Peer-to-Peer Architecture for Massively Multiplayer Online Games

Niels Jeppesen and Rune A. Juel Mønnike

Kongens Lyngby 2011
IMM-B.Sc.-2011-7

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-B.Sc.

Summary

Hosting traditional Massively Multiplayer Online Game (MMOG) servers today requires large data centers, due to the use of the client-server architecture. This is a very costly solution, barring smaller developers without monetary backing from developing large scale MMOGs. We wish to change this situation, by developing a significantly less expensive and potentially better solution, based on peer-to-peer architecture. In this thesis we focus on the basic technical aspects of creating and maintaining the peer-to-peer network graph. There are a number of existing solutions on this subject, but they are generally based on technology made for file systems and do not perform well enough nor provide the features required in modern MMOGs. We have developed the concept of a *dynamic graph*, containing a central node, supernodes, and nodes. The central node is the network entry point and network founder. Nodes are participating players in the game, while supernodes run on capable peer machines and make up the actual network graph. Thus the supernodes handle communication between nodes. They exist at a specific location in the game world, which is not predefined, but rather dynamic depending on where they are needed in order to balance the load on the network. We find that our solution provides good performance between players in close proximity in terms of in-game location. As such, it should provide low latency between interacting player, resulting in a smooth and responsive gameplay. We conclude that our solution satisfies the performance requirements of modern online games and may be viable for real games.

Resumé

Online spil med massivt deltagerantal kræver i dag store datacentre som følge af den klient-server struktur der anvendes. Dette er en kostbar løsning, som umuliggør at mindre spiludviklere, uden den nødvendige finansielle opbakning, kan udvikle og vedligeholde denne type spil. Dette ønsker vi at ændre ved at komme med en billigere og potentielt bedre løsning baseret på en peer-to-peer arkitektur. I denne afhandling fokuserer vi p oprettelse og vedligeholdelse af en sådan peer-to-peer graf. Der findes en række forslag til løsninger indenfor dette emne, men de er generelt baseret på teknologi oprindeligt designet med henblik på filsystemer, og mangler både ydelsen og funktionaliteterne krævet i morderne online spil med massivt deltagerantal. Vi har udviklet en *dynamisk graf*, som indeholder en central node, supernoder og noder. Den centrale node fungerer som forbindelsespunkt og grundlægger af netværket. Noder repræsenterer spillere i spillet, mens supernoder udgør selve netværksgrafen og køres på udvalgte spilleres computere. Supernoderne håndterer således kommunikationen mellem noder og er placeret specifikke steder i spilverdenen, som ikke er prædefineret, mens i stedet valgt dynamisk afhængigt af hvor de gør mest gavn. Vi finder, at vores løsning giver god ydeevne for spillere der befinder sig i nærheden af hinanden i spillet. Derved giver løsningen hurtig kommunikation mellem interagerende spillere, hvilket resulterer i et flydende spil med lav reaktionstid. Vi konkluderer derved at vores løsning tilfredsstiller de krav der stiller til ydeevne i moderne online spil og derved vil kunne anvendes i reelle videospil.

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the B.Sc. degree in engineering. It was composed in the period from February 1st till June 20th and grants 15 ETCS points.

The thesis analyzes some of the problems of applying a peer-to-peer network architecture to multiplayer video games, and mainly focus on creating a network graph, which can be employed in peer-to-peer massively multiplayer online games. Our supervisor Robin Sharp has been very helpful in pointing us in the right directions during the process.

Lyngby, June 2011

Rune A. Juel Mønnike (s082938)

Niels Jeppesen (s082927)

Terms

MMOG (or *MMO Game(s)*) Massively multiplayer online game. Many players act and interact in the same fictional world. This is an extension of a normal multiplayer game, which is limited to anything from 4 to 64 players. In MMOGs, several hundred or even a thousand players may play in a connected online game.

RPG Role-playing game. Players control one in-game character and act through it. This genre is somewhat blurred, but may include goals such as leveling up (by slaying bad guys or completing quests), picking skills for the character, gathering loot such as weapons and gold. In general it is based upon improving this one character the player controls.

RTS Real-time strategy. Players act at the same time, as opposed to turn-based where they wait for each other.

Worldstate The state of the in-game world at a given point in time. This could include for instance position of players, states of items (is a chest open or closed?), time of day, etc.

Node Any single entity (a computer or an application) participating in a network.

Supernode A node that is given special treatment (more work or more control) in a network. May be chosen due to certain advantageous properties, such as more bandwidth or processing power.

Churning When nodes connect and disconnect from a peer-to-peer network at a very high rate.

Port In order to distinguish different network packets arriving at a machine, packets are assigned a target port number. Applications then inform the operation system which ports they wish to receive packets from.

Sync (Short for "synchronization"). Worldstates across participating players should agree on important game factors such as player positions, status, actions where relevant. When they do that, they are said to be in sync.

Player character The in-game character (sometimes called *avatar*) the player controls. The representation of the players actions in the game.

Shard When MMOGs choose to run many simultaneous instances of the same game world, each such instance is called a shard.

Zone (or region) The areas a game world may be split into, most often by geographical delimitation.

Indie (Short for "independent") Developers releasing games without being associated with mainstream publishing ways. Games released from indie developers are often only available for purchase on the internet and are not backed financially by large publishing houses.

Server farm A (large) collection of servers, often in a data center.

Gameplay A term describing the collection of rules, mechanics, interactions and more that make up the core idea of a game.

Contents

Introduction

In this thesis we propose a new approach to designing and maintaining a peer-to-peer graph for Massively Multiplayer Online Games (MMOGs). The solution is not strictly peer-to-peer, as we introduce the essential concept of supernodes as peer-hosted mini servers. We hope to contribute to further development in this field in order to provide a low latency solution, which could be applicable for real games.

During the last two decades the video game industry has grown from almost nothing, to a multi-billion dollar industry, where the best selling titles generate more revenue than the biggest movie blockbusters.

Most modern games are online games, which allow players to challenge and/or cooperate with other players through the internet. Especially the group of games known as massively multi-player online games have taken advantage of the massive growth in the number of broadband users, in order to create digital online worlds, in which thousands of players may participate. Several of these games have become immensely popular, such as World of Warcraft which now, more than six years after its initial launch, still has over 11 million monthly subscribers worldwide[3].

Though many different game types and genres exist on many different platforms, almost all of these use some variant of the client-server network model for network communication. This solution works fine for types of games where only few players are connected, however, MMOGs with thousands of players often require huge clusters of servers, reserving MMOGs for big, well funded developers. And even so, the server clusters may still not be powerful enough to handle big gatherings of players without problems.

An alternative to the client-server network model does exist, in form of the peer-to-peer model. However, only a few games have been developed which make use of the peer-to-peer concept, and to our knowledge none of these are large scale MMOGs. The peer-to-peer model has proved both effective and popular for file sharing and a fair amount of research has been done on the subject, but when it comes to using peer-to-peer networking in games only very limited research has been done (see [5, 9, 11, 1]).

In this thesis we wish to analyze the requirements set by both players and developers of modern games, and the technologies available in the context of network communication. We will also investigate some existing solutions on how to handle network communication in games, some of which are used today and some of which are mere proposals. Lastly we propose our own solution based on the peer-to-peer model with some extensions, which handle various current problems with the model. We take a look at an application that simulates this graph on a local machine, in order to evaluate the viability of our solution.

Chapter 1

Analysis of Game Requirements and Technologies

In this chapter we will identify various factors that are important for games, Massively Multiplayer Online Games (MMOGs), and multiplayer games in general.

1.1 Games

Video games are like most other games, it is all about challenges, competition and having fun. Players have been able to challenge their friends in video games for many years, but until the arrival of the internet, video games required the players to be on the same location.

Network multiplayer video games existed long before the internet became widespread, using different network protocols such as IPX, UDP and TCP on the local area network. For obvious reasons these games were mostly designed to handle only a dozen players, but with the internet developers suddenly saw an opportunity to

create an entire online virtual world and populate it with hundreds or thousands of human players. Thus the MMOG was born, but along with online gaming and in particular MMOGs, new problems arose.

1.1.1 Genres

Knowing that a game is a MMO Game, is not telling you much about the actual game, except that it can involve many players sharing a consistent online world. MMO is used as a prefix for more well defined genres of games such as Role-Player Games (RPGs) and First-Person-Shooters (FPSs), thus known as MMORPGs and MMOFPSs. While there are many video game genres, the above mentioned are the two of the most popular today and also they share some important characteristics. Therefore we will analyze the MMORPG and MMOFPS genres, before we consider the requirements for a MMOG network communication solution best suited for these types of games.

As in classic table-top RPGs, the player in video RPGs assumes position of an in-game character and builds up the character through choices and actions. Even in real-time RPGs the pace of the game is often slow and many aspects of the character control, especially regarding combat, are handled automatically in order to keep focus on the RPG elements rather than on how you act in combat. RPGs fit very well into the concept of a large, persistent, player populated world, as RPGs are all about slowly building up a persistent virtual character. This is probably also one of the reasons why MMORPGs are the most popular type of MMOGs today.

In a FPS the player views the world through the eyes of a character, usually a soldier, with a common objective being to eliminate all enemies using the arsenal of weapons provided by the game. Many FPSs are very fast-paced games and requires a high degree of precision, when it comes to movement and especially shooting, where the player, as opposed to RPGs, has to do all the aiming himself. FPSs fit the MMOG concept well, especially when added some RPG elements making the persistence more meaningful. Adding RPG elements to other types of games is very common today, yet very few MMOFPSs have been made.

1.2 Requirements

Before we start looking into technologies and solutions available for MMOG network communication, it is important to establish the requirements in order

for the different genres to function well online.

The different genres of games have different requirements when it comes to latency, bandwidth and level of consistency between players. In fact, even the different phases and actions within a game have different requirements[6]. MMORPGs and MMOFPSs share many (real-time player interaction and movement) of these requirements and are at the same time two very popular genres, which is why these will be examined. In the following we always consider real-time games with direct player interaction, because for example turn based games do not share most of these requirements.

1.2.1 Latency

In general computer networks, latency either refers to the one-way time (time from a package is sent until it is received by the recipient) or the round-trip time (time from a package is sent until a response is received at the original sender). In video games the term latency is often used synonymously with terms such as lag and ping, but the meaning of these terms vary slightly. Lag means to fall behind and is the time from the player takes an action to the game actually reacts to the player's input, meaning that lag can actually appear in single player games due to local machine not being powerful enough. However in most online games, lag is a result of the client awaiting server response before an action is performed, or that client actions are revoked by the server through a scheme such as dead reckoning, often returning the player to an earlier game state. Thus high latency usually result in lag, while lag is not necessarily a result of high latency. Ping just refers to the round-trip latency.

Low latency is very important for both a fluent and a fair game-play in games where players interact. For FPSs a latency of approximately 100 milliseconds has been found to be the tolerable threshold, and for third-person RPGs a latency of 500 milliseconds provides a measurable degradation[6]. From personal experiences we believe, that a latency of 500 milliseconds is well above the level most players will tolerate in a modern RPG, and the MMORPG World of Warcraft does, from our experience, mostly provide a latency less than 100 milliseconds. This leads us to the conclusion that modern MMORPGs should aim at providing an average latency below 100 milliseconds, but that the latency threshold may be much higher. MMOFPSs should not only aim at providing low latency, but are required to keep the latency below 100 milliseconds in order to provide a fluent and responsive gameplay at all time.

1.2.2 Bandwidth

The connection bandwidth describes the amount of data which can be transferred over the connection within a given time frame. Game clients, running on the players' computers, usually do not require much bandwidth. It is however very important that the bandwidth is sufficient, because if not, the latency may greatly increase. For game servers the requirements to bandwidth are much higher than for clients, as opposed to a full peer-to-peer system the bandwidth requirement is be distributed equally amongst all peers. No matter how one decides to handle the network communication, one has to design the system in such a way that the players' bandwidth is sufficient (a qualified guess would be 256 Kbps download and 128 Kbps upload).

1.2.3 Consistency

Keeping a game synchronized and consistent across all participants, while keeping bandwidth requirement and latency low turns out to be a major problem in online games. There are several different ways to solve this issue, all with different strengths and weaknesses. A classic way is to have an authority (usually the server), which rules in case of conflicts. The players usually are allowed to take certain actions before asking the server, in order to reduce lag. In case of a conflict the actions can later be revoked by the authority using for example a dead reckoning scheme. We will not discuss this in further detail, but just note the importance of maintaining consistency, while keeping latency, bandwidth and CPU time, required by the synchronization scheme, low.

1.2.4 Properties

To better compare different solutions to the problem we are working with, we have identified a number of properties, or sub-problems, on which the solutions will be evaluated:

Availability describes the risk of the network being unreachable or completely down. Basically networks with a single point of failure, such as networks with a centralized server have lower availability than fully distributed peer-to-peer networks.

Robustness is how well the solution handles consistency and player churning. Failure to handle these issues correctly can be just as bad as a complete

network failure, since an inconsistent gameworld probably will be game-breaking.

Performance is basically all about latency. However, there are many different factors which may affect latency, such as bandwidth use, refresh rate and player churning. We are both concerned with average and maximum latencies, and it is important to realize that not all messages have the same latency requirements. The *relevance* of the messages differs between players.

Security describes how resistant the system is to both players wanting to manipulate data (cheaters) and hackers wanting simply to break the network. Even though our solution does not handle security, we will briefly look into how well other solutions handle this.

Implementation is about how well we expect a real-life implementation of a given solution to work.

1.3 Technologies

In the following sections we will briefly describe some of the technological concepts used and discuss the various benefits, drawbacks and consequences of these.

1.3.1 TCP

Transmission Control Protocol (TCP) is one of the transport layer protocols used to move packages around on the internet. It guarantees delivery of packets, and at the same time, that packets are delivered to the receiver in the same order they are sent from the transmitter. This is obviously a beneficial property in any application, but it comes at the price of significant overhead.

This overhead stems from multiple sources, all intentional parts of the protocol: First, TCP is a connection-based protocol, so the involved parties must continuously send small bits of information to ensure the other half is still there. Secondly TCP ensures delivery of packets by having the receiver send a confirmation back to the transmitter - if the sender does not receive this confirmation, it retransmits the packet(s) in question. Thirdly TCP packets include checksums for error detection. Finally the protocol does flow- and congestion control which limits the data transmitted depending on receiver and network performance, respectively.

Another thing that affects TCP usability for real-time applications is that, even though the connection is seen as a stream from the applications point of view, data is obviously not transferred on a byte-per-byte basis. Instead bytes are buffered up in the transmitters network stack, and send off when enough data is ready. If the game sends out messages far smaller than this threshold, it means they may have to wait a significant amount of time before truly being send out on the network.

Further worsening the situation on a poor network connection is the fact that TCP may not acknowledge every single packet, but only perhaps every fourth. If then, by the fourth packet, an acknowledgement is not returned, all the previous four packets will be resent. This also raises the problem that if the receiver detects a missing packet, it will hold back any other packets received until that missing packet is successfully retransmitted. This is in order not to break the stream property of the protocol, but means that the application will be starved as soon as one single packet fails to transmit.

These two problems are intended properties of the protocol, and work very well when the issue at hand is simply transmitting data in a reliable way. However for a real-time game, these properties usually are only required for *some* of the data, whereas most of the data is required to reach its destination *fast*. There are ways to tune the TCP protocol settings: We can lower the outgoing buffer threshold, and make the protocol acknowledge every single packet. However, while this might help us reduce latency, it will also introduce further overheads, increasing bandwidth usage, since we will then be sending out more packets and will also be returning acknowledgements more often. This kind of tuning has been used by users to reduce lag in games such as World of Warcraft, since the increased bandwidth is no problem to the clients who are mainly concerned with latency. Even small programs have been developed, which will automatically change the Windows registry setting for unexperienced users.[12]

One last fact to consider is that a TCP packet is comparatively larger than a UDP packet because the TCP header itself is larger than an UDP header. [18]

All in all TCP overhead may be five to thirty percent of the transmitted data. [7] Furthermore the overhead introduced by simply opening, closing and maintaining a TCP connection puts an upper limit to the number of connections a peer can hold, even with no actual data being transferred.

1.3.2 UDP

Contrary to the stream-based nature of TCP, the User Datagram Protocol (UDP) is based on *datagrams* — self-contained, individual packages containing one message each from the transmitter to the receiver. UDP does not guarantee delivery or package ordering, meaning messages can arrive in an arbitrary order or not at all.

UDP is also connection-less, meaning there is no built-in way to know if the other end is still available or if a message is received. This requires additional implementation in the application, if such features are necessary. This also means that it is, unlike when using TCP, *not* necessary to establish a connection between two network nodes before data can be transferred. One simply sends out a message to a given IP-address and port. UDP may contain a checksum for verification of addresses and data, which should of course be used unless one has a really good reason not to do so.[15]

The connection-less nature incurs certain problems when a peer is behind a router applying Network Address Translation (NAT), as used in many homes today. NAT-enabled routers can handle outbound TCP connections (initiated from within the local network to an outside host) because of the connection based nature of this protocol. However, with UDP the router has the constant problem of never knowing which of the local machines a message should be forwarded to — this is the same problem we face with new inbound TCP connections.

In both cases the problem is solved by letting the router know which local machine should receive messages targeted at a specific port. A few years ago it was necessary to have the user do this manually, by logging on to the router administration interface and specifying the details. Luckily, today UPnP is widespread enough that we may rely on it to fix this problem in most cases (see section 1.3.3).

The sacrifice of features such as delivery guarantee and ordering is not fruitless though, as UDP may provide both lower latency and bandwidth usage compared to TCP, which can be feasible for types of messages which are *frequently sent* and where *delivery guarantee and order is irrelevant*. However, for messages where the guarantees provided by TCP are important, there is a price to be paid for using UDP. As stated above it is now completely up to the application to ensure packages are delivered and handle bad ordering, which not only requires an implementation of such, but also may end up generating more overhead than that related to the use of TCP. Still this overhead is of course only present for the messages in question.

1.3.3 UPnP and NAT Traversal

Universal Plug and Play (UPnP) is a term referring to a number of network protocols. In general these protocols can be said to enable devices connected to the same network to discover each other and interoperate more easily.

The key feature that we are interested in, is that UPnP allows applications to enumerate and modify the port mappings of a router or gateway on the fly. This allows us to do *NAT traversal*, in that we can expose a given set of ports to the outside internet by mapping them to a specific local machine on the internet gateway device (which is in most cases a router) — without requiring the user to do anything. The specific protocol that facilitates NAT traversal is the Internet Gateway Device (IGD) protocol. [19]

NAT traversal solves both the important problems of whether an application uses TCP or UDP: For UDP it allows us to receive messages at all, while TCP allows us to accept incoming connections. Of course these are only problems if the application is running behind a NAT-enabled router. Since IPv4 addresses are sparse these days, and has been for a number of years, ISPs are likely to give only one IP per costumer and this can be considered a common problem.[16]

The specifics of how UPnP, and more specifically the IGD protocol works is outside the scope of this thesis, but it is extremely relevant to consider this issue since a direct consequents of being behind a NAT-enabled router, without UPnP, is that *peers may not be able to receive incoming packets.*

1.3.4 DHT

Distributed Hash Tables (DHTs) is a way of storing information distributed over many nodes/peers. DHT is not an implementation in itself, but rather a general technique and implemented in such systems as Chord, Pastry and the like.

Hashing algorithms normally have the desired property of generating hashes fairly well spread out in the used range. This property is used in DHTs to store information evenly among the participating nodes. Likewise, a random number generator is also trusted upon to generate well distributed numbers that are assigned as node IDs.

A piece of information, or data, identified by a key as in an ordinary hash table, is then stored by locating the node with the ID closest to the hash of the data.

In the basic setup, nodes are connected to their nearest neighbors by closeness of ID numbers, as in a double liked list. This forms a sort of "ring" of nodes (since the largest ID is connected to the smallest). Apart from the connections making up the ring, most implementations create additional connections in order to optimize the DHT infrastructure.

When a new node enters the network, it is assigned a random ID and then inserts itself between the closest IDs in the current network, and takes over responsibility for a portion of their data. This may involve relocating some of the data to the new node. If the random number generator is well implemented, it will ensure that nodes always share the data ranges more or less equally.

A way to optimize key location lookup, is to establish connection not just to the nearest neighbors, but to other nodes in the "ring" as well. Chord does this in a way that provides lookups in $O(log(n))$ time.[17]

DHTs are well suited to storing data in a peer-to-peer graph, when the query might be for any data in the entire set. However we do not often in games need access to *any* data, but more often to data related to specific properties of the current gamestate — for example the position of the local player character. Furthermore basic DHT implementation provides no way of querying ranges — like getting information on all players within a certain geographical. Extensions do exist, which allows the forming of multicast trees within the overlay, allowing nodes to rather efficiently broadcast messages to a set of subscribers.[4]

1.4 Conclusion

In this chapter we have examined the requirements of two of today's most popular game genres and we have described the relevant properties of game network architectures. Clearly, for a network solution to have a place in commercial games, the solution has to have several of these properties. We have particularly looked into the performance requirements in terms of latency and bandwidth, as solutions which do not meet the requirements in these aspects are not viable for games at all.

Furthermore we have described some of the technologies which are important with regards to network and peer-to-peer solutions. We have explained the core concepts of these technologies, how they work, and how they are related to the subject of this thesis.

Chapter 2

Existing Solutions

In this chapter we will investigate and evaluate some of the existing solutions for handling online game communication, based on properties we have found to be relevant in our analysis. Firstly we will look into the general client-server architecture and then some of the concrete peer-to-peer solutions.

2.1 Client-server

There has been a countless number of MMOGs released using some form of the client-server model. Small indie-games might use a single physical machine as the server, while larger commercial games use large server farms. For example World of Warcraft had over nine thousand servers globally in 2006[8]. In this section we will examine some of the general principles involved.

2.1.1 Description

Developers often choose to use different roles for the servers, and the complete server side of a game can consist of one or more of the following types: authentication server, world (or realm or shard) server, chat server, and game logic

server. Most large games will also have a separate set of servers for storage, e.g. in the form of a database server, but this is of less interest for this thesis.[20]

The authentication server takes care of players signing in to the game. It checks their login information against the database for correctness, but may also check whether the player is allowed to play (the player could be banned or not have a valid subscription). It may also handle such things as signing up new players, character selection, new character creation, and other tasks that need to be sorted before the game itself can be played.

The chat server handles communication between players. Having a separate server for this allows the developer to impose restrictions and give permissions as of to whom players may talk, depending on or regardless of which server they are playing on or where they are in the in-game world.

The world server is responsible for the actual game play. It analyzes input from the players such as desired actions they wish to take and judges whether these are legal in the current game state. If they are, they will be applied and other connected players will be informed. As we shall see below, the world server may not exactly be just one server.

The game logic server can offload the world server with calculations related to physics, artificial intelligence and more.

This distribution of work onto different servers is not always sharply defined, and may vary substantially from one game to another.

The critical problem to solve is how to handle many players at the same time. Most developers seem to find, that it is simply impossible to have all players in the same virtual world and therefore the first measure is to create multiple, simultaneously running instances of the in-game world, called shards. Each shard will then handle a portion of the players. For example, World of Warcraft has approximately 11 million players[3], they do not all play on the same server, but rather on over 800 different shards (world-wide)[21].

The term "server" is a slight misnomer here, since it is not a single physical or even logical machine running the server software. Each world instance is often further divided into smaller regions, also known as zones or instances. This further helps distribute the load. Whenever a game character crosses from one region into another, he is transferred to the responsible server, often creating a game loading phase. The zones can be either static or dynamic, i.e. they may be responsible for a predefined portion of the in-game world, or their responsibility may be assigned on-the-fly. Dividing the world into zones has the

added advantage that a sudden influx of players into a certain location in the game will not affect unrelated players far away.

Even a single zone server is not bound to run on a single physical machine. It may be virtualized and run on a logical machine that exists across several physical ones — further details about how this works is out of the scope of this thesis, but it allows the software to use more processing power than a single machine can provide.

No matter how the server side is implemented, a common trait (at least in commercial games) is that the client is seen as being completely untrustworthy. Players may use software that modifies the game executable in order to obtain personal benefits over other players. Therefore it is necessary to verify each single action the game client wishes to apply to the worldstate.

There are a number of articles and discussions about how to implement large, commercial MMOGs. However this is proprietary information, and it is in the best interest of developers and their companies to keep their solutions secret, making it difficult to obtain specific information.

[10, 20]

2.1.2 Discussion

An obvious upside to the client-server model, and probably the reason it is used so widely commercially, is that *everything* can be controlled from the server. And the server, in turn, is controlled by those who run the game. This makes it easy, compared to peer-to-peer solutions, to weed out cheaters, perform access control, and securely store information. In the same way, the world-state is always maintained by the server, and as such is never in danger when players disconnect.

Since the server environment is set up by the game developers, the problems related to NAT and household firewalls are not present. It can be assumed that the servers will always be able to accept incoming connections and data. Similarly, hardware and bandwidth can be upgraded and extended whenever needed — there is no dependency on the players to achieve sufficient computation power. The central server also ensures just two hops between the players, thus being able to provide good latency, and since the server knows everything it is also able to distribute information optimally, meaning players only receive relevant information.

This centralized structure is the model's greatest strength, but also the source of its weaknesses. Running a data center is expensive, especially on the scale required by popular MMOGs. There is hardware to be bought, power consumption, the cost of property and the cost of personnel to service the hardware.

One may argue that having a central server structure makes the game more prone to failure. However, any game network needs a known entry point, so peer-to-peer models could be just as fragile in this respect. The problem can be overcome in the same way in both cases by having multiple data centers or entry point servers. For instance, World of Warcraft has 10 data centers, in different geographical locations in North America and Europe [13] (although these are only accessible from within the continent they are located in).

Another fact that goes in favor of a traditional client-server model, is that it is just that: traditional. There is a vast knowledge about how to construct and maintain this model. Large developer studios with the capacity to run data centers in this way are unlikely to invest in another model. Especially when this model holds as many unsolved problems as peer-to-peer does at the moment (access control, cheat prevention, and so forth). However, for smaller developers, it can be prohibitively expensive to run or rent a sufficient number of servers, and this may prevent them from developing an MMOG.

2.2 Peer-to-Peer

In this section we will describe three solutions that have been proposed in order to solve the problem of implementing a MMOG using peer-to-peer.

2.2.1 Mercury

2.2.1.1 Introduction

As mentioned earlier, one of the problems with basic DHTs when using it in games, is that only one value can be queried at a time. The goal of the Mercury protocol is to add multi-attribute ranged-based queries to the basic DHT structure, without degrading the overall performance of the system.

In order to allow for ranged queries Mercury has to abstain from using random and hash values when assigning node IDs, which is providing the load balancing in DHTs. Therefore the authors have had to take new measures in order to try

to balance the load. In order to allow multiple attribute queries, the network is divided into multiple branches, called hubs, each of which can be viewed as a separate "ring" of nodes and each of which handles a single attribute.

[1]

2.2.1.2 Properties

Availability The system uses the standard DHT approach where the connection to the network can be obtained through any node. This provides high availability, since in theory there is no single point of failure. However an incoming node needs to know at least one node currently in the network, and thus need to acquire this information somewhere.

Robustness State replication is not really handled, but each hub could hold a data replica rather than a data pointer. Since the DHT structure is used we suppose already known replica solutions for DHTs could be applied to Mercury, but because it is not discussed in the article we are not sure how straight forward this would be.

Performance Compared to regular DHT overlays the Mercury multi-query system provides a good way of getting the relevant data, but the increased functionality is at the expense of speed. The average number of hops seems to be between 10 and 20 for 10k to 30k nodes, meaning a query has to pass through a several nodes before a result is returned. In the example games shown in the article with 20 and 40 players the average number of hops are 4.44 and 4.61 respectively, rendering the system unsuited for FPSs even with few players and unsuited for RPGs as well, as the number of hops grows, resulting in a total latency of 250 milliseconds for 5 hops with 50 milliseconds inter-node latency.

Security This issue is not discussed at all in the article. The structure provides no obvious way of managing authority or avoid cheating.

Implementation

- The tests including the game implementation gives some idea of the system performance, which might be sufficient for some types of games.

- The system does not handle data replication or persistence in any way.
- Router firewall and NAT is not accounted for in the system, which assumes all nodes can connect to each other, which might not be the case in real world situations.

2.2.1.3 Conclusion

The Mercury protocol provides some interesting features, which are not present in the regular DHTs and the system seems to scale quite well, even with thousands of peers.

However, we feel that the solution is very general, and not really aimed at finding a MMOG peer-to-peer solution. The system does not handle replication, persistence, authority or security in any way. Furthermore the solution disregards the problems posed by router firewall and NAT, making us wonder if it could even be successfully implemented in a consumer application at all.

2.2.2 Peer-to-Peer Support for Massively Multiplayer Games

2.2.2.1 Introduction

The solution proposed here is also based on DHTs, but takes advantage of the Pastry extension Scribe, which is a multicast tree protocol.

The solution divides the world into regions and assumes players are only interested in information about their current region. Player interaction messages are sent directly via UDP, while updates relevant to all players are broadcasted through Scribe. Each region has an coordinator, which handles shared objects and may resolve disputes between players, and a backup coordinator is ready to take over if the coordinator leaves.

While players may move between regions there is no inter-region interaction and there is a strict limit to how many players there can be in a region.

[11]

2.2.2.2 Properties

Availability The solution uses a centralized account and management server, which can be considered a single point of failure.

Robustness The system handles coordinator churning through the backup coordinators, but in the experiments only one backup is available for each coordinator, which may be insufficient. More backup could however be added, since *all* of these have to fail simultaneously in order for a region failure to occur. The shared game object state is broadcasted best effort, which may result in inconsistencies being present a while before being corrected.

Performance The tests show 1% of messages taking more than 18 hops, which could be a problem, as it might result in lag-spikes, depending on the message type. The average number of hops is low though, giving a decent performance. Player-interaction should be really fast, since this is done directly between players.

Little bandwidth is used as nodes, according to the article, receive 50-120 messages per second, due to the use of Scribe, which is a positive feature.

Security The use of supernodes and a centralized server provide some security and also player interactions are executed by all parts affected by the actions, where coordinator is arbiter. However the article states that the solution does not deal with security in detail.

Implementation

- While the tests shows that the solution can handle thousands of peers, a basic requirement is that there are only few players per region and the regions are separate. This, we feel, is a major restriction, which means that a game using this solution can not provide the experience provided by modern MMOGs where hundreds of players may gather (though also sometimes causing major server problems).
- The use of Pastry and Scribe still means the solution has problems with routers applying NAT, which again is a problem that needs to be solved for the solution to work in consumer applications.

2.2.2.3 Conclusion

This solution is definitely interesting and may work for games with many small regions, which players can move between. The use of Scribe and UDP, as well as the way coordinator manages region and shared objects is quite clever. With this said, we think that the player limit per region is very small (10 in the tests), and we do not agree in the way this removes the possibility of having big shared open worlds, which is a part of what we think MMOGs are all about.

We do like how the use of coordinators makes the system able to handle the router NAT problem. The system shows good scalability, but still only under the given restrictions.

A system much like this is described by [9].

2.2.3 Hydra

2.2.3.1 Introduction

Hydra is not a complete implementation in itself, but rather a platform that developers can build upon. As such, there are some features that have been deemed out of scope by the authors. It provides a layer between the game client and the underlying network. This layer enables a peer-to-peer network between involved nodes, and provides failure resistance.

In short, it works by splitting the player-base into many smaller regions. Each region has a primarily responsible "slice", as well as one or more backup "slices". All slices are instances of a game server, running on any machines in the network. Clients will be connected to the primary slice, which will forward all messages to the backup slice(s). Should the primary slice fail, a backup slice will take over.

The protocol in use is UDP, but with an extra layer to provide optional reliability for messages.

[5]

2.2.3.2 Properties

Availability Hydra uses a "global tracker" as the entry point to the network. This is a well-known server to all clients, to which they will connect in order to be informed which slice they should connect to. It is of course a single point of failure, but since it does little else than keep a list of primary slices, it is unlikely to be overburdened and crash.

Robustness This depends on the number of backup slices. All messages are sent to both a primary supernode and one or more backup nodes. If the primary node fails the backup nodes will have the same game state, and nothing is lost. Since all clients maintain a connection to their relevant backup slices, and backup slices are always reasonably up-to-date on the worldstate, the failure of a primary slice has minimal effect on the game. One of the backup slices will simply become primary, and a new backup slice can be created.

Since the number of backup slices can be chosen arbitrarily, the Hydra architecture can be made as stable as one wishes it to be. The entire system of synchronization between primary- and backup slices, and the process of fail-overs seem very well thought out. The only case in which the system can fail, is if *all* slices fail at the same time, which is unlikely to happen except in the case of a total network failure.

Performance In the overlay structure of Hydra, there seems to be only ever two hops between clients: one to their common slice, and one to the destination client. However, the system uses a step time of 150 milliseconds, which is the frequency with which messages are sent out. This implies that two hops take 300 milliseconds, which is not fast enough for FPSs and even too slow for RPGs by today's standards. It is possible that this could be fixed simple by lowering the step time, but the article does not discuss what implications this would have.

This hop time assumes that clients which have interest in each other's information exist in the same region, since there is no inter-region Hydra, in the form it is presented in the article, only supports sharing information with players connected to the same server. It is left to the developer of the application using Hydra to work out how transitions between regions (and thus servers) should be implemented.

Due to hardware limitations the system was not tested with more than 15 clients, according to the article. This seems like a very small number, considering that

this will limit the number of players that can interact at any time. There is no cross-server communication, so players will not be able to see players in other in-game regions.

Security The fact that Hydra uses a global entry point can be used to provide some means of client authentication. However, the article declares that "[...]cheat prevention for the Hydra architecture remains as future work."

One could imagine that it would be possible to use the existing structure of a primary slice with several backup slices to perform verification of client actions. The primary slice could even offload all this to its backup slices. These slices would of course still be running on client computers, and therefore distrusted, but such is the nature of peer-to-peer network.

Implementation

- The tests mentioned in the article include only a small number of nodes, so it is difficult to know how Hydra would fare in on a true MMOG scale.
- The global tracker maintains a list of clients that could be turned into primary- or backup slices. This could be used to avoid fruitlessly attempting to connect to a machine behind a firewall.
- Hydra "delegates the responsibility for scalability to the game developer", meaning that it "is up to the game developer to divide the game world into separate regions so the expected load on each slice (i.e. number of clients connected) will not be excessive"

2.2.3.3 Conclusion

Hydra seems like a viable solution, if one can design a game so that the number of clients in each in-game region is small. It makes the very daring assumption that the game world is separated into *small and completely disjoint* regions. In our opinion, this is a very harsh restriction to impose. Due to the size constraints we feel that inter-region communication should be present, which would be impossible in Hydra without further extension.

The platform furthermore has some problems with scalability, in the sense that it does not deal with this in any way. If too many players enter one region,

the behavior is undefined. This can be assumed to mean that it will become unplayable at some point, if a relatively small population limit is not enforced.

All in all it seems that Hydra, in the state is it described in the article, is not ready to be put into use for a MMO game. However, it is a quite robust system for smaller games without strict latency requirements, with respect to failures and synchronization.

2.3 Commercial Use of Peer-to-Peer

There are no large commercial-scale MMOG implemented using peer-to-peer networks, and it is difficult to find more than vague discussions even on the indie- and amateur stage (i.e. discussion boards on the internet). The possible reasons for this have already been touched upon in section 2.1.2, and can be summarized as follows:

Access control It is difficult to control who enters the network, and verify who they are.

World control It is difficult to control what happens in the worldstate while the game is being played. This opens up problems with item duplication, people walking through walls, players not taking damage, and probably a lot more.

Security If data such as player character information (items, strengths) are stored in the peer-to-peer network, we can not guarantee people do not change it. The only way to do this would be to stored it in a controlled environment, such as a developer-owned server.

Habits and tools It is very well researched and known how to build client-server environments in the regular sense, where all the above problems have viable solutions. Developing a peer-to-peer based game from scratch would be costly and dangerous (in the business sense of the word).

Controlled environment In a classic client-server setting, the developer controls all factors about the server environment. For example, developers know they can upscale data center capacity while it is impossible to upscale the computational power or bandwidth of clients.

These are all serious issues that would need to be solved, before a peer-to-peer network could be used in commercial context. However, the technical (habits

and tools) issue seems to be the most pressing in order to move this field forward. It appears that there are no existing solutions capable of handling real-life scale MMOGs.

Some of the problems listed above may be solved simply by circumventing them. For instance, one could avoid the need to upscale computational power, by making sure that each player exerts no greater pressure on the peer-to-peer graph than he himself provides. The need for access control could be circumvented by changing the business model or gameplay in a way that made it uninteresting for players to cheat it — much like Wikipedia made it uninteresting (at least to a degree) to vandalize its entries. These issues are out of scope for this thesis, though, in which we look at the basic technical issue only.

2.4 Conclusion

In this chapter we have examined the regular client-server structure, as well as three existing peer-to-peer solutions for massively multiplayer online games. It is obvious that the client-server structure has several beneficial properties, and this becomes even more understandable when we looked into the commercial use of peer-to-peer in games. The peer-to-peer solutions we investigated showed potential, but they also showed that the concepts they are built upon are not fully developed and therefore clearly do not meet the requirements of modern MMOGs.

Aside from this, it seems that the existing concepts are mainly built on structures originally made for peer-to-peer file systems (DHTs), They lack both the performance required in games and the support for truly massive open gameworlds.

Chapter 3

Proposed Solution

In this chapter we will introduce our proposed solution. We will discuss our ideas, and the related theory. Finally we will look into how the proposed solution can be implemented.

3.1 Ideas

One of our main goals, in this thesis, has been to construct a network graph, which would result in as few hops between nodes as possible in order to provide a low latency. While using DHTs may provide a good average latency, the client-server architecture is far superior since all nodes are connected through two hops only. We therefore wished to explore different network graphs, in order to see how well these would perform and if they they could perhaps even be combined.

3.1.1 Supernodes

One of the first conclusions we reached to, was that we wanted to use a supernode-structured network. This kind of structure differs from a pure peer-to-peer, by

granting some nodes special tasks. This unbalances the network, but it is not necessarily a problem as long as the load is not *too skewed*. As we have seen, both [11] and [5] use this structure, referring to supernodes as coordinators and region managers. Supernodes are also used by Skype[14], where millions of users are online simultaneously, showing that the use of supernodes in consumer applications definitely is a viable strategy.

The use of supernodes enables us to solve some problems of high importance. Firstly, by letting regular nodes connect only to supernodes, we handle the NAT issue, assuming only nodes which allow incoming connections are assigned to be supernodes. Secondly, supernodes may serve as an authority, much like the server in a regular client-server system, helping to provide both consistency and security. As we shall see later the supernode structure is also important in other parts of the solution.

3.1.2 Network Graph

The region based strategy proposed in [11] and [5] is very much an either-or solution: If the player is in a region he receives all information, and if he is not in the region he receives no information at all from that region. We really do not approve of this isolation of the regions. It results in the game world being divided into a series of small, totally isolated areas which the player moves between, as opposed to a vast open world.

Our key idea therefore was, to use the geographical location of the players' in-game characters to create a network graph. This ensures that nearby players are connected in very few hops. The number of hops then increases the further apart the players are. After making this our goal, we started looking for a graph which would satisfy this goal.

3.1.2.1 Tree Graph

The first type of graph we looked at in order to achieve our goal of having few hops between nodes was a tree graph. One of our first concepts can be viewed in figure 3.1. In this graph we would have the root of the tree being a central server, which would be both the entrance point, as well as provide persistence and handle inter-region communication if necessary. The central server could in principle be hosted from a regular PC, but the requirements in bandwidth and computing power will vary much depending on the number of nodes, the level of region isolation and many other parameters.

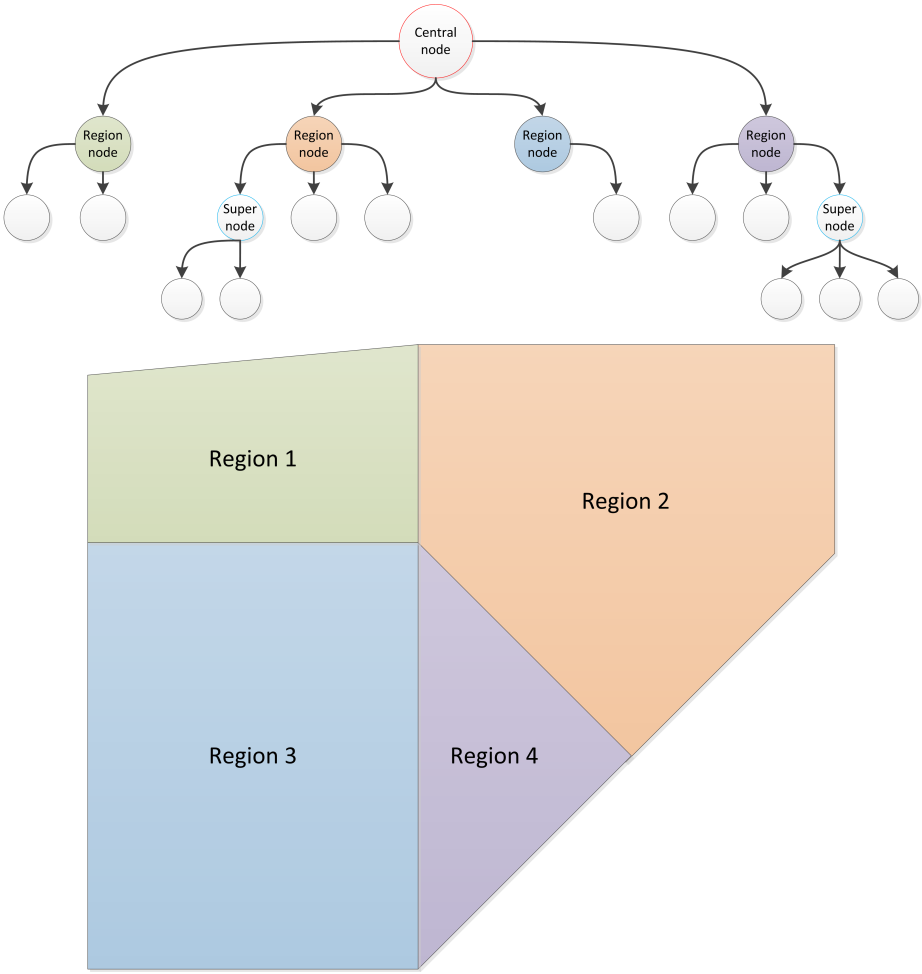

Figure 3.1 – Tree graph network with fixed regions.

When a node connects to the central node, it is forwarded to the region node responsible for the region the player is in. If no such node exists the new node is made responsible for this region, and thus becomes a region node itself.

When a node arrives at a region node or any other supernode it is either added to the supernode's set of children or forwarded to a child supernode. The latter happens if the supernode has reached its maximum number of children (the tree's branching factor). Children are promoted to supernodes when needed.

When a supernode leaves, the tree will of course have to be repaired. Repairing

can be managed easily by the parent of the leaving node, assuming all nodes in the tree know the path from themselves to the root.

If region sub-trees are well-balanced then even with thousands of nodes the depth (effectively hops between nodes) will be small if we use a huge branching factor (number of maximum children per supernode). For example with a branching factor of 10, we will have a depth of only 3, for 1000 nodes. The maximum number of hops between two nodes will then be 8 $(2 * (1 + log(n)))$.

It became apparent, however, that this approach had a series of critical problems. Firstly, the performance in terms of number of hops was, in some cases, not better than that of the DHT-based solutions we had examined. For example, in edge cases, players close in-game would be on totally different sub-trees meaning information would have to be routed all the way up and down in the tree. Secondly, the load on the supernodes was (almost) linearly dependant on the number of nodes in the region, since they had to receive information about all other nodes.

From this we concluded that in order to allow many players in one region, it is critically important to be able to *determine whether information is relevant to a node, before you send it.* If you cannot do this you will end up forwarding way too much information and the entire network will be flooded with irrelevant information.

3.1.2.2 Grid Graph

A problem with the tree- and DHT-based graphs is that they do not exploit the fact that players in character-based games, such as FPSs and RPGs, are usually only interested in their immediate surroundings. The division into regions makes some use of this, but we believe it could be further exploited in order to reduce latency between nearby players.

Wanting to exploit this property, we started designing a viable grid network. The basic concept can be seen in figure 3.2, where the supernodes are laid out in 5x5 grid. The basic idea was that players are always connected to the nearest supernode. The dimensions, density and form of the supernode-grid should of course fit the gameworld, such that the population of players are distributed equally amongst the supernodes.

This grid structure ensures that adjacent nodes are connected through *few hops.* In theory, the closer they are, the fewer hops. This is a property we find critically important, as it ensures nearby players — which are the most relevant to a given

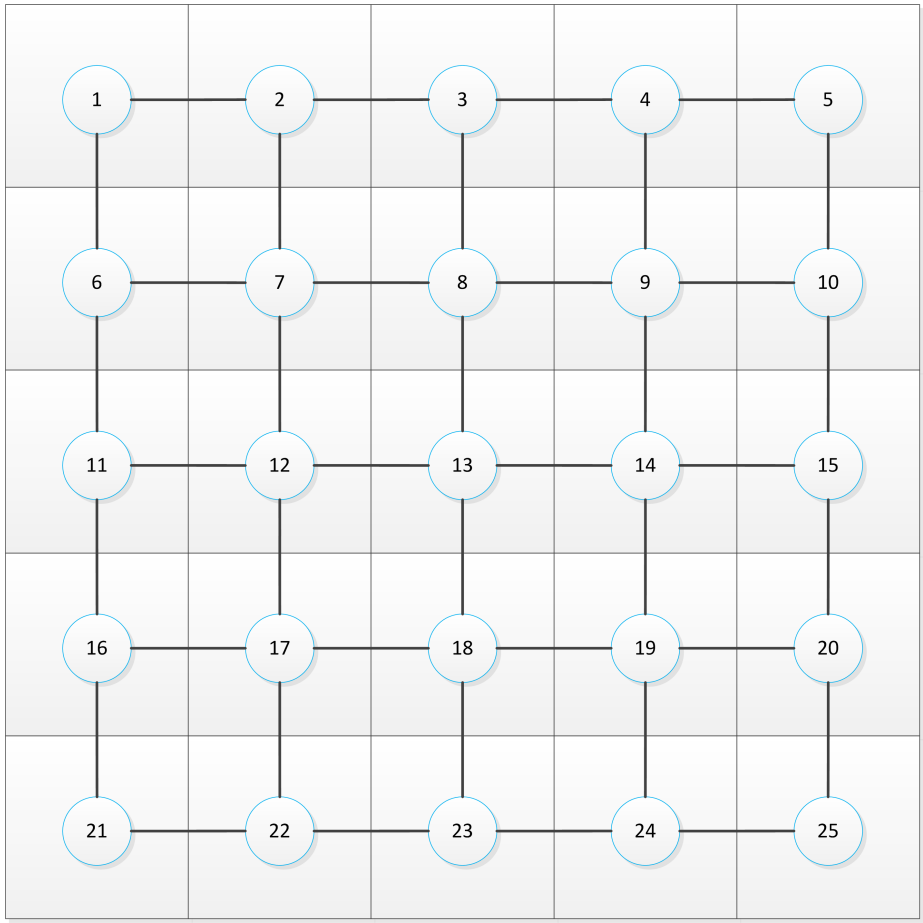

Figure 3.2 – Simple grid graph

player — respond with low latency. Also, since we assume players are always connected to the nearest supernode, we here have a way of knowing if a message is *relevant* to a given supernode, based on the area in which the supernode may have children.

The grid can either be static or dynamic. With a static predefined grid, we can make the grid very specific to the gameworld, which allows us to optimize the graph in several ways. If we for example have big objects such as walls or houses, blocking the view of the players, we may chose to drop some connection and/or supernodes as shown in figure 3.3 in order to reduce bandwidth use. The basic static approach has two main issues though: Since we have predefined

the number of supernodes, player gatherings which are not accounted for in the setup will place heavy load on few supernodes. Furthermore there is no guarantee that we have access to the specific amount of supernodes (i.e. not enough players with open NATs), thus we need some way to construct the graph with less supernodes than it was designed for.

This lead us to the dynamic approach, which intuitively should solve exactly the two issues which troubled the static structure. However, it is not immediately clear how grid density can be changed without having to move *all* supernodes,

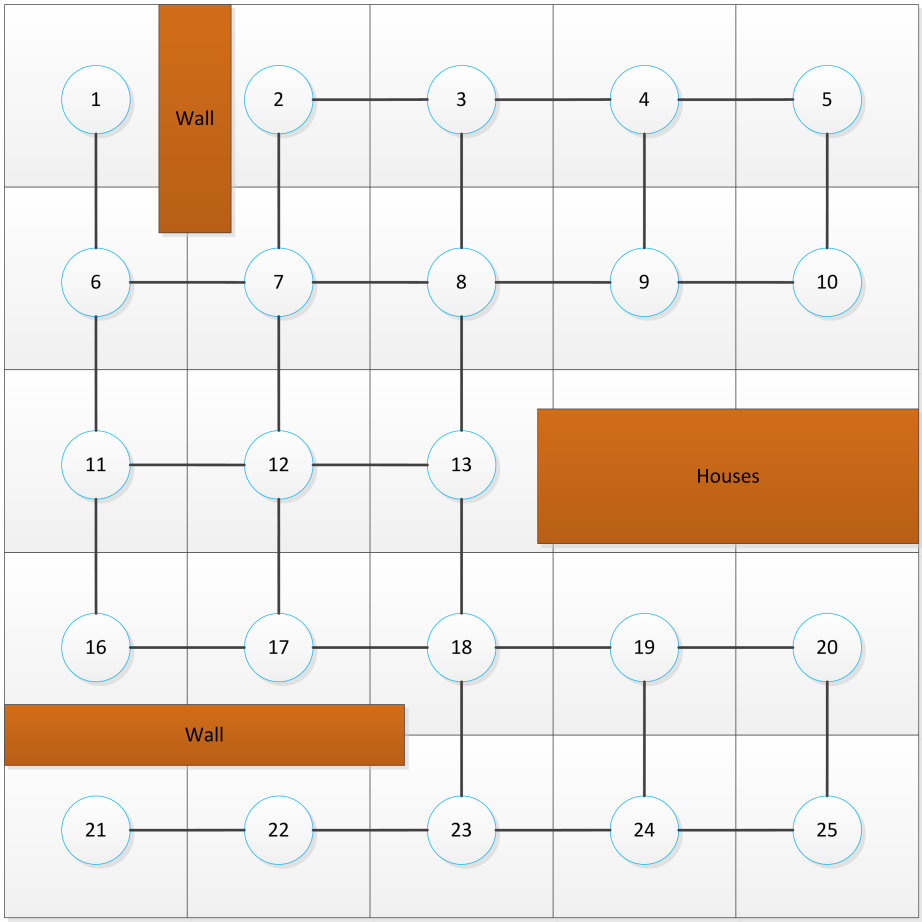

Figure 3.3 – Grid graph with some connections left out because blocking objects make them unnecessary

basically causing an expensive total restructuring of the network. Furthermore increasing the density of the *entire* grid is clearly not an efficient way of tackling the problem of players gathering in specific areas very well.

While the grid graph is good for distributing area specific information, it has some drawbacks. The way messages are passed between supernodes means that a supernode may receive the same message through different neighbors, which may increase bandwidth use several times (this drawback does however have the positive side effect that supernode failures may have less influence, since messages are likely to be routed around the failed node). Also it is both expensive and slow to send information to all nodes in the network.

To provide a more efficient way of flooding the network one could use a tree or DHT overlay to handle global information, thus having several different graphs simultaneously.

3.1.2.3 Unbalanced Grid and Tree Graphs

Clearly in order to handle a geographically poor distribution of players, we had to create a graph with differently sized areas of responsibility for the supernodes, such that each supernode handles roughly the same amount of players. Figure 3.4 shows the *areas of responsibility* for such a graph, where the colored areas are the most densely populated areas, while area 1 is the least densely populated area. For the grid-based solution one could have each supernode connected to all neighboring nodes. In the figure this would give node 42 six neighbors, while node 22 would have just two. This is just one of many ways connections could be established.

As player density in different areas change the graph should *adjust*. For example, if more players moved into area 1, the area should be split into four smaller areas, and as players leave areas, the opposite should happen. By doing this we ensure both that supernodes are not overburdened and that the latency between nearby players stay low.

At some point it will no longer make sense to divide the areas of responsibility, since all players in the area are so close to each other that they are all more or less equally important. Also, if the areas are made very small, players will have to change supernode very often when moving around, possibly generating unnecessary overhead. We may then simply stop dividing areas and hope that the responsible supernodes can handle the amount of players present, or block new players from entering the area.

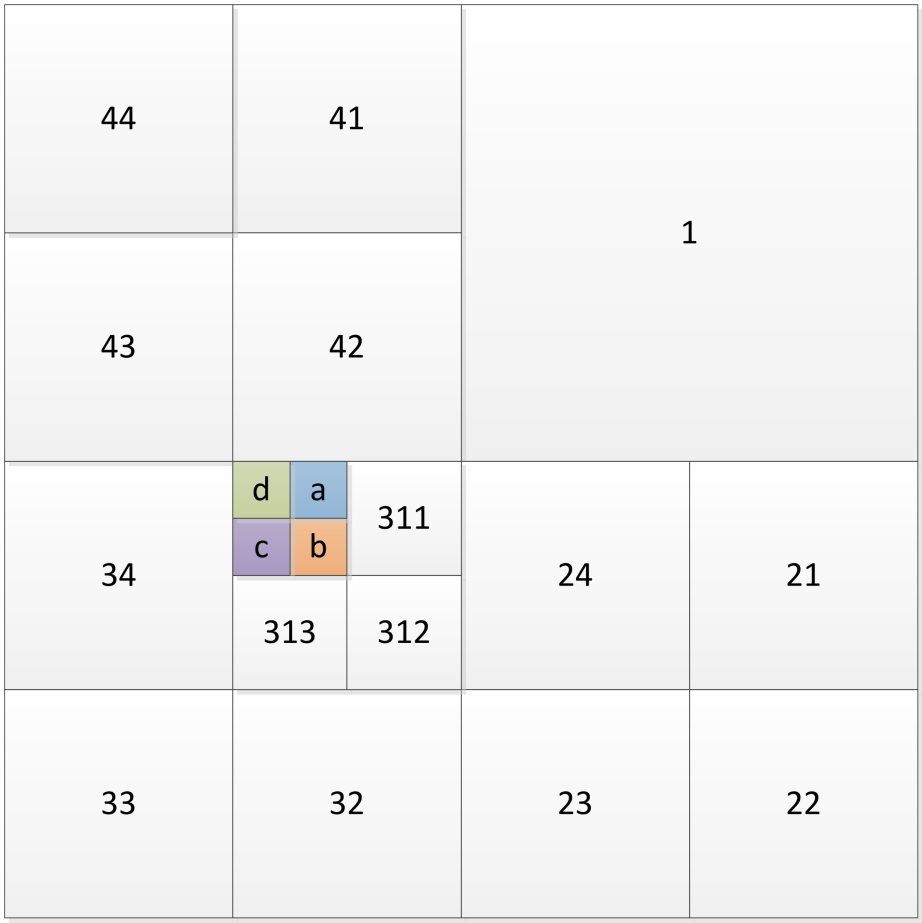

Figure 3.4 – Grid graph with differently sized areas of responsibility

Having explored this use of dynamically sized areas of responsibility, we looked at how this could be applied to the tree-graphs we explored earlier. We once again look in figure 3.4, but rather than having the responsible nodes connecting to their neighbors, they are connected through the tree in figure 3.5. The difference between this tree and the previously discussed trees is that this tree is not balanced and the supernodes are each bound to an area of responsibility.

Just as for the grid, we can then determine if a message might be relevant to a supernode, before forwarding it, thus significantly reducing bandwidth use for supernodes close to the root. At the same time the new structure also means that most geographically close nodes will also be close in the tree structure.

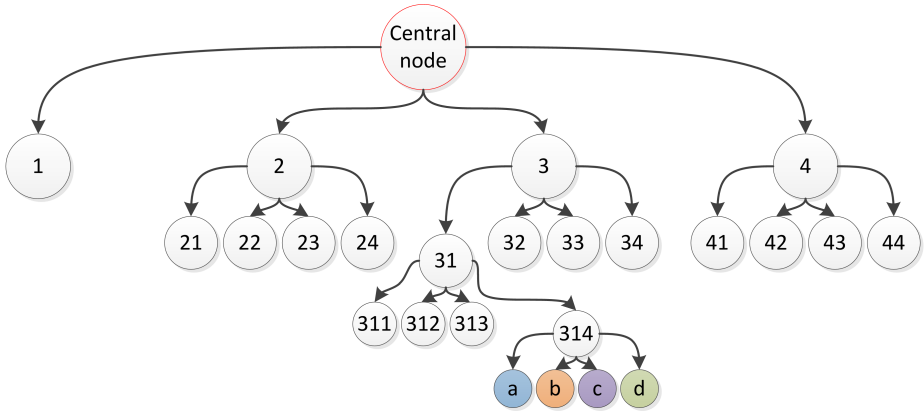

Figure 3.5 – Tree graph for figure 3.4.

However, the latter is not always the case, which is one of the main issues with the tree. The problem is that around the edges of the areas players may interact with players in entirely different subtrees. For example, in the figure 3.4 there are seven hops from a node in area 311 to a node in area 24, which will most likely result in noticeable lag, if those players start interacting.

The edge problem related to the tree graph might be solved by adding direct connections between subtrees, or simply combining the grid and tree approaches completely, which might actually be interesting. At this point, however, we developed another graph concept with a more promising architecture, which made us stop exploring these structures further.

The solution given in [2] uses a region structure, which shares some properties with grid and tree concepts proposed here.

3.1.2.4 Dynamic Grid

The graph we ended up finding most interesting is very dynamic and loosely built, compared to the grids and trees we had previously explored.

The core concept of the graph is: We assume players, or nodes, move about in a 2D space in the game-world (a third dimension is simply ignored). Much like in the regular grid structure, supernodes are placed in the same space as the nodes, although they are of course not visible to the players. But unlike regular grids, the supernodes are not placed in specific predefined locations

but may be placed arbitrarily in the world. This also means that connections have to be made such that the network is not broken into several sub-graphs, since supernode neighbors are not well defined. The regular nodes then simply connect to the supernode closest to their position in the world. Figure 3.6 shows such a network.

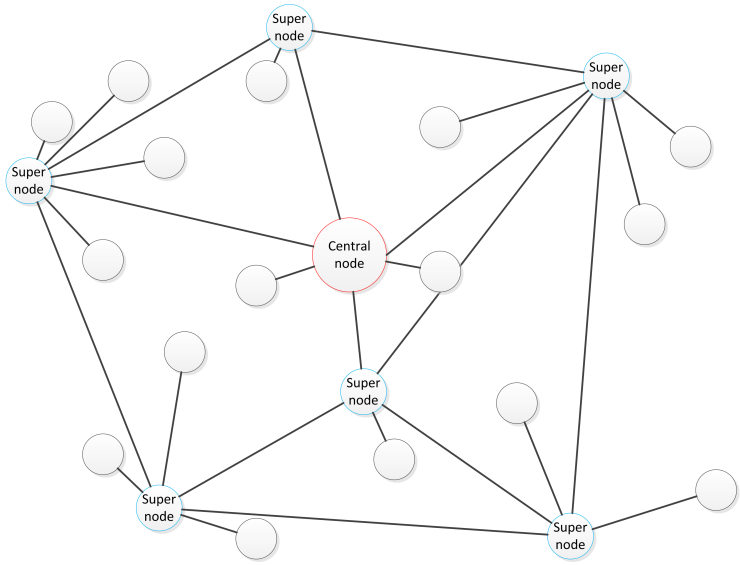

Figure 3.6 – Dynamic grid graph

In order to avoid the graph breaking into pieces we propose the following: Assume a supernode is placed in the origin of a Cartesian coordinate system. Then the supernode should have at least one neighbor in each quadrant. This ensure the graph remain in one piece.

This approach shares many properties with the basic grid structure, but is much more flexible when it comes to supporting the arbitrary movement of players in many differently shaped environments. Two of the key factors are of course how supernodes are placed and how they are connected, but it is exactly this choice that provides the flexibility.

The general idea is that the supernodes are connected to their supernode neighbors. As stated earlier, this property suggests that nearby nodes are closely connected in the graph, meaning there is only very few hops between them. While preserving this key property, we are also able to place the supernode exactly where they are needed. We are not forced to place them in specific locations in order to preserve any specific pattern, just as players are not likely to

position themselves in any specific pattern either. We will explore this solution further in the following sections.

While we in the structures above operate in two-dimensional spaces, the task of moving to three dimensions seems trivial. The reason we stay in 2D is that, while most games are in 3D, the third dimension is usually significantly more limited than the others and most players usually stay on the ground plane. For some games moving to three dimensions might be relevant though, such as games taking place in outer space.

3.2 Theory

We will now take a further look into some of the structure and theory behind our solution including theoretical performance and behavior.

3.2.1 Types of Nodes

In our system we propose three different types of nodes with different responsibilities.

Nodes are much like clients in a regular client-server system. They represent a player in the game and are connected to a supernode, but have no obligations beyond this. If suited, they may be called upon to set up a new supernode on the peer's machine.

Supernodes are run by peers, but are separate from the nodes with which they share machine. There are several good reasons for keeping the two separate, both with regards to security and robustness. A supernode acts as "mini-server" for group of nodes, and is also connected to some of the surrounding supernodes which we refer to as neighbors. Even though the supernode has a game-world position, it does not actually exist in the game, but only in the network. Just like a regular server, supernodes can be used as an authority to provide consistency and security.

The Central Node is the manager and founder of the entire network. The central node has all the functionalities of a supernode, but may also act as a network entry point, persistence manager and final authority. As entry point the central node supplies information about the network to new nodes and keeps a list of nodes, which can be called upon to create new supernodes.

The central node could be hosted by anyone, but for a big MMOG it might require more bandwidth and computing power than most people have at home. Alternatively, one might choose to distribute the tasks of the central node amongst different computers — and/or have several central nodes to provide better availability and spread the load.

However, the requirement of hosting the central node should still be nothing compared to hosting a regular MMOG server. The exact roles of the nodes will vary depending on the requirements of the game.

3.2.2 Behavior

In order for the system to remain consistent and avoid failure, it is critically important that the behavior of the nodes are well-defined. Since both servers (supernodes) and clients (nodes) are hosted by peers, the peer-to-peer structure has to deal with more scenarios than client-server systems. Let us first look at the stable network shown in figure 3.6 and then go through different scenarios.

3.2.2.1 Node Arrives

The player arrives at some location in the world. It may be predefined, random or based on stored data. In order to ease load on the systems it is wise *not* to have all players spawning in the same location.

Node arrival starts with the node contacting an entry point, in order to obtain information about the current supernodes in the network. Using that information the node then connects to the supernode closest to the node's current position. See figure 3.7.

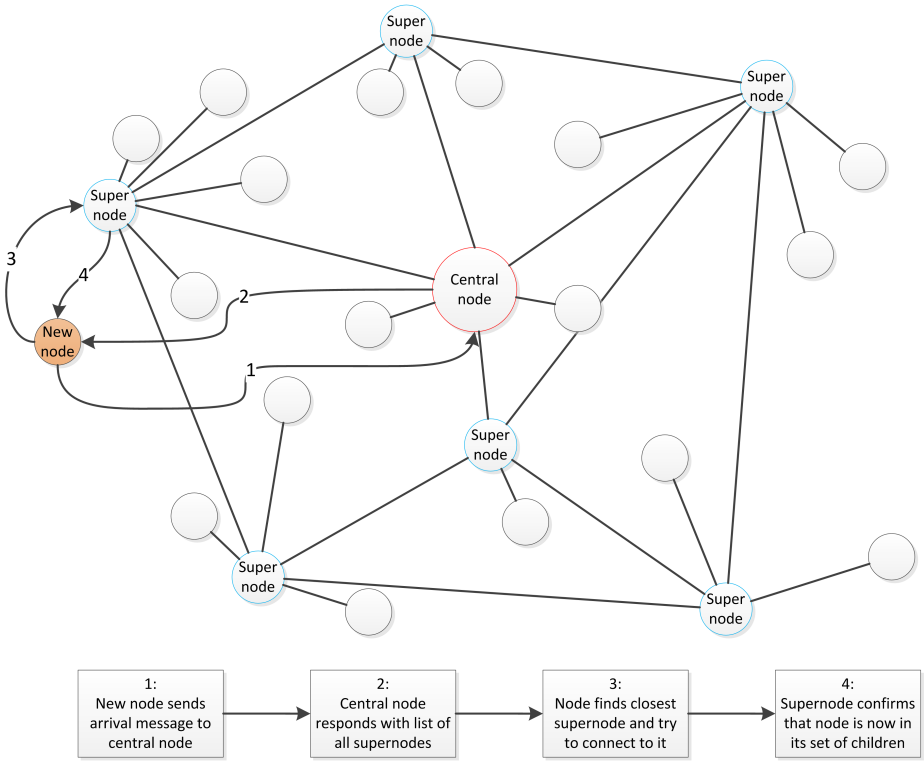

Figure 3.7 – Node arrives.

3.2.2.2 Supernode Arrives

As the threshold of the number of child nodes is reached by a supernode it may send a call for help to the central node. It is then the central node's job to provide a new supernode from the set of applicable nodes. However, is it the responsibility of the supernode requesting help to provide the location of the new supernode, since only this supernode knows where a new supernode would aid the most.

When the supernode is in place it will try to find and connect to its neighbors (other supernodes), as well as send out a message about its arrival and position. This message should be forwarded to the entire network, such that all nodes can evaluate the new supernode, and use it if feasible. See figure 3.8.

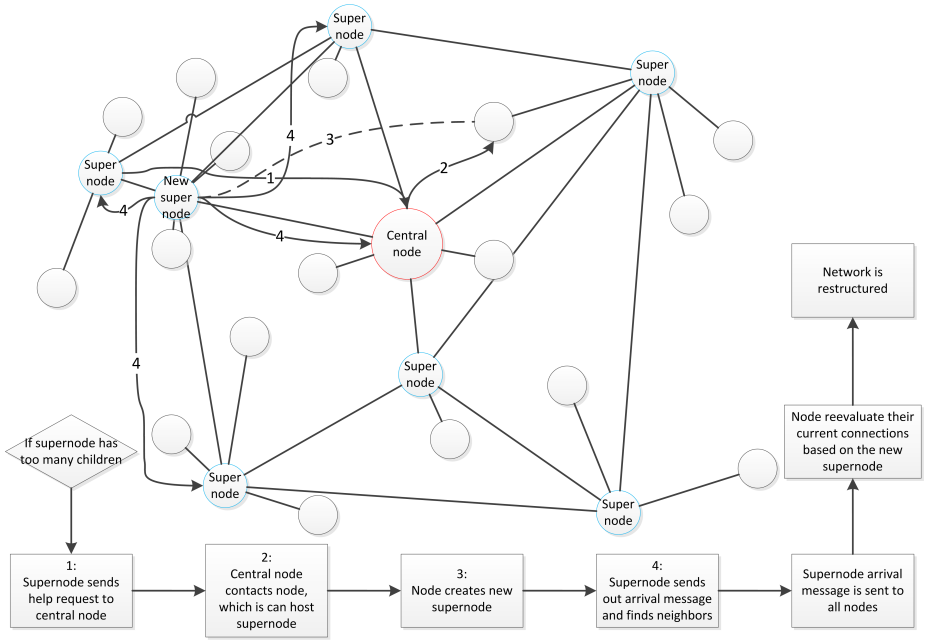

Figure 3.8 – New supernode is created.

3.2.2.3 Node Moves

To keep the property that nodes are always connected to the nearest supernode, nodes are required to change supernodes as they move around. This can be accomplished simply by having the node finding the closest supernode once in a while. Even if the interval is small and there are thousands of supernodes, computation will be easily handled by today's processors. See figure 3.9.

However keeping the previously mentioned property might not always be beneficial, as we shall see in section 3.3.2. One might choose to keep the current supernode even if it is not the closest, as long as it is still very close to the node. This means that we now only guarantee that the node is very close to its supernode *or* has the closest supernode.

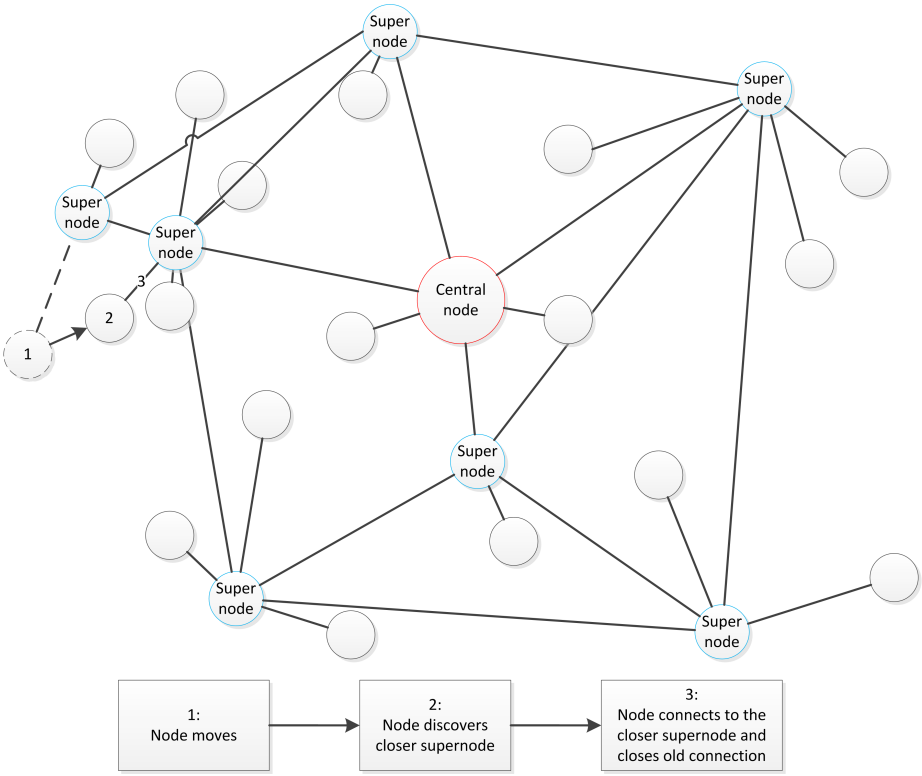

Figure 3.9 – Node moves.

3.2.2.4 Node Leaves

Handling leaving nodes is straight forward. When a node departures gracefully (not due to failure) it simply informs its supernode that it is leaving and closes the connection. If the node fails the supernode will detect this with a timeout, and clean up just as if the node departed gracefully. In both cases the supernode should notify the central node, so the central node no longer believes the node is available as a potential supernode (if this was the case).

3.2.2.5 Supernode Leaves

There are three possible reasons to why a supernode might leave: The peer hosting it leaves gracefully, the peer hosting it fails, or the supernode is shut

down because it is no longer needed.

When a supernode leaves gracefully it informs its children and neighbors that it is leaving, but then remains active until all these have closed their connections or a certain amount of time has passed. This way, the other nodes have time to find a suitable substitute for the leaving supernode, which allows the game to continue more fluently than had the connection just been cut. See figure 3.10.

In case a supernode fails, the nodes should inform the central node which will await confirmation from several parts before announcing the departure of the supernode to all nodes. This way it is less likely that a supernode is mistakenly announced dropped. Nodes discovering a failed supernode will not wait for confirmation from the central node, but will immediately attempt to connect to a different one.

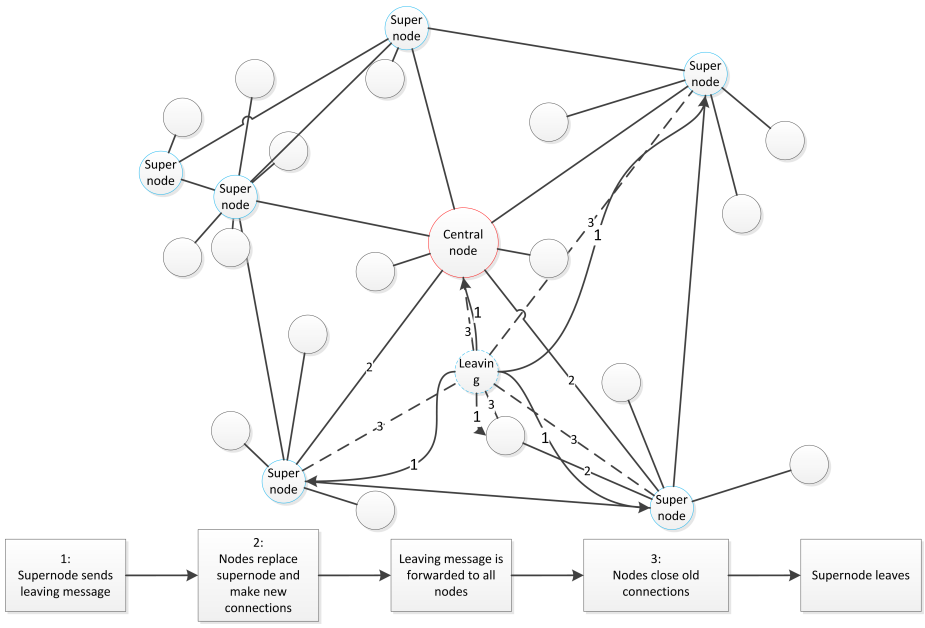

Figure 3.10 – Supernode leaves.

3.2.3 Performance

As we saw in section 1.2, good performance is vitally important in order to meet the expectations of today's players. We will therefore analyze the theoretical performance of our solution.

3.2.3.1 Latency

The latency of the system is directly influenced by the number of nodes a message passes through before reaching its destination. The actual latency is the sum of the latencies on the message's path, but we have no direct control over these latencies and thus our goal is to make the path (number of hops) as short as possible.

The relatively loose structure of the graph allows the graph to take very different shapes, effectively influencing the speed in which messages are spread. We consider s to be the number of supernodes and n to be the number of regular nodes in the network. We know that $n/c_{max} < s < n/c_{min}$, where c_{max} is the maximum number of children per supernode and c_{min} is the minimum. $hops(n_1, n_2)$ is the number of hops between two nodes n_1 and n_2. We then see that:

- The supernodes may form a line, in which case the $hops_{max}(n_1, n_2)$ for any pair of nodes n_1, n_2 is s.
- In a well-balanced network, we expect the graph to look much like a strict grid graph. Here $hops_{max}(n_1, n_2)$ for any pair of nodes n_1, n_2 should be close to $\sqrt{s}$.

While the supernodes theoretically may form a line, it is not likely to happen in reality, except when s is really small. Thus one would expect the average graph to have $\sqrt{s} < hops_{max}(n_1, n_2) << s$, which is clearly very bad compared to the $logn$ hops provided by DHTs.

We could reduce maximum number of hops significantly by adding long distance connections, much like those used in DHTs. However it turns out that a short path between adjacent nodes is more relevant.

The most interesting numbers then, are the relation between h and the distance d, h/d between two nodes, and the average hops between a node and the x nodes

closest to it. If both these number are small and almost constant no matter how big n is, the graph is performing well in terms of latency.

3.2.3.2 Bandwidth

As discussed earlier bandwidth is not normally an issue on the client side of online games. But it is in fact bandwidth which puts some restriction on how we can build the peer-to-peer network. With unlimited bandwidth we might have all peers communicating directly with each other, reducing latency to a minimum. The trade-off is that by adding more connections we may reduce latency, but at the cost of bandwidth.

In general we are not concerned with the bandwidth of the regular nodes, because the amount of messages they receive easily can be regulated by their supernode. Also there is no reason to believe that they receive more information than clients in regular client-server games, as they should mostly have one active connection.

Our main concern lies with the supernodes, which have to maintain many connections at the same time, and send and receive information over these — much like regular servers.

Imagine a network with 100 nodes and four interconnected supernodes each managing 25 nodes. All nodes are sending out a package[1] 20 times a second, each containing a position message. They also receive 20 packages per second, containing 100 position messages each, summing up to 2,000 position messages per second. If a position message consists of 128 bits (4×32 bit, four integers), the position messages alone use 256 Kb/s download, while the upload is clearly insignificant. Almost any broadband connection can handle this amount of traffic.

For the four supernodes it looks very different though. They each receive $25 \cdot 20 = 500$ packages and messages from their children per second. But they also receive at least three packages with 25 messages each from their neighbors. In fact they might receive the same information about all other 75 nodes, from all neighbors, summing to $225 \cdot 20 = 4,500$ messages per second. This is because the neighbors do not know in advance if the given supernode has already received the messages. Thus the supernode might receive 5,000 messages per second, using 640 Kb/s download. The biggest problem is the upload. Here the supernode might send

[1] A package is a collection of messages. Banding messages together like this helps reduce overhead.

as much as $2,000 \cdot 25 + 4,500 \cdot 3 = 63,500$ messages per second, or 8.13 Mb/s, which is a lot more than most broadband connections can handle.

The numbers used above are all fictive, and some messages might not have to be sent, but still it shows a critical issue which needs to be handled. The supernodes cannot handle sending large amounts of information to many children. Either less messages will have to be sent or supernodes need to have fewer children (which effectively will increase hops and thereby latency).

A way to reduce the amount of messages is to enforce a limit on how far some messages travel, since the information might at some point become irrelevant, due to the graphical nature of the game[1]. Also providing players with unnecessary information might degrade security if the user is clever enough to look at the messages being received. We enforce this limit by introducing the *area of interest*[2] for a node, which is the area covered by a circle with the node as center and a given value as radius. The supernodes can use this to filter messages to nodes, simply by calculating the distance between the original sender and the node they are filtering for.

To filter messages being forwarded to a supernode is somewhat harder, since the supernode sending the messages does not know if the message might be relevant to the other supernode's children. We have to figure out if the supernode might have a child node interested in the message.

We do this by introducing an *area of relevance*. As opposed to the area of interest, which is bound to a node, the area of relevance is bound to a message and calculated when the message is sent from its origin.

$$\text{area of relevance} = 2 \cdot dist(o, s_o) + \text{area of interest},$$

where o is the original sender and s_o is the supernode of the origin. We now know that supernodes for which $dist(s_o, s) > \text{area of relevance}$ have no interest in the message. If a supernode, s, had a node, p, within o's area of interest, but where $dist(s_o, s) > \text{area of relevance}$, then $dist(p, s) > dist(p, s_o)$ and p would pick s_o as its supernode instead of s.

While the area of relevance helps us filtering the messages, it is clear that supernodes will still receive some irrelevant information. We see that bandwidth usage can be reduced further by keeping the distance between nodes and their

[1] Most games limit the area of the game world a player can see in some way. This is called the view distance.

[2] The term "area of interest" may refer either to the area surrounding a player, which the player should receive information about, or the radius of the circle, which encapsulates this area, in which case it is simple a number.

supernodes short, thus reducing area of relevance. The problem of nodes gathering is not really solved here either, but we will look more into handling this in section 3.3.5.

Another way to reduce the amount of messages sent by supernodes is to reduce frequency of updates. This could be done dynamically with respect to distance, such that nodes further apart receives less frequent updates about each other.

Due to the loose structure networks may also be constructed such that, if there are s supernodes, then $s - 1$ of them wants the same neighbor, resulting in this supernode having $s - 1$ active connections (see figure 3.11). This is clearly not sustainable and therefore a supernode has to be able to drop connections to neighbors, as long as it retains at least one neighbor in each coordinate quadrant.

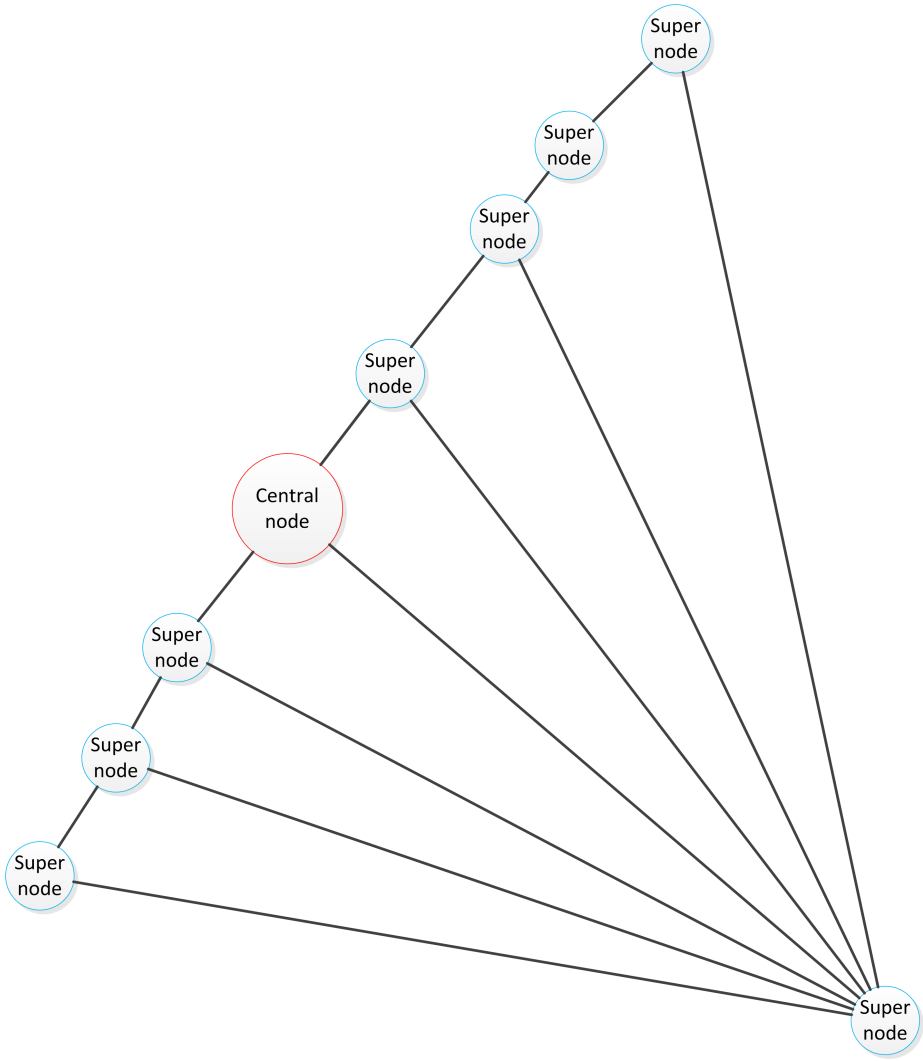

Figure 3.11 – Dynamic grid where one node has many neighbors.

3.3 Implementation

The level of detail in which the solution has been described so far leaves many questions unanswered about actual implementation. In this section we will we will go through the most important issues and choices concerning implementation, and how these are affected by how the developer wants his game to

work.

3.3.1 Central Node

As mentioned in section 3.2.1 there are several ways to implement the central node proposed in our solution. How we choose to implement it should reflect how we want the game to work.

If we want regular players to be able to host the central node, the central node software has to be implemented with regard to this. It has to initiate the game-world and act as the first supernode. It has to work as entry point, providing access and network information to new nodes. If we want a persistent world, it has to provide persistence, which also means that the other supernodes have to inform it about persistent changes in the world. Lastly, it might also have to act as final authority in disputes between supernodes, thus providing consistence and security. Of course, there is a limit to how big a population a regular computer can handle as entry point and storage.

There would also have to be some system to enable players to find active central nodes, so that they may enter the network. This could be accomplished by a lobby mechanism seen in many multiplayer games today: The node would connect to a central, well-known, list server and announce its availability. Connecting players then retrieve a list from here before choosing a central node. Essentially we introduce one more layer of nodes to the system here, but running a list server is unarguably very simple and light weight.

On the other hand the developer might want to be the only one hosting the central node, much like the developers host most of the MMOG servers today. In this way they can manage accounts, charge subscription fees and provide security. It is very likely that a commercially hosted central node would be split up into one or more servers providing entry points, data persistence, authority and acting as basic supernodes. This seems almost like a regular MMOG server, but it is important to remember that the vast majority of supernodes are hosted by peers. By hosting the central node themselves, developers are able to obtain much of the same control as with regular MMOG servers, but use only a fraction of the bandwidth and computing power used by regular servers.

There is a choice between having peers hosting smaller (but still massively populated) worlds or having commercially hosted and bigger world.

3.3.2 Supernodes

While we have already described the basic concepts of how supernodes work and how they find neighbors, some questions remain. How are supernodes placed such that nodes are well-distributed among them and the area of relevance is kept low? How do we keep the number of neighbors low for supernodes, without breaking the graph or increasing latency? When should a node change its supernode? And when should supernodes be closed?

The location of supernodes is important, both in terms of performance and load balance. Unfortunately, we have not found any particularly good way of choosing the location, yet. In our simulation (section 4) a supernode in need of help chooses the location of the new supernode as the average of its children's positions, but this seems far from optimal in many cases. A better choice would be the center of a circle which covers the area most densely populated by the supernode's children. One might also want to pick a location close to the child furthest away from the supernode, in order to shorten the distance between node and supernode and thereby reducing area of relevance.

As we saw earlier the basic concept of our solution allows for many supernodes to pick the same neighbor. In order to ease the load on the given supernode it should be allowed to reject or drop some of its neighbors. The graph remains intact as long as it keeps the closest neighbor in each coordinate quadrant, but keeping only one might increase hops to some relevant supernodes drastically. We therefore propose that supernodes should be allowed to have more than one neighbor in each quadrant, and only start dropping them when an upper threshold is reached.

As nodes move they should change supernodes — but as mentioned it might not be feasible for them to change if the node is still very close to the old supernode and thus is likely to change back. We do not want a node to change back and forth between supernodes too often, as this would create a lot of overhead. Therefore it should be considered not to have nodes changing supernode before the distance between the node and supernode exceeds some threshold. Exactly how far this is depends on the game, but in order for message routing to work the supernode of a node has either to be within the node's area of interest, or to be the supernode closest to the node.

Having too many supernodes will increase the average number of hops between nodes, therefore we should close supernodes that are no longer needed. This can be done by the nodes leaving if they have only few children. How many "few" is should be relative to how many children a supernode must have before it calls for assistance, and it would be wise to consider whether shutting down

would cause a new supernode to be created immediately.

3.3.3 Communication

We propose that nodes communicate with messages of different types, such as position, arrival and leaving messages. The types of messages and their nature depends on the game, but there are at least three different classes of messages. Furthermore one needs to choose which underlying protocol to use for communication.

The three classes of messages are:

- State messages, such as position updates, usually account for the vast majority of messages sent (over 99% [11]). The characteristics are that they are sent often and contain information about something specific to the given point in time. Thus we are only interested in the most recent message and it does not matter if a message is lost. This makes state messages very suited for the UDP protocol.
- Local event messages are triggered by events such as player attacks and other interactions. It may also be non game-related events, such as when nodes leave or arrive. What these messages have in common is they are important and the order of the event execution might also be important. These types of messages are therefore suited to be send using the TCP protocol.
- Global event messages in our solution are messages such as supernode arrival messages. They are much like the local event messages, but have to reach all nodes in the network. However, the order of global events are generally not as important as the order of local events, such as player interaction events.

While the two first classes are fitted for two different protocols they are both locally limited in our dynamic grid, while global events flood the network in an inefficient way. The speed of the flooding can be aided by long distance connects, while another overlay (such as DHT) needs to be used to decrease the bandwidth use. However, since the number of global events is *very* limited, we do not consider the flooding a problem.

For convenience we would suggest that an implementation of our solution should use the TCP protocol, since it fits well with the way the nodes and supernodes

are *connected*, and provides the properties required by the event messages by nature. Alternatively, if this proves not to deliver sufficient performance, we could use UDP with a custom delivery-guarantee layer for important messages.

3.3.4 Consistency

In order to provide consistency we propose a solution using time-stamps, or ticks. All nodes and supernodes have a tick counter, which they keep synchronized with the central node's tick counter. We will not go into details as to how this can be done, but it allows nodes to attach the time, or tick, to event messages, describing when the action was taken. In this way all nodes and supernodes know exactly in which order events occur and should therefore all end in the same state when messages have been executed.

If disputes occur between a node and supernode, the supernode is assumed to be right. If there is inconsistencies between the states of two supernodes, there are several ways to handle this. The central node and another supernode can act as arbiter, or we can make a decision based on the IDs of the supernodes (as a way to avoid undefined behavior or a complex random decision system).

3.3.5 Parameters

There are several ways to tweak performance and other properties of the solution. Some of the parameters may even be dynamically adjusted to provide the best experience for the players in the given circumstances.

3.3.5.1 Area of Interest

The area of interest parameter is very important in order to avoid flooding the network with irrelevant information. The basic rule about the area of interest is that it should never be larger than the area graphically visible to the player in-game. As mentioned earlier the goal is to provide the player with an absolute minimum of information.

The parameter *greatly* influences the bandwidth required by nodes and especially supernodes. This is due to the fact that if n is the number of players who need to share information, then the amount of messages which needs to be sent to

the nodes is n^2. We presume this is why even today's commercial servers have problems when players gather up in close proximity of each other.

We can use the area of interest parameter to prevent n from growing too large, thereby preventing overload of the supernodes. At the same time we are also able to increase the area of interest in sparsely populated areas, allowing the players to see others further away, thereby granting the world more detail. The area of interest becoming very small in densely populated areas should not influence the players much, as the large amount of nearby players would make them less likely to notice the lack of detail in the distance.

By having the area of interest adjusted dynamically, we should be able to provide both better performance and better game experiences.

3.3.5.2 Update Frequency

Having a high update frequency on for example position messages is very important to make the player movement seem fluent to other players. However, as we saw earlier, the high frequency means that exactly this type of messages may use over 99% of the bandwidth used by the game. We have already seen how we can use the area of interest to reduce the bandwidth use, at the cost of in-game detail.

In order to decrease bandwidth use further, possibly allowing for a larger area of interest, we also propose that the update frequency is lowered, as the distance between players grow. This can simply be done by having the supernodes throw away a fraction of the position updates rather than forwarding them to their children. If the fraction is based on the distance between the two nodes, the players will most likely not notice it, since the frequency will only be reduced for players far apart. We believe that this strategy could greatly reduce the amount of messages being sent by the supernodes, which we proved earlier to be a core issue.

Apart from this, one could of course use traditional methods such as dead reckoning to interpolate when in lack of position messages.

3.4 Conclusion

In this chapter we discussed various ideas we have had for how to effectively construct a peer-to-peer graph for a massively multiplayer online game. It was apparent from the beginning that having supernodes — nodes with more control and responsibility — would be useful. The problem is then how to connect these with each other and with normal nodes. We saw that while tree graphs and grid graphs are possible solutions, they have problems with either too many hops, flooding or viability.

Instead we proposed the dynamic grid graph. In this graph, supernodes are placed dynamically where and when they are needed. We set up rules to ensure that the graph is always connected, and described how the graph would behave under various node behaviors. After discussing these theoretical aspects, we turned to the practical side of things: how the various nodes could be implemented, and different classes of messages. Some messages are vital for correct synchronization of the worldstate across nodes. Other messages, like the position of a player, can and should be dropped if it would otherwise arrive too late.

While our solution guarantees a low number of hops between closely located nodes (in terms of in-game position) it has potential bandwidth problems. We saw how this can be combated by introducing an *area of interest*, describing the in-game area that nodes have interest in, and an *area of relevance* which is a way for supernodes to predict when to forward messages to other supernodes.

Chapter 4

Simulation of a Dynamic Graph

We implemented a simulation application in Java to test of our ideas. It is very limited, and simulates the network on a single computer. Originally we intended to do a larger scale simulation across an actual physical network, but due to time constraints we had to leave that for a later date.

None-the-less we feel that if we had not implemented this simulation, we might have missed important factors in our design. Furthermore it allowed us to collect data on various aspects of the performance of our system, as we will see in section 4.2.

4.1 Implementation

As mentioned above, the simulation runs on a single machine as a single application. To simulate nodes being separate, each runs as its own thread. Nodes communicate through a router class, whose purpose is to simulate an underlying network interface: nodes can send a message to another node, and will also receive incoming messages through this class. The router class can also be

used to introduce message delivery delay (to simulate latency), however at the moment it is always 40 milliseconds.

In order to collect data points we have implemented a class that hooks into various parts of the program, to collect information about message hop counts, messages sent/received, timings, distances between communicating nodes, and so forth. Aside from that, the implementation contains different classes to represent the three types of nodes (normal, super and central) as well as messages and message packages (a collection of messages). Messages are then exchanged in packages between nodes using the router.

The application contains a few tools to test various graph conditions. It is possible to add a number of randomly placed nodes to the network, and also to add nodes manually by clicking on the desired position. Nodes can be moved and removed manually, with some limitations: Supernodes can only be removed (not moved) and the central node can neither be removed nor moved. It is also possible to make all normal nodes move towards a random position.

The simulation consists of nodes behaving according to our solution in a given situation. This includes arranging themselves with supernodes and exchanging position updates and other necessary messages. The seed to the pseudo-random number generator is the same at every application launch, providing us with reproducible test scenarios.

A problem that we encountered with our threaded approach, is that obviously simulating each node as a separate thread quickly becomes taxing for the host machine. We found that, on our machines, the limit is approximately 350 simultaneous nodes, sending 20 position messages per second each — but this will of course vary from machine to machine. Threading also forced us to use synchronization in certain places to avoid race conditions.

While this simulation is obviously not the same as a full-scale simulation across a real network, it still provided some interesting insights into the benefits of our solution which we will look into next.

In this thesis we will not go into further details about the implementation of the simulation, as this is rather trivial and unrelated to the core issue.

4.2 Results

4.2.1 Message hops and node distances

One of the main goals of our solution is to keep the number of hops low for nodes close to each other. To measure this, we rigged our simulation software to record the number of hops which a message travels and the distance between the communicating nodes in question. The average of these measurements can be seen in figure 4.1.

There is an obvious difference in the graph before and after a distance of 500 between communicating nodes. This is due to the fact that, when these tests were run, the area of interests for nodes were set to 500[1]. Inside the area of interests we see that nodes very close have a hop count of 2, meaning they are connected to the same super node. As nodes come further apart, the average number of hops slowly rises to 4 — signifying an average of 3 supernodes between nodes at the edge of the area of interest.

When the distance exceeds the area of interest, we see a sudden change in the spread of the measurements. Only supernodes receive messages that originate from further away than the this area, which explains the larger spread: Supernodes will be forwarded a message if it is within their area of relevance, which is

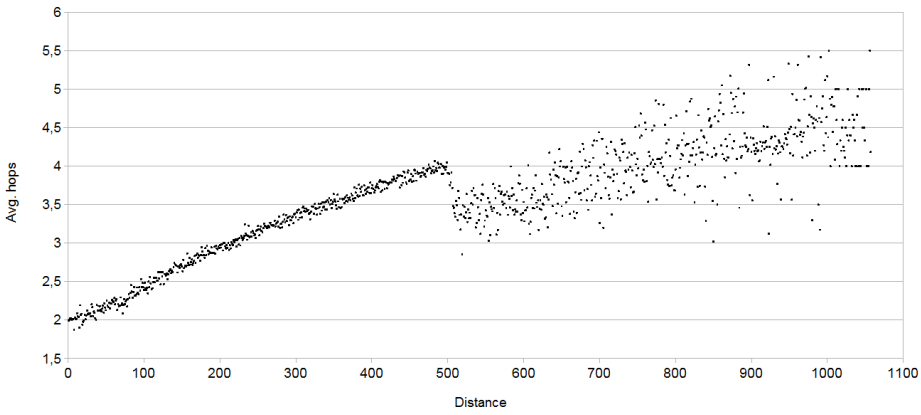

Figure 4.1 – Average number of hops as a function of node distance. Conditions: 157 nodes moved randomly twice.

[1]There is no significance to this number, it is merely for testing purposes. The entire simulation area stretches from -500 to 500 on both axes.

much larger.

Since no normal nodes receive messages from outside the interest area, there is a sudden drop in the average number of hops messages take, as seen on the plot. This is because, since the message terminates at a supernode, it will do one hop less than if it was passed on to a supernodes child.

Another number that is interesting to look at, is how many hops a message takes in relation to how far away the receiving nodes are. This is plotted in figure 4.2. What we see is that as the number of nodes rises, the number of hops messages take increases slowly in comparison to the distance. However, it is difficult to conclude anything from this, since as more nodes enter the world, comparatively more will be far away from any given node, thus increasing the average number of hops messages take. It is obvious from the plot that h increases faster than d, which makes sense, since there is a limit on how many children a supernode can have, thus increasing the number of hops between nodes that are close. A trend line for these data points would seem to be linear, but we actually expect it to grow slower than linearly, especially if long distance links exist. It can be assumed, then, that the number of hops will not increase significantly as more nodes are added.

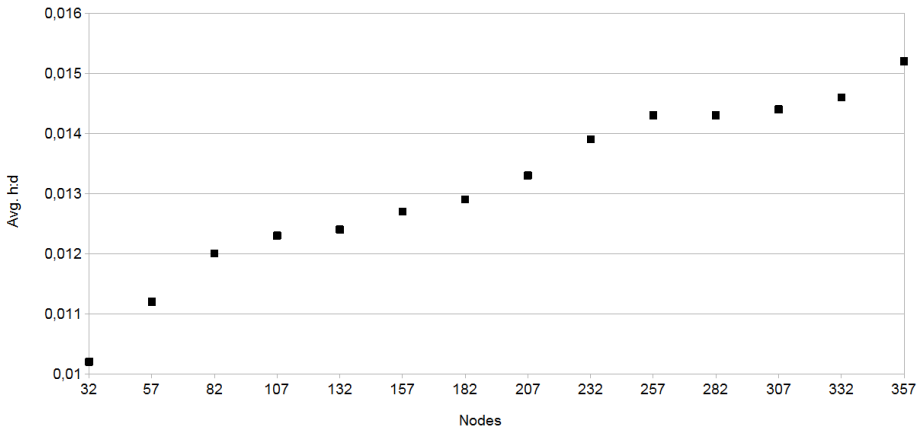

Figure 4.2 – The average of $\frac{h}{d}$ as a function of the number of nodes. Here h is the number of hops a message travels and d is the distance between sender and receiver of that message.

4.2.2 Number of nodes, hops, and supernodes

Figure 4.3 shows how the average number of hops a message takes between nodes rises as more nodes are added to the world. In the very beginning this number is low, because most of the nodes within each other's interest area are connected to the same supernode.

The average number of hops messages take rises as nodes are added, because nodes now might be close enough to each other to be of interest, yet have more supernodes between them. As the number of nodes is increased, this average will increase because of the limit on supernode children.

The average number of hops messages take then seems to level out just above 3.5. This is interesting as it shows that messages do not seem to travel very far, even with a lot of nodes in the network. Not having messages travel far means less bandwidth used, which obviously is beneficial.

Another aspect is how the number of supernodes rises in relation to the number of nodes in the graph. This can be seen in figure 4.4.

However, this is not really too interesting, because it is merely a function of how many children a supernode is allowed to have, and how the nodes are placed in the world. As it can be seen in the figure, the curve is somewhat unsteady — probably due to the distribution of positions generated by the random number

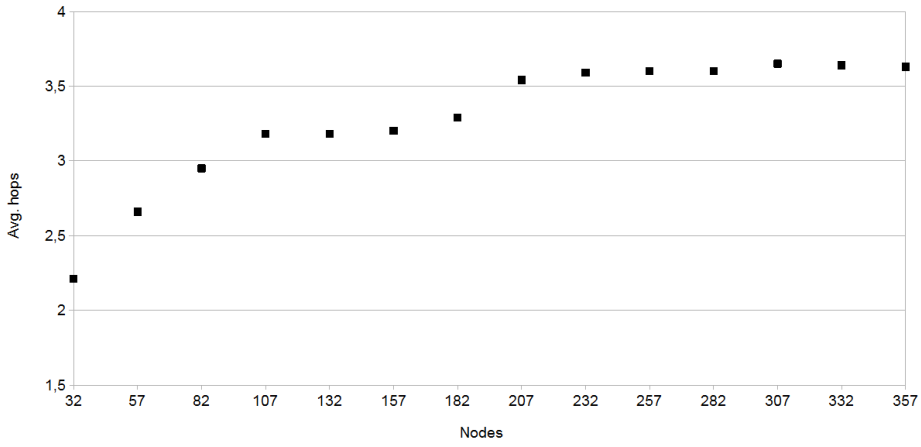

Figure 4.3 – The average number of hops messages travel as a function of the number of nodes.

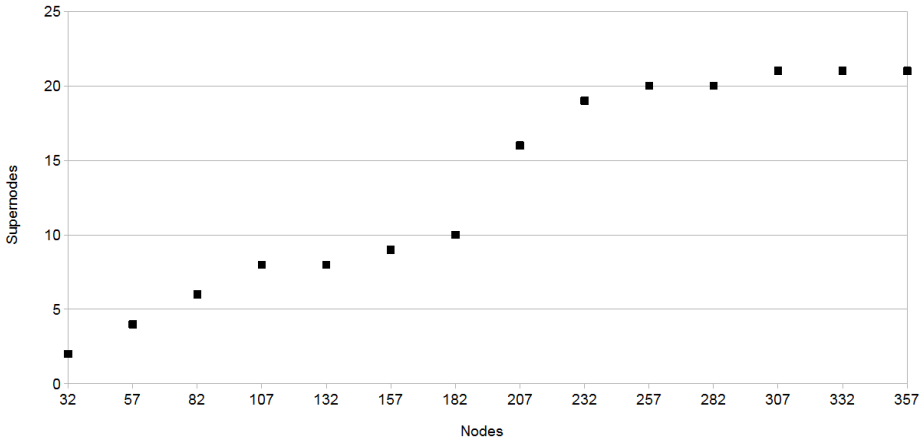

Figure 4.4 – Number of supernodes as a function of the number of nodes.

generator: Sometimes this may prevent supernode creation by producing nodes that are assigned lightly loaded supernodes. At other times it may provoke faster supernode creation by overloading one area in the gameworld.

4.3 Conclusion

In this chapter we have evaluated the data collected from our simulation application. Even though our simulation does not run over an actual network, it still proves an important point. It appears constructing and maintaining the peer-to-peer graph the way we have devised does indeed keep the number of hops between nodes low.

Our solution is not necessarily better for nodes far apart, but this is not important in the case of online games. Important, and true for the dynamic graph, is the fact that nodes close to each other have a low number of hops between them — and that this number does not grow significantly as the number of nodes in the network increases.

Chapter 5

Conclusion

There are already a number of proposals as how to implement a massively multiplayer online game using peer-to-peer methods. Most of the proposals seem to be in early stages of developments and are far from ready to be used in a real game. When we started this project, we expected to find ready-to-use concepts and wanted to test how these performed. Instead we could conclude from the authors' own words that the solutions are not yet ready to be implemented in real games, and therefore we chose to focus more on developing our own solution.

We have looked at the requirements and available technologies for such a solution. We found that players are sensitive to latency in online games, and may become frustrated if it is too high. As such, a low latency is an important goal. We investigated how we might achieve that by looking at different ways to design a peer-to-peer graph. It quickly became apparent that a strictly peers-only graph was not feasible, and we thus introduced the concept of supernodes — nodes with more control and importance. Furthermore we introduced the central node as a governing body and possible entry point into the network.

After looking at various solutions that first came into mind, such as a static grid or tree graph, we decided on a slightly different solution; the dynamic grid. In the dynamic grid there is no pre-defined pattern of supernode arrangement. They are not placed in geographically static positions, nor are they arranged in a tree structure. Instead they exist in a flat side-by-side arrangement wherever

and when they are needed. Nodes representing actual players will then connect to the nearest supernode. If a supernode becomes overloaded, as defined by a threshold, a new supernode will be created in a suitable position. In this way we guarantee low latency (or at least few hops) between players in close proximity of each other in the game world.

At first we wanted to implement a simulation across the actual internet, using the ideas we had developed. However, at a point we realized that this was not possible within our time-frame, and so we chose to develop a simpler simulation instead. This simulation runs on a single machine, where each node runs in its own thread. This simulation did two things for us: It provided us with data about how our solution performs, and it helped us discover problems we might otherwise not have seen.

The data collected from the simulation supports our conclusion that we do indeed provide a low number of hops between closely located players.

We believe that we have found a solution that provides a much more feasible way of constructing a peer-to-peer graph for massively multiplayer online games, than the solutions proposed earlier. Although it is not strictly peer-to-peer, because some peers are more empowered than others, this is a necessary trade-off in order to overcome the problem of not using a central data center. Our solution should make it possible to implement a large scale game without the huge financial backing that is required in traditional client-server implementations.

References

[1] A. R. Bharambe, M. Agrawal, and S. Seshan, *Mercury: Supporting scalable multi-attribute range queries*, in SIGCOMM '04, 2004, pp. 353–366.

[2] O. D. Billing and J. Petersen, *Aspects of peer to peer game networks*. Master's Thesis, DIKU Copenhagen University, 2006.

[3] Blizzard, *World of warcraft subscriber base reaches 12 million worldwide*. http://us.blizzard.com/en-us/company/press/pressreleases.html?101007, Oct. 2010. [Online; accessed 04-June-2011].

[4] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, *Scribe: A large-scale and decentralized application-level multicast infrastructure*, IEEE Journal on Selected Areas in Communications (JSAC), 20 (2002), pp. 100–110.

[5] L. Chan, J. Yong, J. Bai, B. Leong, and R. Tan, *Hydra: A massively-multiplayer peer-to-peer architecture for the game developer*, in Proceedings of NetGames 07, 2007.

[6] M. Claypool and K. Claypool, *Latency can kill: Precision and deadline in online games abstract*, in MMSys'10, Feb. 2010.

[7] A. Cockcroft, *Tips for tcp/ip monitoring and tuning to make your network sing*. http://sunsite.uakom.sk/sunworldonline/swol-12-1996/swol-12-perf.html, Dec. 1996. [Online; accessed 04-June-2011].

[8] B. Hack, M. Morhaime, J.-F. Grollemund, and N. Bradford, *Introduction to vivendi games*. http://www.sec.gov/Archives/edgar/data/1127055/000095012306007628/y22210exv99w1.htm, June 2006. [Online; accessed 06-June-2011].

[9] T. Hampel, T. Bopp, and R. Hinn, *A peer-to-peer architecture for massive multiplayer online games*, in Proceedings of NetGames 06, 2006, p. 48.

[10] D. Jame, G. Walton, B. Robbins, E. Dunin, et al., *Persistent worlds whitepaper*. http://www.igda.org/online/IGDA_PSW_Whitepaper_2004.pdf pages 49–57, 2004.

[11] B. Knutsson, H. Lu, W. Xu, and B. Hopkins, *Peer-to-peer support for massively multiplayer games*, in IEEE INFOCOM 2004, 2004.

[12] Leatrix, *Leatrix latency fix*. http://www.wowinterface.com/downloads/info13581-LeatrixLatencyFix.html. [Online; accessed 08-June-2011].

[13] R. Miller, *Wow's back end: 10 data centers, 75,000 cores*. http://www.datacenterknowledge.com/archives/2009/11/25/wows-back-end-10-data-centers-75000-cores/, Nov. 2009. [Online; accessed 15-June-2011].

[14] P. Parkes, *Skype downtime today*. http://blogs.skype.com/en/2010/12/skype_downtime_today.html, Dec. 2010. [Online; accessed 12-June-2011].

[15] J. Postel, *User Datagram Protocol*. RFC 768 (Standard), Aug. 1980.

[16] J. Tyson, *How network address translation works — how stuff works*. http://www.howstuffworks.com/nat.htm. [Online; accessed 05-June-2011].

[17] B. Whiley, *Distributed hash tables, part i*, Linux Journal, 114 (2003). http://www.linuxjournal.com/article/6797 [Online; accessed 05-Juny-2011].

[18] Wikipedia, *Transmission control protocol — Wikipedia, the free encyclopedia*. http://en.wikipedia.org/wiki/Transmission_Control_Protocol, June 2011. [Online; accessed 04-June-2011].

[19] ——, *Universal plug and play — Wikipedia, the free encyclopedia*. http://en.wikipedia.org/wiki/Upnp, June 2011. [Online; accessed 05-June-2011].

[20] T. V. Wilson, *The technology of mmorpgs — how stuff works*. http://electronics.howstuffworks.com/mmorpg6.htm. [Online; accessed 06-June-2011].

[21] WoWWiki, *Realms list*. http://www.wowwiki.com/Realms_list. [Online; accessed 06-June-2011].