Massive Multiplayer Online First Person Shooter as a Peer–to–Peer game

Christian Grümme

17th May 2008
Massive multiplayer online games mostly have a client-server architecture and most peer-to-peer networks realize content distribution. As the server is always a bottleneck in a client-server architecture with a growing number of clients, it is worthwhile to analyze the replacement of this architecture by a peer-to-peer system. This thesis analyzes the requirements of such a system, presents a sample implementation and evaluates its suitability in regard to information distribution and scalability. Further topics like cheating protection and robustness to the drop-out of nodes will be discussed briefly.
I declare on oath, that this diploma thesis is entirely by myself, except where the aids and appliances that are already known by the adviser, that all used aids and appliances are stated and that everything has been marked, that has been excerpted, with or without modifications, from other papers.

Hiermit erkläre ich an Eides statt, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle verwendeten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderung entnommen wurde.
## Contents

1. Introduction ........................................ 7

2. Overview ........................................... 9
   2.1. Computer games .................................. 9
   2.1.1. History of computer games ................... 9
   2.1.2. Multiplayer games ............................. 10
   2.1.3. Massive multiplayer online games ........... 10
   2.2. Peer-to-Peer .................................... 10
   2.2.1. Peer-to-peer ................................ 10
   2.2.2. Client–server systems ....................... 11
   2.2.3. Hybrid systems ............................... 11
   2.2.4. Comparison ................................ 12
   2.2.5. An example ................................ 13

3. Objective and approach ................................ 14
   3.1. Objective .......................................... 14
   3.2. Approach .......................................... 14
   3.2.1. Server node requirements .................... 15
   3.2.2. Communication ................................ 15
   3.2.3. Zoning ........................................... 16
   3.2.4. Request ......................................... 17
   3.2.5. Routing ........................................ 21
   3.3. Synchronizing .................................... 22

4. Implementation ........................................ 23
   4.1. Implementation decisions ....................... 23
   4.2. The Sectorizer .................................... 24
   4.2.1. Ascending cube ................................ 24
   4.2.2. Sectors ......................................... 25
   4.2.3. Generating a sectorization ................... 25
   4.2.4. Sector parsing ................................ 25
   4.3. The middleware node ............................. 26
   4.3.1. Request queue ................................ 26
   4.3.2. Swarm table .................................. 26
   4.3.3. Sector table .................................. 28
1. Introduction

Today computer games have a big influence on people in modern societies just as other mass media. They have especially changed the daily life of young people. The importance and acceptance of computer games differ in the industrialized countries. In South Korea, for example, computer games have a high status. A video game culture is manifested in industrial nations, so it has a major economical influence. To play on the Internet with or against other people is also important. The attraction is to have game experiences in common with other people and to make a direct performance comparison. There are a growing variety of game modes with multiplayer abilities that have their own rules. So the weakness of computer opponents, such as a lack of artificial intelligence or schematic action, falls away. Massive Multiplayer Online Games (MMOG) are a special genre in this context. They just have multiplayer modes and operate with a huge number of players.

Multiplayer games are mainly client–server based, especially MMOGs. That has the big disadvantages, among others, that the working load of the servers becomes higher the more clients participate in a game. In contrast to that, Peer-to-peer architectures profits from a large number of participants.

In this thesis MMOGs and peer-to-peer systems are brought together by realizing a MMOG as a peer-to-peer system. This thesis analyzes the requirements of such a system, presents a sample implementation and evaluates its suitability in regard to information distribution and scalability. This is done in an abstract way, so that MMOG is handled as an application that can be exchanged. So it is possible to adopt the presented approach to other applications that have the same requirements as the given game, e.g. audio and video transmissions.

It was decided to use Quake 3 Arena as basis for the presented implementation because it is a very popular and successful first person shooter game and it has publicly accessible source code. First person shooters are a category of computer games, in which the character of the player accesses the representation of a freely accessible, three-dimensional game world through the eyes of a person. The game is mainly dominated by the struggle against a variety of different opponents with various firearms. So why take a first person shooter as a basis? They are fast-paced and have a complex game world, so the extremes of the requirement can be studied.

The approach given in this thesis is to divide the game world into smaller entities, called sectors. Then each node of the peer-to-peer network has to determine which sectors are relevant to his client. Each node exchanges information with those nodes that share relevant sectors with it. The presented implementation provides a distributed data structure to realize this.

The structure of this thesis is as follows: The next chapter gives a brief overview of the
two topics, computer games and the peer-to-peer technique. It explains them and associated terms and concepts. In the following chapter the principal topic is presented. The requirements, which are needed to realize the idea, are analyzed and an approach to an implementation is described. The fourth chapter introduces the sample implementation to the idea, describes its design, functionality and major data structures and proves their correctness. The next chapter deals with the evaluation of the sample implementation of the fourth chapter. It describes the evaluation setting and the result of the evaluation. The last chapter reviews the results and points out further topics. The appendix consists of a definition of the developed protocol that is used by the sample implementation, and a quick start guide for the use of the sample implementation in order to reconstruct the evaluation setting.
2. Overview

This chapter gives an overview about the two main topics of this thesis that will be dealt with together. The first topic will be computer games followed by the peer-to-peer technique. Some terms like *massive multiplayer online games* and *client-server architecture* will be defined.

2.1. Computer games

2.1.1. History of computer games

The development of computer games started about 50 years ago from some technical experiments at universities and became one of the most popular leisure forms of the 21st century.

There were attempts made, even on the first computers, to implement games, such as checkers. The first computer game, that offered new opportunities beyond ordinary games, is often quoted as the 1958 game *Tennis for Two* [Dav83] developed by the US American William Higinbotham. Tic-tac-toe and other games were implemented on the PDP-8 (1965). The development was heavily dependent on the technical progress of computer technology. The development initially only was a minor matter on mainframe computers at universities, which were actually intended for other purposes. It was in the 1970s made possible by combining the now relatively inexpensive, simple logic chips with existing TV receivers to create simple electronic gaming machines for a mass public. Very successful, for example, was *Pong* of Nolan Bushnell. Companies such as Atari or Magnavox released the computer game in the form of video game consoles for home entertainment. So a rapidly growing mass market has developed.

With the introduction of home and personal computers (PCs) in the 1980s, there were initially developed two technically separate types of computer games: The video game (then also telegames) based on specific game consoles and the computer game for home computers and later more and more for the PC. In 1983, there was a slump in the video game market mainly caused by saturation of the market by inferior video games and the technical superiority of the home computer to game consoles. In Japan, where the home computer was not so successful as in North America and Europe, Nintendo started a new era of video games in the same year, that reached two years later also North America and Europe.

Since the mid-1990s, the two areas for games consoles and PC’s converged for marketing reasons. The same storage media (as the CD-ROM or DVD) and a compatible hardware allowed to develop the same games for various consoles as well as for PCs in parallel in order to produce cheaper and for a wider mass market.
Computer games have become a widespread and important form of entertainment. Many countries have developed their own computer games industry, sometimes these are more advanced than the indigenous film industry [Ass07].

2.1.2. Multiplayer games

Many computer games also support the so-called multi-player mode, in which several human players play against or with each other. The games are played either on the same computer (while playing often with the help of split-screen technology), via networked devices, often via the internet, or on a so-called LAN party, where many like-minded person link their computers with each other. The sporting competition of computer games is called e-sports. Examples of games that are played in such competition are: Quake 3 Arena, Unreal Tournament, Warcraft 3 and Counter-Strike. The multi-player mode allows a direct comparison of playing skills.

2.1.3. Massive multiplayer online games

Via the Internet, it is possible for a high number of participants to play a computer game simultaneously. The game is running on a server and each user can connect from a networked computer to the game. The most important form of these online games are the massive multiplayer online role-playing games, short MMORPGs, where up to several thousand people are involved in a role-playing. The companies that produces such games often charge a fee to use the servers in addition to the purchase price of the game. This continuing revenue stream is an important source for the continuous improvements made to each game and the covering of server expenses and enhancement. The most successful MMORPG is World of Warcraft, which exceeded in January 2008 the world’s 10-million-player limit [Ent08].

2.2. Peer-to-Peer

The peer-to-peer architecture and the older client-server architecture are two different models for a distributed system. A distributed system is characterized by several independent processes which do not share memory so they have to communicate through messages. Usually a distributed system is connected over a computer network. The rules of communication, how the particular entities communicate, is defined by a protocol.

2.2.1. Peer-to-peer

A peer-to-peer system (P2P) is a computer network in which each computer is equal. Each computer can provide or use services. There is no role such as server or client. It is usually realized as an overlay network over an existing network.

P2P networks are currently used for:

- File sharing, e.g. BitTorrent protocol.
- VoIP, e.g. SIP protocol.
- Streaming media.
- Instant messaging.
- Software publication and distribution.
- Media publication and distribution.

There are different types of P2P networks:

**Centralized P2P network:** need a central maintenance server to work, e.g. Napster.

**Pure P2P network:** do not have a central entity, each peer is equal, e.g. Gnutella 0.4, Freenet.

**Hybrid P2P network** determine dynamically several central server for maintenance, e.g. Gnutella 0.6, JXTA.

![Peer-to-peer system](image)

2.2.2. **Client–server systems**

A client–server system is the opposite of a peer-to-peer system. The server provides a service and the client use the service. The server is on standby to react on a clients request, but stays passive while the client requests a service actively. It is the dominating principle in computer networks as the standard protocol of the internet shows, e.g. HTTP, DNS, FTP and SMPT.

2.2.3. **Hybrid systems**

Hybrid systems are a combination of both models. There are some nodes in the network that have special functions. These nodes are either some servers or normal nodes that have been granted with those functions (super-peer or super-node).

These special functions can be:
Figure 2.2.: Client–server system

- To keep data on peers and to respond on requests for that data.
- To maintain data what the peers have for available resources. The peers have to let the server know what resources they want to share.
- To resolve references of a node into an absolute address.

2.2.4. Comparison

In the following the advantages of each architecture in comparison to the other is pointed out.

Advantages of the client–server architecture:

- Consistence: To update data in a peer-to-peer system every node with this data has to be updated, while in a client–server system just the data of the server has to be updated.
- Security: All relevant data is stored on the server. Servers can better control access and resources, to guarantee that only those clients with the appropriate permissions may access and change data.
- Mature: There are many client–server technologies available and highly tested, so they can ensure security, a good interface, etc.

Advantages of the peer-to-peer system:

- Bandwidth: The server fulfills the client’s request. Its bandwidth have to split up between the clients and can be overloaded, while in a peer-to-peer system the bandwidth increases with more nodes, since the peer-to-peer system’s overall bandwidth can be thought as the sum of the bandwidths of every node in that network.
- Robustness: When the server fails, client’s requests cannot be fulfilled. In a good peer-to-peer system resources are usually distributed among many nodes. So if some nodes are failing, the remaining nodes should have the needed data.
2.2.5. An example

Bittorrent

Bittorrent is a peer-to-peer protocol which was designed to distribute large files over the internet [Coh01]. It communicates over TCP. The first version of the bittorrent protocol uses trackers as servers, so that this protocol is not a pure peer-to-peer protocol.

To download a file or a set of files a client downloads a torrent-file. This torrent-file provides the address of the tracker, the name and size of the files to download, and a hash code of each chunk (part of the requested data). The client connects to the tracker which sends a list of current other clients back. The clients connect to several of these and they tell each other through a bit field which chunks they have got. Then the client downloads random chunks, that it is lacking, from one of the connected clients that has the required chunks. A client stops sending chunks to a second client when the second client stops sending chunks to everyone.

Distributed hash table

As the bittorrent protocol is not a pure peer-to-peer protocol, the developers created an extension to remove the weakness of a central tracker [Coh01]. There is an extension to solve that problem, the "distributed sloppy hash table" (DHT), which is UDP- and Kademlia-based. The Kademlia is one implementation for a DTH and it is used for the bittorrent extension.

What is a distributed hash table? It uses a hash function to assign a key to data objects over a linear range of values. These values are distributed as evenly as possible over the nodes. Each node is at least responsible to some of the key areas, often there are several nodes responsible for the same area, whereby the responsibilities change dynamically. A protocol regulates the addition of new nodes in an existing system, usually by another node, which is already part of the system. The protocol provides the connections to the adjacent nodes and it also regulates the construction of routing tables. DHT routing tables are used to find out which other nodes are responsible for a given set of data. The definition of distance depends on the chosen topology and varies in different systems. It has mostly nothing to do with the physical organization of the node. The Euclidean space is an example, this can be simply the node with the lowest Euclidean distance to the key chosen to hold its value. Routing tables should allow each node to reach the node with a certain key in $O(\log n)$ steps.

A data object can be stored directly, that means a node with its key contained within it, or indirectly, by a reference to the address of the node, which contains this data object.

Since the key has no semantic significance, the managed data objects are application-independent. Though a generic interface, which only contains the two functions `publish(key, value)` and `lookup(key)`, it is possible to exchange the algorithms that implement it.
3. Objective and approach

3.1. Objective

As seen in the Overview MMOG’s mostly have a client-server architecture and most peer-to-peer networks realize content distribution. As the server is always the bottleneck in a client-server architecture with a growing number of clients, it is worthwhile to analyze the replacement of this architecture by a peer-to-peer system. This thesis analyzes the requirements of such a system, presents a sample implementation and evaluates the possibility of the suitability with regard to information distribution and scalability. Further topics like cheating protection and robustness to the drop-out of nodes will be discussed in brief.

3.2. Approach

A classical client-server architecture consists typically of one server and multiple clients (see graphic 2.2 on page 12), while a classical peer-to-peer system consists of just peers. In order to translate the architecture a peer has to fulfill the role of a client as well as some tasks of the server. To simplify matters, a peer is divided into a client (the original client) and a part that takes over the part of the server, the server node.

![Pure P2P-System: Each client has its own server node](image)

Figure 3.1.: Pure P2P-System: Each client has its own server node

From the viewpoint of the client there is just one server. Since it is the original client, it does not need any adjustments. It is connected to its server node as if it is connected
to a normal server. The server itself has to be distributed and to become a peer-to-peer system. The original server just consists of server nodes (the swarm) which are assumed to be running on different computer systems. It should be possible to run a single server node as the complete server system with no difference from the original non peer-to-peer server. Clients are able to log on to a specific server node or to let the server system decide which server node they should be logged on to. So it can be also possible that a server node is responsible for several clients (the server itself remains a pure peer-to-peer system). For the entire game cycle each server node is responsible for the clients that have logged on to it. A further study area could be migration of clients to other server nodes. In the following context it will be assumed that a server node just has one client. All propositions will be similar as if they were made for more clients per server node.

3.2.1. Server node requirements

Each server node has to provide its client with certain data. This data can be categorized by its urgency and need for reliability. Some data has to be transmitted in real time so that the user does not notice a delay [LBA04] such as changes in the visual environment of a client, e.g. an appearance of another client and other data that is allowed to have a delay, e.g. three seconds such as chat messages or end of game. For the server node the data differ also by their destinations. Some data, such as the end of game, are for all clients, that means for all server nodes. Other data are just for a group of clients, which refer to a group of server nodes, for example a chat massage to a specific client or a group of clients which see each other in the game.

3.2.2. Communication

In the case that a server node wants to send a message to a specific client which is not logged on to this server node, it is useful to have a list of all clients which participate in
this game and to which server node they belong to. So the server node has just to send a message for the client to its server node. When a server node has to send data to all server nodes/clients, it sends a broadcast to all server nodes. The broadcast may implemented as sending the message to each server node directly or per redirection, depending on the urgency of the message.

The main task for a server node is to provide its clients the information about what their perspective and if they are shot another client, in short the client’s status. What is obvious is the playground and its characteristics. The map and its firm characteristics (e.g. indestructible bridge and its textures) are known by each server node or even each client at the beginning. For the alterable characteristics (e.g. weapon respawn or movable objects) the server node has to determine what state they are in. The server node has to determine what other users are within the view of its client and if so to request their state (e.g. appearance and exact location).

When a group of users sees each other their server nodes have to communicate with each other and form a cluster. So how does a server node know with which other server nodes it has to communicate to and how? The answer is given in the next subsection.

### 3.2.3. Zoning

For a simpler determination of other visible clients it is useful to section the entire game map in three-dimensional sectors. Either this partitioning is given as an extra file corresponding to its map or a server node creates one on the fly and distributes it to the other server nodes. A precreated file may be useful to consider hotspots of the map [TLL05].

With this tool a server node has just to determine which other clients are in the sectors of its client and additionally which clients are in the sectors around the sectors of its clients. This is the major economization in the information management in comparing with the original server because the original server has to manage all information about the game world including all other clients.

In three dimensions each sector has 26 touching sectors except those which are on the border of the map. Is it necessary to determine which clients are in the surrounding sectors? The answer depends on the location of the client and on the size of the sector. How big should a sector be? For this question the following two facts are relevant. First, a client is usually not in the middle of a sector, so it might see into the surrounding sectors. Second, the client is expected to move, so in order to minimize latency it is useful to already know who is in the new sector that the client is entering. To simplify, we define that one sector is as big as the sight radius of one client.

The data flow have to be bidirectional. When a client sees a second client, it is presumably that the second client can also see the first client or just have to turn around in order to see the first client. Even when someone sees into a sector far away from its location, e.g. because it has a sniper rifle, it has to be bidirectional to tell if it hit someone.
3.2.4. Request

In the following a server node controls a sector, when a sector manages information concerning this area. That may be the case when a client of that server node is in this sector, or a client was the last one that has left that sector. So this server node must have all relevant information about this sector. So when two server nodes have each a client in the same sector they both control it and they have to synchronize their information. When their clients left the sector at the same time, one server node remain in control of this sector. Each server node which controls a sector has to be connected to the other server nodes that are controlling this sector as well and also to those server nodes which are controlling the surrounding sectors. So they exchange the behavior of their clients in those sectors. The set of nodes that are controlling a sector $x$ is called the cluster of sector $x$.

**Postulation 1.** A server node has interest in a sector $s$, if

- One of its clients has its location in sector $s$.
- One of its clients has its location in a sector $t$, and $s$ and $t$ are sharing a border which can be directly crossed by the client.
Postulation 2. The sectors:

- Each sector has to be controlled by at least one server node.
- Each server node has to control at least its sectors of interest.
- A sector $s$, which no server node has interest in, is controlled just by the last server node that had interest in $s$.

When a client is entering a new sector the sectors of interest changes for its server node $sn$. For each sector $y$ $sn$ loses interest in, the server nodes, that are controlling $y$, have to be informed that $sn$ is no longer in control of $y$, except it was the last one in it. The server node $sn$ has to memorize with a time stamp at least one server node that stayed in control of this sector. If $sn$ is connected to an other server node with which it does not share a sector of interest, it can cancel this connection. $sn$ leaves the cluster of this sector. Then for each new sector $x$ $sn$ gains interest in, $sn$ has to determine which server nodes are controlling $x$ at the moment. If $s$ knows at least one server node $sn2$, that is in control of $x$, $sn$ can connect to $sn2$ and get a list of all other sector nodes controlling

---

*E.g. an open 3D-space and a regular cubic partitioning of the map, each sector has 26 neighbors, except those at the edge of the map.*
Figure 3.5.: New sectors of interest while moving to another sector

$x$ in order to connect to them as well. If it was successful, $sn$ belongs to the cluster of sector $x$. When $sn2$ has no interest in $x$ but was holding it because its client was the last one leaving it, $sn2$ releases the control of the sector $x$. $sn$ tries to determine such a server node $sn2$ in two different ways.

**Postulation 3.** The server nodes:

- A server node $sn$ has the following data-objects:

  1. List of all server nodes, containing their ID (SNID) and URI.

  2. A sector table, which has a list of all sectors, with each containing a tuple of sector number, time stamp (‘now’ if $sn$ controls this sector) and a list with all SNID of at least one server node which was controlling this sector at the time stamp (or when in control all other sector nodes which are currently controlling this sector as well).

- One server node starts the game and initially controls all sectors.

- A new server node can join or leave the game at any time.

- When no server node is present, the game is ended.
- Clients can join or leave a specific server node at any time.

What are those two different ways to determine a server node that is controlling a certain sector? Each server node has a list of all sectors and a server node that controls it. Either this information is up-to-date because the server node controls it at the moment or it is outdated and it has a time stamp when this information was last seen up-to-date by this server node. The first way is to ask the server node, that is last known to be in control of that sector. Either it is still controlling this sector or it responds with a list with at least one other server node that controlled the sector but with a newer time stamp. Then the requesting server node asks the server node from the list, who is controlling this sector, etc. The requesting server node can take a second way simultaneously. It asks the server node to which it is already connected and which is currently controlling a sector that got the shortest in the Euclidean metric distance to the requested sector. The server node, that has been asked, responds either with a node, that controls the requested sector, or with an outdated list of a server node, it has last seen in this sector, and a server node, that has currently the shortest distance to the requested sector from its point of view. With this list the requesting server node can make another request.

**Algorithm 1. Request chain 1**

1. Ask last seen SNID controlling s
2. IF SNID is not controlling s THEN
3. overwrite entry of s in the sector table with the sector table entry of SNID
4. GOTO 1

**Algorithm 2. Request chain 2**

1. Let NODE be the server node in the current routing table with the (euclidean) nearest client to s
2. Ask NODE if it controls $s$
3. IF NODE is not controlling $s$ THEN
4. set NODE to the server node, that has the (euclidean) nearest client to $s$ in the current routing table of NODE

While the first way guarantees a request-chain to the right answer, see below, the second way tries to speed up the request. In the most common scenario, the server node is already connected to the server node that is controlling the sector. When its client is entering, the second way needs just one request to succeed because those server nodes already know all server nodes that are in control of the surrounding sectors.

If both algorithms are stopping because $sn$ waited to long for an answer, $sn$ assumes that some server nodes are lost. In this case $sn$ sends a broadcast to all other server nodes to request who is controlling $s$. If no one is currently controlling $s$, $sn$ gains control of $s$ and sends another broadcast with the new owner of $s$, so the other server nodes can update their sector table.
If a server node gets lost, there is no problem because its clients are counted as lost too and potential lost map information will be reset. In the case of portable items of high interest e.g. a flag in Capture the Flag - game type, a regular low priority broadcast is send out with its location by the server node that is in charge of the relevant sector.

When a server node joins a game, it copies the list with the sectors and which server nodes control them from another server node.

**Proof**

Let \( t_1, t_2, \cdots, t_n \) be the sequence of points in time where the control of sector \( s \) changes.

A server node \( a \) knows at leased server node \( b \) which is in control of sector \( s \) at time \( t_1 \). When \( a \) wants to know who is in control of sector \( s \) it asked \( b \). Either \( b \) is in control of sector \( s \) or lost control of sector \( s \) at \( t_k \) with \( k \in \{2, \cdots, n\} \). According to the procedure above every server node that lost control of sector \( s \) at a time point \( t_i \) with \( i \in \{1, \cdots, n\} \) it knows a server node that is still in control of sector \( s \) after \( t_i \). So in the second case \( b \) knows server node \( c \) who is in control of sector \( s \) after \( t_k \). Since every sector is controlled by at least one server node which request-chain leads to a server node that is in control of sector \( s \) after \( t_n \).

3.2.5. Routing

All server nodes that are in control of that sector and that also have a client in this sector have to send an update to each server node in the cluster. This communication were most efficient with a hardware based multicast communication network. In this scenario a server node just has to send the status of its client in one packet to all the other server nodes to an associated multicast address and listens for packets of the other server nodes for the status of their clients on the same multicast address.

There are networks that support hardware based multicast, like M6BONE [M6B01], but most of the routers of the regular internet do not support hardware based multicast. The most simple way is that every server node has a connection to every other server node with which it has to change data. This may be huge stress for a single server node when a big number of other server nodes has a client in the same sector. But every change in a sector has to be communicated to every other node controlling this sector to guarantee a consistent game state in this sector. To solve this problem we consider a model where the hole swarm is a graph. The server nodes represent the nodes and the connections between the server nodes represent the edges of the graph. So a cluster forms a complete subgraph. The degree of a node in this subgraph is in consequence the connection load of the server node. In order to reduce the connection load of a server node the subgraph can be converted into a bipartite graph in separating the nodes in two groups (e.g. by even and odd SNID’s). Each group forms an isolated set of nodes, so that every node in the one group is connected to the other group and not connected to its group. Every server node forwards data it gets from one node to the other connected server nodes. A mechanism is needed to prevent infinitely circulating data. A hop counter or sequence number with the corresponding SNID can satisfy this requirement. This
reduces the connection load by half. This idea can be generalized as followed:

Each node $sn$ in a cluster $c$, which contains $n$ server nodes, will be assigned to a set $i$ with $i \in \{0, \ldots, k - 1\} \subset \mathbb{N}$. Each massage $sn$ receives, that $sn$ has not sent by itself, will be forwarded to each node of the set $(i + 1) \mod k$. So the connection load has been reduced from $n$ to $\sqrt{n^k}$.

This method predominantly discusses the connection load but the traffic load is not considered. The traffic can be reduced when an individual data packet can be size-reduced aggregated with other data packet concerning the same sector. For example, when each server node just sends a packet with the current client's state, this data cannot be good size-reduced aggregated. When a server node sends an update of a sector status where the clients are represented by a certain bit pattern, a server node can aggregate all incoming data to one data packet. In this case the traffic $x$ will also be reduced to $\sqrt{x^k}$.

The disadvantage of this method is that each hop increases the ping time of a data packet, in the worst case $k \ast \max\{\text{Ping time between every node in cluster}\}$. So here a compromise between the connection load and the ping time has to be made. Another problem is the consistent partitioning of the nodes to the groups. The SNID can be chosen randomly and by a big number of server nodes the SNID will be probably form a consistent partitioning, e.g. a node is in group SNID $\mod k$. Since a cluster is highly dynamic it is difficult to guarantee a consistent partitioning within a cluster. To break the peer-to-peer principle in this case would solve the problem. A special node or external computer could handle joining and leaving clusters and put them into a group by entering a cluster.

3.3. Synchronizing

Server nodes that share a sector of interest have to synchronize their client’s action between each other. A possible way is to synchronize time among all server nodes by the Network Time Protocol (NTP) [Mil92]. Because the maximal time difference created by the NTP can still be 20 milliseconds, inconsistencies have to be resolved. A possible solution is when the server node, with the lowest SNID involved, overwrites the other server nodes. But it would be more convenient to postulate a game that uses a logical clock [Lam78] [Mat89] to avoid synchronization problems or problems with clock drift. A logical time depends not on the system time but gives monotone incrementing values.
4. Implementation

4.1. Implementation decisions

On August 19, 2005 id Software released the source code for Quake III Arena under the GNU General Public License. The source code is written in C and some other features like the map editor are written in C++. But the game is not completely free, due the textures and other data were not yet released. There are several projects that are based on the release of the Quake III – source code such as OpenArena or World of Padman which are completely free but not compatible any more with the original game.

There is also a project that perseveres compatibly with the original Quake III Arena along with debugging and additional features, ioquake. The original source code has no user friendly maintenance so ioquake was taken as a basis for the implementation.

![Diagram](image)

**Figure 4.1.: Implementation details**

The implementation itself is divided into three parts. The Sectorizer, which generates from a quake map file a sector description file, the node, which is a Java-based middleware that holds the major data structures and realizes the communication with other server nodes, and a communication module for the ioquake server. So the Sectorizer and the communication module are integrated in the original ioquake dedicated server. The original ioquake client does not need any modifications. This separation into modified
A dedicated server and a middleware follows the separation of game logic and server node. So this middleware can be a basis to implement this peer-to-peer approach in other applications.

4.2. The Sectorizier

The main challenge is to determine what happens around a client. Here is assumed that each client has a restricted sight of view, the sight radius. It is also assumed that each client has the same sight radius. If that is not true then take the sight radius as an equivalent to the maximum sight radius of all clients. So the hole map is discretized in cubes with a minimum diameter of the sight radius. The Sectorizier does just a simple nearly uniform sectorization, because so it is much faster to determine in what sector a point is and it uses less memory.

4.2.1. Ascending cube

An ascending cube as a cube with eight points is defined by:

- Point c1: Point with minimum over all points in the cube.
- Point c2: Next point to c1 in the third dimension.
- Point c3: Next point to c1 in the first dimension.
- Point c4: Next point to c3 in the third dimension.
- Point c5: Next point to c1 in the second dimension.
- Point c6: Next point to c5 in the third dimension.

![Figure 4.2: An ascending cube](image-url)
• Point c7: Point in the plane of c1,c3 and c5, so c1,c3, c5 and c7 is a square.

• Point c8: Next point to c7 in the third dimension.

So the sectorization just needs a sight radius $x$. Starting with the minimum point of the map the sectorization defines every $x$ steps a new sector. If a cube does not fit at the end of the map the previous sector is extended to the end of the map. The numbering of the sectors is first done in direction $x_1$, then in direction $x_2$ and at last in direction $x_3$. Due the most maps does not have different levels in $x_3$, the default setting just sectorize in 2D by giving the cubes in $x_3$ the maximum size.

4.2.2. Sectors

For the sectorization, described above (see subsection 4.2.1 on page 24), two standalone programs were programmed in ANSI C, so they can be easily integrated into ioquake and run under windows as well as on Linux/Unix. The Sectorizer generates a XML file to a given map file and a config file and the Sector Parser makes this file useful for sector calculation.

4.2.3. Generating a sectorization

To provide a map based sectorization the game map has to be analyzed. Quake III uses id’s BSP (Binary space partitioning) map format. Given a point and a sight angle, a 3D render traverse throw the BSP tree and so can rapidly calculate, what elements of the map are visible. But how is this BSP Tree stored in the BSP file? Keoaa Proudfoot provides unofficial Quake III map specifications [Pro00] which are very helpful in understanding the BSP map format.

So the Sectorizer needs to read the header of the BSP file to find out where the node lump starts and next to read out the coordinates of the first node which represents the whole map. With this information the Sectorizer generates an XML file with the sectorization in ascending cube style according to parameters given in a config file. For a more detailed sectorization it is necessary to traverse through the BSP Tree.

4.2.4. Sector parsing

To parse the XML file, given by the Sectorizer, the Sector Parser uses the GPL libxml2 library [Veil] which provides this function. The values are parsed from the XML file into a global structure, which stores extremal coordinates of the map, sector size (in each direction) and number of sectors (in each direction). The function, that calculates the sectors to a given point, is using this global structure.

Due to the libxml2 library is needing to be used, this library has to be installed. In order to compile ioquake with the Sector Parser its Make file has to be adjusted.
4.3. The middleware node

The software node is a middleware that realizes the communication within the peer-to-peer net. It was implemented in Java to provide portability and easy extensibility. Also the simplicity of using the Java library played a major role because node uses it intensively, for example it uses threads, an XML-parser, sockets and concurrent implementations of sets. All these things were inconvenient to implement in C. node implements also the SNP2P protocol that is specified later (see chapter A on page 44).

The entry point of nodes is its console. By typing in help a list of all available commands is printed in the console. Some commands are for debugging purpose such as connecttocluster to connect to a cluster. Other commands are to show the current status of the node or the swarm. Two commands are essential for the work of the nodes. The argument start initiates a swarm and connect connects to an initiated swarm.

4.3.1. Request queue

The Request queue is an abstract data type that has been developed especially for this implementation. It serves to handle the registration of a node in the swarm or in a cluster. When a node tries to enter, a Request queue will be set up in order to all memorize all other nodes that are entering until the original node picks this queue. It provides the following operations:

void insert(Object element, boolean isStartElement) : This method inserts a new element in the Request queue. If isStartElement is set to true, a queue can be got for this element with getQueue(elem). If there is no element with isStartElement set to true, no elements with isStartElement equals false will be entered.

Set<T> getQueue(Object elem) : Returns a set of all insert elements since a given value was inserted with isStartElement set to true. After returning of the set, the given element will be taken out of the Request queue as well as all elements from the beginning with isStartElement set to false until the first element with this isStartElement equals true appears.

4.3.2. Swarm table

The swarm table is the set of all registered nodes in the swarm. In this data structure other objects can get a set of all SNIDs or can resolve address data of a server node by its SNID.

A server node has to take care that its entry in the swarm table is correct at all members of the swarm. Therefore a server node registers at the swarm in the following way. A server node has to know an address of a server node, that is already a member of the swarm, it wants to register, and sends a register request to him. The entry server node checks the SNID of the requester by checking its own swarm table, a set of booked SNIDs. If there is no such SNID, it will be booked, a Request queue will be created for it and all other nodes will ask, if this SNID is available. If yes, each node will book this
SNID. This step is necessary because two nodes could register at the same time with the same SNID but at different entry server nodes. The entry nodes returns by success a list of all entries of the swarm table or by failure with a rejection. By a rejection the entering node generates a new SNID and starts all over again. By a success the entering node places the received list in its swarm table. Then each server node on the list will be informed of the entering. After that the entering server node fetches the Request queue from its entry node and informs all nodes of this Request queue of its entering. That ensures that every node knows that the entering server node belongs to the swarm, even when they have registered while the registration of this node. At last the new server node requests the sectorization file from the entry node and initiates its Sector table with a copy of the sector table of the entry node.

**Lemma 1.** Assumed that there is reliable communication and no server node fails. When a server node $x$ registers itself at a swarm $s$ in the way described above, $x$ knows every server node in $s$ and every server node in $s$ knows $x$.

*Proof.* Therefore that if the swarm size is zero, no server node can register, and if the swarm size is one, it is trivial, it is assumed that the swarm size is bigger than or equal to two.

**Case 1:** No other server node registers while $x$ is registering.

In this case $x$ registers at an initialized entry server node $y$ and gets the list of all other server nodes $l = \{s\} / y$ in the swarm and $x$ and $y$ know each other and $x$ knows $l$, so $x$ knows $s$. Then $x$ introduces itself to all server nodes in $l$. So every server node in $s$ knows $x$.

**Case 2:** A set of new server nodes $n$ registers while $x$ is registering.

$x$ registers at an initialized entry server node $y \in s$. Without loss of generality it’s assumed that $n = p, q$ and $p$ registers at $r \in s, r \neq y$ and has introduced itself to $y$ while $x$ has not yet gained its Request queue from $y$ and $q$ registers at $t \in s, t \neq y$ and has introduced itself to $y$ after $x$ has gained its Request queue from $y$.

- Due $p$ has introduced itself to $y$ before $x$ gains its Request queue, $p$ is in the Request queue when $x$ fetches it. So $x$ knows $p$ and introduces itself to $p$.
- After $x$ has fetched its Request queue, $x$ does not know $q$, but $x$ and $t$ know each other. Either $x$ has introduced itself to $t$ before $q$ has connected to $t$, then $x$ is in the first swarm list $q$ has received, or $x$ has introduced itself to $t$ after $q$ has connected to $t$, then $x$ is in the Request queue for $q$. $q$ has not yet fetched its Request queue at that moment because it has not introduced itself to $y$. So in both cases $q$ knows $x$ and introduces itself to $x$.

The leaving of a swarm is not as complicated as the entering. At first the leaving server node leaves all clusters and all clusters, that it still holds after that, will be hand over to another node and a broadcast is made to inform all nodes of the swarm of this
change. Then the leaving server node just sends a message, that it leaves the swarm, to all members. Each server node, that receives such a message, deletes the corresponding entry in its swarm table. While leaving all register requests to the swarm will be rejected with a reference to ask another server node.

4.3.3. Sector table

The Sector table is a set of cluster entries. Each cluster entry is the representation of its cluster of the corresponding sector. In a cluster entry is stored the state of the cluster, a time stamp and a list of SNIDs. The state can be active, managed, invalid or outdated. The state invalid indicates an uninitialized Sector table. The Sector table is initialized while the registration and all entries will be set outdated.

To enter a cluster, at first a server node has to find a server node that is already in the requested cluster. Therefore, it takes its outdated entry of this cluster in its Sector table and asks one of the nodes in the entry if it is still in the requested cluster. If no, the asked server node sends its entry of the requested cluster. If yes, the asked server node sets a lock so that it cannot leave the sector, then it behaves similarly to the entry of a server node to the swarm. The asked server node also sets a Request queue for this node and sends a list of all nodes that are currently in the corresponding cluster. The requesting server node introduces itself to the server nodes on the list, which inform the requesting server node if they are still in the cluster or if they have already left the cluster. After that the requesting node fetches its Request queue from the asked node. The asked node unlocks itself so that it is allowed to leave, and the requesting node introduces itself in the same way to the server nodes in the queue as it did to the others previously. If the asked server node does not have the state active in this cluster, but managed, it releases the sector control to the requesting server node and then it updates its entry to outdated with the SNID of the requesting server node and a time stamp.

Lemma 2. Assumed that there is reliable communication and no server node fails. When a server node $x$ in the swarm $s$ enters a cluster $c$ in the way described above (see subsection 4.3.3 on page 28), $x$ knows every server node in $c$ and every server node in $c$ knows $x$.

Proof. It is basically the same proof as the proof of lemma 1.

The leaving of a cluster is more complicated then the leaving of the swarm because one node has to remain in control of the sector. A leaving sector node has to find a server node in the cluster that is not currently leaving, that server node has to set a leave lock so that it cannot leave this cluster. Then the leaving server node informs all other server nodes in the cluster that it is leaving, they inform their dedicated server of the leaving. Then it tells that server node with the leave lock to unlock so that it can leave the sector. Through that locking it is assured that one node remains in the cluster. When it could not find any server node, that allows him to leave, the leaving server node has to stay. If there are no other server nodes in the cluster it just sets the state of the cluster from active to managed.
### Table 4.1.: States of a cluster entry

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
<th>Reaction by request</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid</td>
<td>Sector table entry not initialized</td>
<td>Nothing</td>
</tr>
<tr>
<td>Outdated</td>
<td>Has an outdated entry</td>
<td>Send entry list and last update time</td>
</tr>
<tr>
<td>Active</td>
<td>Has currently in the sector control</td>
<td>Locks leaving and sends entry list</td>
</tr>
<tr>
<td>Managed</td>
<td>Has left sector, holds as last one</td>
<td>Releases sector control and sets to outdated</td>
</tr>
</tbody>
</table>

**Lemma 3.** Assumed that there is a reliable communication and no server node fails. When a server node $s$ in a cluster $c$ of sector $s_1$ tries to leave $c$, at least one server node stays in control of $s_1$.

**Proof.**

**Case 1:** $s$ is the only sector node in $c$.
In this case $s$ sets its sector entry of $c$ to managed.

**Case 2:** $s$ finds a server node $t \in c$ that allows him to leave.
So $s$ leaves $c$ and $t$ stays in control of $s_1$.

**Case 3:** $s$ finds no server node in $c$ that allows him to leave.
In that case $s$ assumes all other server nodes in $c$ are leaving and $s$ remains. When all other nodes have left $c$ and $s$ still wants to leave, $s$ sets its sector entry of $c$ to managed.

\[ \square \]

### 4.3.4. Communication with the dedicated server

A dedicated server can connect via a TCP port, over which the sectorization file and the start command are also sent. The connected dedicated server sends the game states of its local client to the server node with the sector numbers it is in via UDP to its server node. Then the server node forwards to all relevant clusters it is already connected to. After that it tries to connect to every relevant clusters it is not connected to and disconnects from every cluster it is connected to but that is not relevant. Every server node has one threat that is responsible for each cluster, so that it cannot connect and disconnect at the same time.

If a game data packet via UDP from another server nodes arrives, it will be forwarded directly to the connected dedicated server.

### 4.3.5. Profiler

A profiler is added to determine the time a packet needs from one server node to a remote dedicated server. At a fixed time interval a special packet is send to all server nodes in
all connected clusters. This packet will be forwarded to their dedicated server. The dedicated server recognizes this special packet and sends a packet directly to the profiler of the server node that had sent the packet.

4.4. Game adjustment

*Quake 3* and therefore *ioquake 3* are divided into four parts: server, client, user interface and the binary executable [Hyp01]. The server controls and arbitrates the game. It decides what happens in the game world and has the last word so that the game world remains consistent. The client just sees the part of the game world that is relevant to it. Any data that is received by the client are given from the server. The client is also responsible for the game interface to the user, e.g. drawing the screen, playing sounds and processing the user input. It also has an understanding of the game physics so it can predict motion and make gameplay smoother over an Internet connection, but it does not settle the final position of any players. Both parts carry out what the game needs in order to play. The user interface handles the menu, pages of controls used to set up personal preferences and game parameters. The fourth part cannot be seen in the source code because it is the binary executable. This runs the virtual machines, required to play the game, manages the communication between the virtual machines on each computer, controls the network connection and handles the OpenGL drawing of the graphics. Any communication between virtual machines is going to have to pass through the binary executable.

In order to set *ioquake* up for its server node a function is added that checks and processes the additional command line argument `sn_set`. Followed by `connect addr`
<port> <port> it connects to a server node, first port is the TCP-port and the second the UDP-port of the server node, and waits for a start signal of the swarm, and followed by start, it sends a sectorization file and starts a swarm.

4.4.1. Quake 3 networking

The article *The Quake3 Networking Model* [Bri06] explains how the client-server communication works. Some communication packets have no sequence numbers like an info or connect packet. When a connection is successful, each packet, sent between the client and the server, has an incremental sequence number. The server sends a complete game state to the client and memorizes it. Then it sends continuously a delta message, that means all changes, between the memorized game state and the current game state until the client acknowledges a specific game state by a number and its actions. Then the server processes these actions and sends a delta message between the acknowledged game state and the current, and so on.

An id field has been added to the players state, in order to determine if a player is a local player (equals an '0') or is a remote player (id equals the id of the corresponding server node). When a data packet of the server node arrives, the id of the origin will be extracted. Then the corresponding player will be searched and updated. If there is no such player, it will be assumed that a new player is in visual range, so a new player will be added with the received id. The data packet also holds a time stamp so the updated player state will be signed with this time stamp. When a time stamp with '-1' arrives, it will be assumed that the player has left the sight of view, so the player will be removed from the local game.
5. Evaluation

To determine if the chosen way leads to the goal, the implementations had to be tested. Therefore, the implementation ran in several different settings to determine periodically the ping rate in all connected clusters of selected nodes. Furthermore, the data in- and output of a random sampling of nodes was logged. This data was split up into the data of the P2P protocol and the actual data from the game.

5.1. Test setting

Due the previously discussed reasons the choice of the game map has a great impact on the success of the approach (see chapter 3 on page 14), a special map was created that fits certain aspects. At first a sector size had to be chosen. The sector size depends on an appropriate line of sight. The final choice was made according to game elements.

A rocket which is the slowest missile in Quake 3 moves 900 distance units in a second. So to see a rocket about 5 seconds would be a good empirical value. So the defined value for a line of sight and with it the sector size is 4500 distance units. Unfortunately, the Quake 3 map specification limits the possibilities to design a game map. The boundaries are 64 spawn points, 256 map entities and $65535 \times 65535$ distance units for the size of a map.

With these limitations in mind a game map was created, named sector12_12, that contains the maximum amount of uniform sectors of 4500 distance units, so $12 \times 12$ sectors. This map also contains the maximum of spawn points and map entities evenly distributed over the game map. This is necessary to keep the artificial players occupied and in motion. Without entities the artificial players would do nothing and the large number of spawn points provides a diversified location of the artificial players. The Sectorizer was set to do a uniform sectorisation with 12 sectors each dimension of the 2D-plane and one sector in height. Three test runs were made with a small map that is part of Quake 3, named q3dm1 (see table 5.2 on page 33), in order to have a relation, how big the influence the choice of the map is.

The test runs took place at the students computer pool. Up to 10 computer that had all the same configurations (see table 5.1 on page 33) were used. The execution followed all the same pattern. A node that initiated the swarm was started, then the other nodes connected themselves to the first node. Therefore, all nodes were evenly allocated over the participating computers. After all nodes have established a swarm, for each node a quake server client with one artificial player was started. At every computer some randomly picked nodes started a profiler, that performed the pinging and its logging.
System: Dell
CPU: Intel Pentium 4
CPU Speed: 3,0 GHz
Main memory: 1 GB
Operating System: MS Windows Server 2003 Ent. with SP2
Network card: Intel Pro/1000 MT
Java version: JRE 6 Update 6

Table 5.1.: Test configuration

<table>
<thead>
<tr>
<th>Run No.</th>
<th>Number of nodes</th>
<th>Number of running computers</th>
<th>Number of profilers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>40</td>
<td>10</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 5.2.: Test run on the small map

5.1.1. Limitations

The limited number of computers available forced us to run several nodes and game servers on one computer. That causes various problems. First of all the network load is naturally far greater than it would be with one node and game server, but also the CPU load is affected by the number of network connections. While a swarm of 8 nodes running along with their game server (with the sector12_12 map) on one computer causes a CPU load of about 50 % (determined by MS Task Manager), a swarm of 8 nodes running on two computers causes a CPU load of about 60 % per computer.

Not only the CPU reduces the network performance but also the operating system. MS Windows 2003 Server limits since Service Pack 1 the number of TCP connections at 10 connections per seconds which is a problem when 4 or more computer participate at a swarm. In this case some TCP connections mainly to the initialize node are failing. What effect it has is dealt with later.

In contrast the artificial players are no limitation to the test setting because they are permanently active and are moving across the game map. Unlike real players, who may lie in waiting, the behavior of the artificial players is more favorable in this test setting because they are constantly moving and may change sectors more often.

5.2. Results

To prove the playability, the ping rate has to be under 300 ms, so the profiler has to time a ping rate under 600 ms (back and forth). The profilers of the test runs were visualized to understand if the goal is accomplished. Therefore, the minimal, average and maximal ping of each sample was calculated. Lost packets were not considered. Each sample recorded how many nodes and clusters the node of the profiler was connected to (see
<table>
<thead>
<tr>
<th>Run No.</th>
<th>Number of nodes</th>
<th>Number of running computers</th>
<th>Number of profilers</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>20</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>40</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>80</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>80</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>20</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>12</td>
<td>40</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>13</td>
<td>58</td>
<td>29</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.3.: Test run on the big map

![Profiler of test run no. 13](image)

Figure 5.1.: A profiler of test run 13

The result were ambivalent because on the one hand most samples had an average ping rate under 600 ms, but on the other hand the influence of the TCP limit had a great impact. Several nodes had continuing TCP connection errors. Some profilers could not pick samples because their node could not connect to a cluster. The best example for this case is test run 06. It was to notice, that with a growing number of nodes, the number of TCP connection errors were growing. The main reason was, that the node, which had initiated the swarm, was too heavily under load. This can be solved by granting some sectors, which have the state managed, to a registering node. This idea were sadly not implemented because the schedule of this theses did not allowed it.

The test runs also showed, that the choice of the test map had not such a big influence. On the small map the number of connected nodes are more fluctuating as in the big map, but their ping rates were comparable.
5.2.1. Scalability

At the end of each test run, the amount of bytes from some randomly picked nodes were recorded, in order to measure the scalability. The recorded amount was composed by the input and output of the protocol (connecting/disconnecting to a cluster or a cluster request) and by the input and output of the game data. This data is sorted by the amount of data packets that the picked node had received.

To determine, if the size of the map matters, the samples were split up. The visualizations show that on the small map the percentage of the sent application data is higher (see figure 5.3 on page 35). That can be referred to a smaller number of players. The percentage of the protocol traffic is nearly constant with a growing amount of packets and with a growing amount of players. It has to be analyzed if this fact is a consequence of the connection limit or of the design of the implementation. The computers, without this connection limit, are not available for the testing of the implementation, so that we could not analyze this consequence. With the presumption, that the connection limit had a small influence, the implementation has a suitable scalability and the results verify...
Figure 5.4.: The visualization of all samples

Figure 5.5.: The visualization of the samples of the small map

the approach.
Figure 5.6: The visualization of the samples of the small big

Net traffic distribution - Sector12, v3
Application in
Application out
Protocol in
Protocol out

Fig. 5.6: The visualization of the samples of the small big
6. Conclusion and future research

6.1. Conclusion

The objective was to realize a MMOG as a peer-to-peer game. The approach to fulfill this objective was to divide the game world into smaller sectors. The presented implementation realizes each peer as consisting of a dedicated server, a modification from the original one, an instance of the Java middleware nodes and the unmodified original client. The implementation provides a distributed data structure, the Sector Table together with the Request queue, that allows communicating with just those other peers that share relevant sectors.

The criteria to the objective is that the implementation is playable, that means the average ping rate of the evaluation setting is under 300 ms. The evaluation of the implementation shows that the posed objective has been accomplished, but in consequence of the TCP limit, it could not clearly be verified.

6.2. Future research

This thesis introduces new fields that seems to be interesting.

The main idea to sectorize a given map is here just implemented in a generic way. So to study sectorizations aside from cuboid that consider the geometry of the map and hotspots, would be a field that falls into the ambit of computational geometry.

Cheating is a serious matter in the game industry especially in online games. They want to ensure fairness to all players. Due in this work each user runs an own open source dedicated server, it can modify the source to its benefits, e.g. to add extra items to its player or to automatically adjust its shooting direction to the location of an enemy (aimbot). A method has to be developed that detects and blocks such modifications.

The presented implementation enables a pure peer-to-peer system from the clients perspective although the presented approach allows the implementation of a peer-to-peer system from the server perspective. That means that several clients are able to log on to the same server. So an implementation that realizes multi-client server, can be studied from the perspective of scalability in comparison to this implementation.

Client logs on to a specific server and stays there until it leaves the game. That is no problem with the assumption that dedicated server and client are running on the same machine but with considerations like multi-client server this assumption becomes invalid. So to provide an average utilization of all servers, clients have to be migrate dynamically to other servers. The migration to the nearest server, that means to the server with the lowest ping rate, is an extension to this topic.
The presented approach tries to fit all requirements, for the function of the game Quake 3 or first person shooters in general, to match to the setting in a peer-to-peer system. Another approach would be to figure out the requirements of the peer-to-peer system and to match them to the setting of a first person shooter. In other words, what game design fits to the presented approach? Some ideas have been given such as to implement a logical clock instead of using the system time and to aggregate player states or sector states.

The routing presented in section 3.2.5 (see page 21) is not implemented. So with the ability to aggregate (see above) the presented routing promises a good approach to decrease not only traffic but also the degree of a server node.

In order to verify the worked out results, the presented implementation should be tested again in a setting without a TCP connection limit.
Tools and aids used

For the implementation:

- Ubuntu 7.10
- Microsoft Windows XP
- Kate
- Gcc 4.3
- Doxygen
- Java SE 6
- Eclipse 3.3 with CDT

For writing this document:

- Microsoft Windows XP
- Ubuntu 7.10
- MiKTeX 2.7
- TeXnicCenter 1 Beta 7.01
- Kate
- Microsoft Paint 5.1
- Inkscape 0.46
- Gnuplot 4.3
- Open Office 2.3
- dict.cc
- wikipedia.org
- wikimedia.org
Bibliography


Appendix
A. The P2PSN – Protocol

This section describes the protocol with which a specific node communicates to another node. This protocol is inspired by the HTTP-Protocol [BLFF96] to ensure simple read-ability to a user and to a machine. All messages expect some of the application section are usually transfered by TCP.

As HTTP it is line orientated. Every Line is ended by ‘\r\n’. Optional lines are embraced with [ ] and expressions, that are allowed to occur zero or more times, are embraced with ( )*. A | signals that a line may have the value left or right of the | but not both.

The header of a message start with three digits, that is the request code or answer code and indicates the message type. It is followed by an optional human readable description. At last there is the protocols name and version divided by a ‘/’, in this case P2PSN/0.1.

There are different types of messages. Request (R) for a request to a single server node and the corresponding answer (A). There are also generic (G) messages that can be used additionally and simple messages (M) that are send to one or more server nodes and that expect no answer.

A.1. Misc

The range from 001 – 099 are generic and miscellaneous answer and request code.

<table>
<thead>
<tr>
<th>Co.</th>
<th>Message</th>
<th>Ty.</th>
<th>Description</th>
<th>Re.</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>001 Request OK P2PSN/0.1\r\n</td>
<td>G</td>
<td>A generic answer for a successful request.</td>
<td>–</td>
</tr>
<tr>
<td>002</td>
<td>002 Bad Request P2PSN/0.1\r\n [Description]</td>
<td>G</td>
<td>A generic answer for a invalid request. A description of the error can be optional appended.</td>
<td>–</td>
</tr>
<tr>
<td>003</td>
<td>003 Not Yet Initialized P2PSN/0.1\r\n</td>
<td>G</td>
<td>A generic answer for a invalid status of the queried Server Node.</td>
<td>–</td>
</tr>
<tr>
<td>010</td>
<td>010 Get Swarm Info P2PSN/0.1\r\n</td>
<td>R</td>
<td>Request for informations from the queried swarm.</td>
<td>011</td>
</tr>
<tr>
<td>011</td>
<td>011 Swarm Info P2PSN/0.1\r\n data</td>
<td>A</td>
<td>Answer from a request for informations from the swarm.</td>
<td>010</td>
</tr>
</tbody>
</table>
A.2. Enter/Leave the swarm

The range from 100 – 199 are answer and request code that proceed with the connection and disconnection of server nodes to and from the swarm.

<table>
<thead>
<tr>
<th>Co.</th>
<th>Message</th>
<th>Ty.</th>
<th>Description</th>
<th>Re.</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>101 Node Register P2PSN/0.1\r\n</td>
<td>R</td>
<td>Request to enter a swarm. The header is followed by a representation of the requesting server node. The first port is the TCP-port and the second one is the UDP-port.</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>(&lt;SNID&gt;@&lt;address&gt;:&lt;port&gt;/&lt;port&gt;)\r\n</td>
<td></td>
<td></td>
<td>105</td>
</tr>
<tr>
<td>102</td>
<td>102 Node Register Forward P2PSN/0.1\r\n</td>
<td>R</td>
<td>Request from an entry server node to the swarm if the sent representation is unique.</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td>(&lt;SNID&gt;@&lt;address&gt;:&lt;port&gt;/&lt;port&gt;)\r\n</td>
<td></td>
<td></td>
<td>104</td>
</tr>
<tr>
<td>103</td>
<td>103 Node Register OK P2PSN/0.1\r\n</td>
<td>A</td>
<td>A positive answer for a Register Forward request for the corresponding SNID.</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>(&lt;SNID&gt;)\r\n</td>
<td></td>
<td></td>
<td>102</td>
</tr>
<tr>
<td>104</td>
<td>104 Node Register Denied P2PSN/0.1\r\n</td>
<td>A</td>
<td>A negative answer for a Register Forward request for the corresponding SNID.</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>(&lt;SNID&gt;)\r\n</td>
<td></td>
<td></td>
<td>102</td>
</tr>
<tr>
<td>105</td>
<td>105 Swarm Full P2PSN/0.1\r\n</td>
<td>A</td>
<td>A negative answer for a Register request due maximum of server nodes is reached in this swarm.</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>(&lt;SNID&gt;)\r\n</td>
<td></td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>107</td>
<td>107 Node Register Ack P2PSN/0.1\r\n</td>
<td>A</td>
<td>An optional positive answer for a Register request.</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>(&lt;SNID&gt;)\r\n</td>
<td></td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>108</td>
<td>108 New Node Register Ack P2PSN/0.1\r\n</td>
<td>M</td>
<td>A registering server nodes sends this message to every server node in the swarm, except his entry server node, to inform of his connecting to the swarm.</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>(&lt;SNID&gt;)\r\n</td>
<td></td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>109</td>
<td>109 Node Disconnect P2PSN/0.1\r\n</td>
<td>R</td>
<td>Disconnect from the swarm request.</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td>(&lt;SNID&gt;)\r\n</td>
<td></td>
<td></td>
<td>101</td>
</tr>
<tr>
<td>110</td>
<td>110 Get Node List P2PSN/0.1\r\n</td>
<td>R</td>
<td>An optional request to get a list of all server nodes. This request is actually not used.</td>
<td>111</td>
</tr>
</tbody>
</table>

(proceeding on the next page)
<table>
<thead>
<tr>
<th>Co.</th>
<th>Message</th>
<th>Ty.</th>
<th>Description</th>
<th>Re.</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>111 Send Node List P2PSN/0.1</td>
<td>A</td>
<td>A positive answer for a Register request by sending a list of all server nodes of the swarm or a positive answer to the register request. The first server node representation belongs to the entry node and the following to the rest of the swarm.</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>110</td>
</tr>
<tr>
<td>112</td>
<td>112 Get New Nodes P2PSN/0.1</td>
<td>R</td>
<td>A request to receive his Request queue.</td>
<td>113</td>
</tr>
<tr>
<td>113</td>
<td>113 Send New Nodes P2PSN/0.1</td>
<td>A</td>
<td>Answer to the Request queue request.</td>
<td>112</td>
</tr>
<tr>
<td>120</td>
<td>120 Get Sectorisation P2PSN/0.1</td>
<td>R</td>
<td>Request to resolve the sectorization file.</td>
<td>121</td>
</tr>
<tr>
<td>121</td>
<td>121 Send Sectorisation P2PSN/0.1</td>
<td>A</td>
<td>Answer to the sectorization file request. The data section are file size bytes that is the sectorization file.</td>
<td>120</td>
</tr>
<tr>
<td>130</td>
<td>130 Get Sector Table P2PSN/0.1</td>
<td>R</td>
<td>A request to resolve the sector table of a server node.</td>
<td>131</td>
</tr>
<tr>
<td>131</td>
<td>131 Send Sector Table P2PSN/0.1</td>
<td>A</td>
<td>Answer to the sector table request. Each entry of the requested server node will be sent as a sector table entry state followed by the list of the SNIDs of the server nodes in the cluster.</td>
<td>130</td>
</tr>
</tbody>
</table>

**A.3. Enter/Leave a sector cluster**

The range from 200 – 299 are request and answer codes for leaving and entering a cluster.

<table>
<thead>
<tr>
<th>Co.</th>
<th>Message</th>
<th>Ty.</th>
<th>Description</th>
<th>Re.</th>
</tr>
</thead>
<tbody>
<tr>
<td>201</td>
<td>201 Get Cluster Control P2PSN/0.1</td>
<td>R</td>
<td>Request to enter a cluster.</td>
<td>202</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>203</td>
</tr>
<tr>
<td>202</td>
<td>202 Cluster Control Pass P2PSN/0.1</td>
<td>A</td>
<td>Answer to a enter cluster request when requested server node was managing this sector.</td>
<td>201</td>
</tr>
</tbody>
</table>

(proceeding on the next page)
<table>
<thead>
<tr>
<th>Co.</th>
<th>Message</th>
<th>Ty.</th>
<th>Description</th>
<th>Re.</th>
</tr>
</thead>
<tbody>
<tr>
<td>203</td>
<td>203 Sector Outdated P2PSN/0.1\r\n &lt;Sector no.&gt;\r\n &lt;time stamp&gt;\r\n &lt;amount&gt;\r\n (&lt;SNID&gt;\r\n)*</td>
<td>A</td>
<td>Answer to a enter cluster request when requested sever node got an outdated entry of this cluster. In the message is a list of SNIDs with a size of amount.</td>
<td>201</td>
</tr>
<tr>
<td>204</td>
<td>204 Cluster List P2PSN/0.1\r\n &lt;Sector no.&gt;\r\n &lt;time stamp&gt;\r\n &lt;amount&gt;\r\n (&lt;SNID&gt;\r\n)*</td>
<td>A</td>
<td>Answer to a enter cluster request when requested sever node is in control of this cluster. In the message is a list of SNIDs with a size of amount and the requested server node sets a Request queue for the requester up.</td>
<td>201</td>
</tr>
<tr>
<td>205</td>
<td>205 New Node in Cluster P2PSN/0.1\r\n &lt;SNID&gt;\r\n &lt;Sector no.&gt;\r\n</td>
<td>R</td>
<td>A message to inform a server node of the cluster entering.</td>
<td>206</td>
</tr>
<tr>
<td>206</td>
<td>206 Cluster Enter OK P2PSN/0.1\r\n</td>
<td>A</td>
<td>Answer to the information a new server node is entering, when informed server node is in control of this sector.</td>
<td>205</td>
</tr>
<tr>
<td>207</td>
<td>207 Left Cluster Already P2PSN/0.1\r\n</td>
<td>A</td>
<td>Answer to the information a new server node is entering, when informed server node has left this cluster.</td>
<td>205</td>
</tr>
<tr>
<td>208</td>
<td>208 Node in Cluster reg. P2PSN/0.1\r\n &lt;SNID&gt;\r\n &lt;Sector no.&gt;\r\n</td>
<td>R</td>
<td>This request gains the Request queue.</td>
<td>209</td>
</tr>
<tr>
<td>209</td>
<td>209 Cluster List Additional P2PSN/0.1\r\n &lt;Sector no.&gt;\r\n &lt;time stamp&gt;\r\n &lt;amount&gt;\r\n (&lt;SNID&gt;\r\n)*</td>
<td>A</td>
<td>Answer to the Request queue request. The Request is a list of amount SNIDs.</td>
<td>208</td>
</tr>
<tr>
<td>210</td>
<td>210 Ask for Leaving P2PSN/0.1\r\n &lt;SNID&gt;\r\n &lt;Sector no.&gt;\r\n</td>
<td>R</td>
<td>Request for leaving a cluster.</td>
<td>211</td>
</tr>
<tr>
<td>211</td>
<td>211 Leaving OK P2PSN/0.1\r\n</td>
<td>A</td>
<td>A positive answer to the leaving cluster request. The requested server node sets up a leave lock and informs the dedicated server of the leaving of the client.</td>
<td>210</td>
</tr>
</tbody>
</table>

(proceeding on the next page)
A.4. Data

The range from 300 – 399 are message codes that have to do with application – server node communication. This part of the protocol can be adjusted for other applications. The data and ping messages are usually transfered by UDP.

<table>
<thead>
<tr>
<th>Co.</th>
<th>Message</th>
<th>Ty.</th>
<th>Description</th>
<th>Re.</th>
</tr>
</thead>
<tbody>
<tr>
<td>301</td>
<td>301 Application Connect P2PSN/0.1\r\n &lt;address&gt;:&lt;TCP-port&gt;\r\n &lt;address&gt;:&lt;UDP-port&gt;\r\n</td>
<td>R</td>
<td>A message with it an application connects to a server node. It hands his connection information over.</td>
<td>001</td>
</tr>
<tr>
<td>302</td>
<td>302 Application Disconnect P2PSN/0.1\r\n</td>
<td>R</td>
<td>Message with that a application disconnects form it’s server node.</td>
<td>001</td>
</tr>
</tbody>
</table>

(proceeding on the next page)
<table>
<thead>
<tr>
<th>Co.</th>
<th>Message</th>
<th>Ty.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>303</td>
<td>303 Application Start P2PSN/0.1\r\n</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>R</td>
<td>Message through that the start of the game is signalized.</td>
</tr>
<tr>
<td>304</td>
<td>304 Application File P2PSN/0.1\r\n</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;SNID&gt;\r\n</td>
<td>R</td>
<td>Message type to realize a file transfer between a server node and his application, e.g. the sectorization file. The data section are file size bytes.</td>
</tr>
<tr>
<td></td>
<td>&lt;file length&gt;\r\n</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>310</td>
<td>310 Application Data P2PSN/0.1\r\n</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(&lt;sector no.&gt;;)*</td>
<td>&lt;SNID&gt;\r\n</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;time stamp&gt;\r\n</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;amount of bytes&gt;\r\n</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>data</td>
<td>M</td>
<td>Message to transfer the application data. For transfer from application to the server node the first field is used for the list of concerning sectors, while for server node to server node and server node to application the first field is used for a SNID.</td>
</tr>
<tr>
<td>320</td>
<td>320 Profiler Ping Request P2PSN/0.1\r\n</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;Address&gt;:&lt;profiler port&gt;\r\n</td>
<td>R</td>
<td>A profiler packet to clock the ping rate. The address and port is the return address of the profiler</td>
</tr>
<tr>
<td>321</td>
<td>321 Profiler Ping Answer P2PSN/0.1\r\n</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;Sequence number&gt;\r\n</td>
<td>A</td>
<td>A answer to the profiler packet with the received sequence number.</td>
</tr>
</tbody>
</table>
B. Quick start

B.1. How to use

B.1.1. The node middleware

Make sure you have Java SE 6 installed or a reference to a working binary. Then go with a console to the directory where the nodes middleware is (node.jar). To start an initiating node type in:

```
java -jar node.jar start
```

Then type in `status`, and something like this will appear:

Local Node:
Id: 83240702
Address: 160.45.112.120
Swarm Port: 28000
Data Port: 29000
Home Port (TCP): 30000
Home Port (UDP): 30000

...

Now start up as many nodes you want to have in your swarm. To start a single node, type in:

```
java -jar node.jar connect 160.45.112.120 28000
```

The IP-address is the IP-address of the initiating node above (or an already connected node), followed by the corresponding swarm port.

B.1.2. ioquake

First make sure you are running Microsoft Windows XP or above and you have the libxml2 installed (see: http://xmlsoft.org/). Extract the archive sn_ioquake.zip and enter its directory with a console. For an usage on a Linux operating system you have to compile it.

To start a dedicated server for the initiating node, type in:
ioq3ded.x86.exe +set sv_pure 0
+map sector12x12_3
+set sv_hostname "dedx"
+set sv_maxclients "64"
+addbot Sarge 2 +sn_set start
+sn_set connect 160.45.112.120 30000 30000

The keyword map is followed by the name of the map that should be played, with
the keyword addbot an artificial players is added, with the keyword sn_set start the
game is started and finally with the keyword sn_set connect the corresponding node is
specified by his IP-address and his TCP home port followed by his UDP home port.

To start a dedicated server for the other nodes, it is done in almost the same way:

ioq3ded.x86.exe +set sv_pure 0
+map sector12x12_3
+set sv_hostname "dedx"
+set sv_maxclients "64"
+addbot Sarge 2
+sn_set connect 160.45.112.121 30000 30000

B.1.3. Profiling

To start a profiler a server node, type in his console:

profiler start

To call a server nodes status, type in:

status

This writes also a status file in the server nodes directory.

B.2. How to compile

The Java middleware does not need to be complied because its byte code is in the jar
file together with its source code.

The dedicated server have to be compiled under the Linux operation system. Make
sure you have all tools and libraries installed that are used for ioquake (see: http://
ioquake3.org/source-codes/) and additionally the libxml2 library.

Then download the revision 1352 of ioquake via subversion:

svn co svn://svn.icculus.org/quake3/trunk ioquake3 -r1352

You can also use the archive ioquake-src.zip instead. Then copy the content sn_source.zip
into the source folder of ioquake. Some files will be overwritten. Then follow the instruc-
tion to compile ioquake. At last extract the archive add-on.zip into the build directory. It
contains among other things the pak0.pk3 file from the Quake 3 Arena Demo (see: http:
//www.idsoftware.com/games/quake/quake3-arena/index.php?game_section=demo) and
updated pk3 files (see: http://ioquake3.org/source-codes/). You have to agree to
the EULA (see: add-on.zip) to use those pk3s.
C. Visualisation of the test runs

C.1. Test run 01
C.2. Test run 02

Profiler of test run no. 02:

- max
- avg
- min
- Connected to
- Cluster
- Nodes

Profiler of test run no. 02:

- max
- avg
- min
- Connected to
- Cluster
- Nodes
C.3. Test run 03
C.4. Test run 04
C.5. Test run 05
Profiler of test run no. 05/

max
avg
min
Cluster
Nodes

Profiler of test run no. 05/

max
avg
min
Cluster
Nodes

63
C.6. Test run 06

Profiler of test run no. 06/

max
avg
min
Cluster
Nodes
C.7. Test run 07
C.8. Test run 08

![Profiler graph for test run 08](image-url)

Profiler of test run no. 08/

- max
- avg
- min
- Cluster
- Nodes

Ping rate

Connected to

Profiler point

max
avg
min
Cluster
Nodes

Profiler of test run no. 08/

max
avg
min
Cluster
Nodes
C.9. Test run 09
C.10. Test run 10

Profiler of test run no. 10/

Ping rate
Connected to
Profiler point
Profiler of test run no. 10/
max
avg
min
Cluster
Nodes

81
C.11. Test run 11

![Graph of Profiler of test run no. 11](image-url)
C.12. Test run 12

![Graph showing ping rate over time with max, avg, and min lines for different profiler points.](image)
C.13. Test run 13
Profiler of test run no. 13/

Ping rate
Connected to
Profiler point
Profiler of test run no. 13/
max
avg
min
Cluster
Nodes

0
250
500
750
1000
1250
1500
1750
2000
0
5
10
15
20
5
10
15
20
25
30
35
40
45
50
55
60
65
70
C.14. Traffic