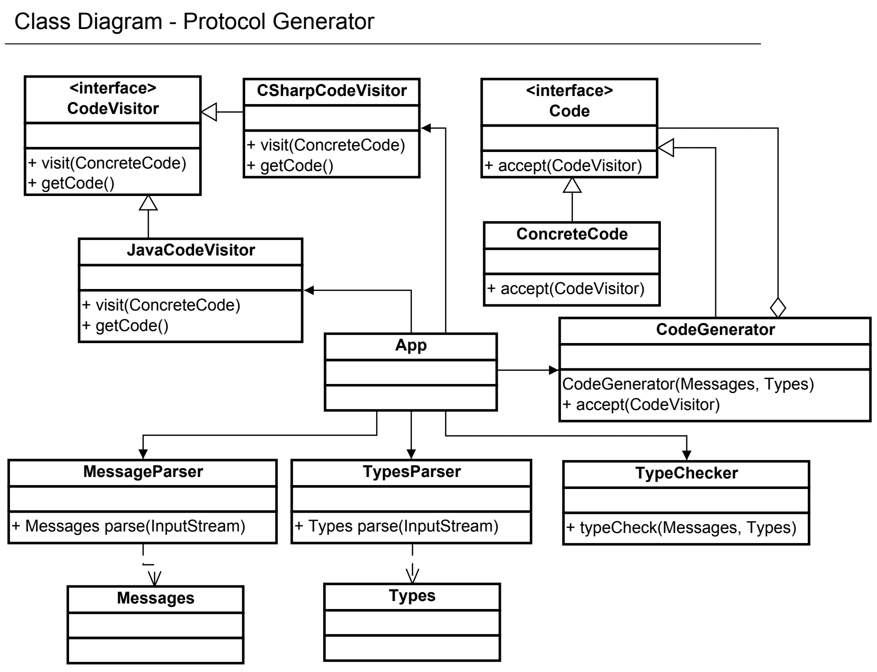


Figure 9: The application hands an InputStream to the MessageParser and TypesParser and gets a Messages and Types object in return. These are first sent to the TypeChecker, which checks the protocol for errors and then passes the Messages and Types to different CodeGenerators. The CodeGenerators produces Abstract Syntax Trees which are finally visited by the CodeVisitors to accumulate the code for the different messages.

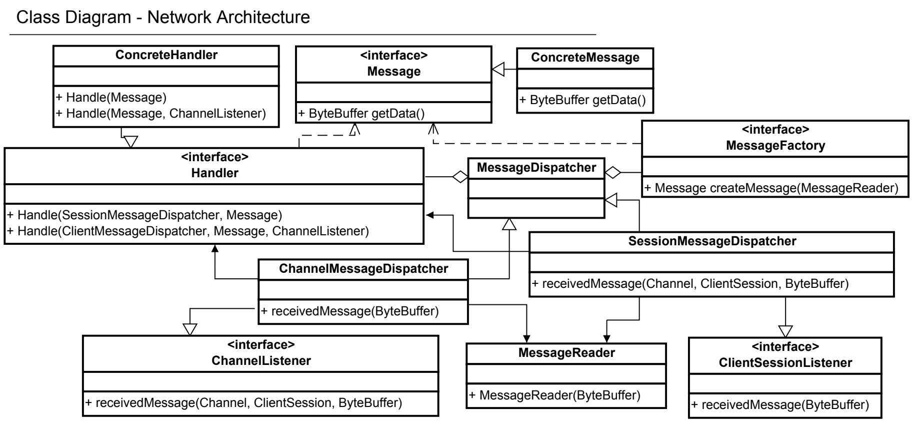

Class Diagram - Network Architecture
ConcreteHandler
+ Handle(Message)
+ Handle(Message, ChannelListener)
<interface>
Message
+ ByteBuffer getData()
ConcreteMessage
+ ByteBuffer getData()
<interface>
MessageFactory
+ Message createMessage(MessageReader)
<interface>
Handler
+ Handle(SessionMessageDispatcher, Message)
+ Handle(ClientMessageDispatcher, Message, ChannelListener)
MessageDispatcher
SessionMessageDispatcher
+ receivedMessage(Channel, ClientSession, ByteBuffer)
ChannelMessageDispatcher
+ receivedMessage(ByteBuffer)
<interface>
ChannelListener
+ receivedMessage(Channel, ClientSession, ByteBuffer)
MessageReader
+ MessageReader(ByteBuffer)
<interface>
ClientSessionListener
+ receivedMessage(ByteBuffer)

Figure 10: Messages are handed from the Darkstar API as a ByteBuffer passed to the messageReceived functions in the SessionMessageDispatcher and ChannelMessageDispatcher. These wrap the ByteBuffer in a MessageReader, read the unique identifier of the message, and create the corresponding message by passing the MessageReader to the MessageFactory. The message is then passed to its corresponding handler.

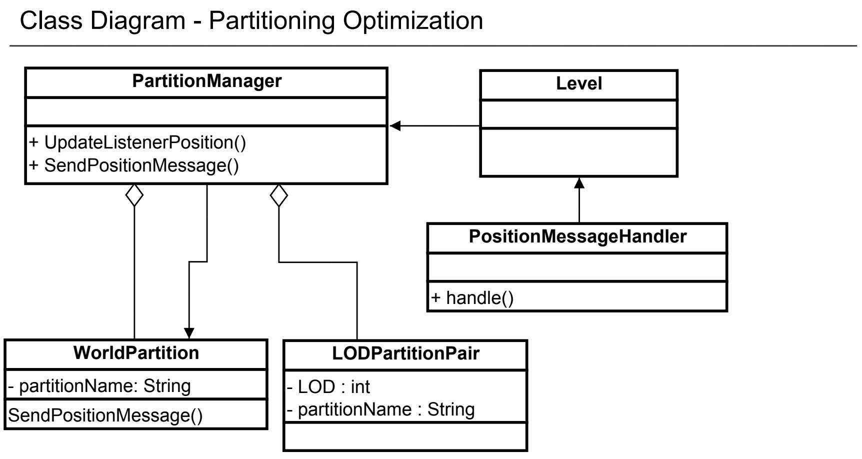


Figure 11: When a message is passed to the position update handler, the players current level is notified which in turn calls the PartitionManager's UpdateListenerPosition and SendPositionMessage functions. The PartitionManager updates which channel the player listens to and sends the message to adjecent partitions depending on the LOD value.