UNIVERSITÄT DUISBURG-ESSEN

FAKULTÄT INGENIEURWISSENSCHAFTEN

ABTEILUNG INFORMATIK

FACHGEBIET VERTEILTE SYSTEME

Diplomarbeit

Design and prototypical development of a mechanism for delay-minimization of interacting participants of a distributed virtual environment

Matthias Helling

Matrikelnummer: 2216126

UNIVERSITÄT DUISBURG ESSEN

November 30, 2009

Betreuer:

Prof. Dr.-Ing. Torben Weis

Dipl.-Inform. Sebastian Schuster

Contents

1 Introduction

With the advent of computer games, related to the increasing computational power of personal computers, game developers always strived for better graphical capabilities, more realistic physical calculations and more beliavable artificial intelligence to enhance the immersion of their attainable audience. Over the decades, computer games emerged from easily developed text-based games into three-dimensional virtual worlds requiring the workforce of hundreds of designers and programmers. At the same time, networking capabilities such as bandwidth and speed increased drastically. The combination of the advances both in computational power and networking capabilities have been used by game developers to create a completely new genre of computer games: the *massively multiplayer online games*, or *MMOG*. Games like **Ultima Online**, **Everquest** and **World of Warcraft** have had a tremendous success in the gaming industry. Unlike single-player games, the game is played via a network and by hundreds or thousands of players in parallel. Most importantly, players remain in the same virtual world, are able to collaborate to solve difficult tasks or fight against each other. The architecture of such games nowadays comprise of servers processing all tasks. They have to guarantee authenticity of players and consistency of the game world, as well as exchanging data between players. These tasks are time and bandwidth consuming, so that powerful machines with broadband internet connections are required. This shows, that operating a modern MMOG is an expensive venture with high risks.

To get rid of as many expenditures as possible, the *peers@play* project uses *peer-to-peer* techniques to source the load out of the server to the individual players' computers, the so-called *clients*. Furthermore, all clients need to build and maintain a common network, in which nodes are called *peers*. Peers in this network are not interconnected completely. Instead, they are ordered after a certain structure. This so-called *overlay* enables the peers to communicate with each other and is part of the peers@play project architecture.

The peers@play project is a cooperation of the universities of Duisburg-Essen, Mannheim and Hannover. It deals with massive multi-player virtual environments (MMVEs), which are solely based upon peer-to-peer techniques and architectures. The creation of such an MMVE consists of several tasks, of that one is the topic of this diploma thesis.
Following, the motivation and the task of this thesis will be presented.

1.1 Motivation

Massive multi-player virtual environments became very popular recently. Most current systems of this kind are relying upon a client/server architecture. In MMVEs, many players use the service at the same time, posing a hard-to-fulfil requirement to the operators of these systems. Servers for these games need to have a high availability and need to provide fast access times to thousands of players. Therefore, servers are computationally powerful and thus have high acquisition costs. Moreover, they need permanent maintenance and monitoring, making running costs worth mentioning as well. The motivation of the peers@play project lies in the strive to reduce acquisition and maintenance costs for operators using peer-to-peer techniques and nevertheless to provide a secure and reliable service with maximal availability for virtually any number of users.

In MMVEs, clients need information on other players that are nearby, including their position and the actions they perform. This is required to enable the visualization of changes in the game world. Moreover, it is crucial that this visualization is possible without high latencies. When other players in the surrounding perform an action, the player has to be able to react on them instantly. Assume a player A attacks another player B with its sword. Player B wants to be able to react by raising its shield or evading. This might not be possible, if the delay for the action message is too high. The server will have evaluated the action before player B even realizes the attack.

In client/server architectures, the detection of these players is simple, because the server holds a global view of the game world. To obtain a list of other players in the vicinity, the client only needs to query the server. Moreover, the server monitors all actions performed by players, so that they are forwarded to each interested client. This process is called *proximity management*. Additionally, the server validates all actions to guarantee consistency and prevent cheating. The delay in client/server architectures usually is negligible, because the server is equipped with a maximum of available bandwidth and each client has a direct connection to it.

In peer-to-peer systems peers are distributed among the game world. Furthermore, each peer is only responsible for a share of the game world. Therefore, proximity management is more complex. Most importantly, peer-to-peer architectures are exposed to a higher average delay than client/server architectures: peers have a limited bandwidth that is not a quarter as good as that of servers.

1.2 Problem Definition

The task of this diploma thesis is to design, implement and evaluate a delay-minimization algorithm for proximity management. This system is called *Proximity Manager* and is a part of the world state propagation layer. It is divided into two phases: the initialization and the maintenance. In the initialization, a peer needs to detect all other peers that are close to itself. Then, connections to a subset of these peers need to be established. This helps to minimize the delay.

To guarantee fluent animations of other players and therewith a high immersion, the delay for action messages needs to be minimal. This is guaranteed by the maintenance phase. However, minimizing the delay is not a trivial task, because players move around the game world. In effect, the vicinity of players change frequently. Furthermore, the proximity manager needs to handle *churn*[1] to avoid inconsistencies.

For message exchange, the proximity manager requires a logical network to be able to address each peer in the network: the *overlay*. Ideally, peer-to-peer systems consist of a high number of participants, each having equal rights in the network. To allow an arbitrary number of peers to participate, it is important that the overlay is *scalable*. This means, that peers are structured in a way, so that they can address each other without having to connect to each other. This is done by *routing*. Messages are forwarded by a subset of peers until they reach their destination. The implementation for such an overlay also a task of this thesis.

[1]The churn is the process of joining and leaving of peers over time.

Besides the overlay, the proximity manager depends on a persistent *storage*. To provide load distribution, each peer only holds a share of the total storage. Furthermore, the storage requires to be fault-tolerant: it may not loose data in case of peer failures. The design and implementation of a storage system is part of this thesis, as well.

2 Fundamentals

This chapter introduces fundamentals required for the understanding of this thesis. It begins with giving important facts about common techniques used in massively multi-player virtual environments. Afterwards, peer-to-peer systems are detailed, including a presentation of the peers@play framework and its layers.

2.1 Massively Multi-player Virtual Environments

The latest trend in computer games and social networking is directed towards virtual worlds thousands of participants can interact in through networks. These virtual worlds are called *massively multi-player virtual environments*. There exist different types of MMVEs, dependent on the target audience and the intentions of the developers.

On one hand, there are massively multi-player online games. They target the broadest possible audience to increase the margin. The gaming experience offered by these games is different to usual single-player games; they allow players to collaborate or compete with a high amount of other players. This is realized by the use of the Internet. They can be sub-classified into different genres, two of it being the *MMORPG* and the *MMOFPS*.

The first one offers a combination of a role playing game (RPG) and a massively multi-player environment. The player slips into the role of an avatar living in the virtual world. Avatars can be customized in their appearance and their proficiency. This can e.g. be the style they fight or the weapons and magical spells they are proficient with. Commonly, these properties are associated with a key concept of role playing games: *avatar classes*. An avatar class is usually chosen on avatar creation and the choice has a deep impact on the gaming experience. The avatar can be trained and advanced in its level and proficiency during the game. The most successful member of this genre is **World of Warcraft** [1]. Because of its success, the game is used as a comparator throughout this thesis. In World of Warcraft, the player can choose between ten avatar classes. The **Warrior** is a melee class with high strength and stamina. The **Mage** is a range fighter, attacking its opponents with magical spells. World of Warcraft uses a pay-to-play payment model, meaning that active players are required to pay a fee for each month of playtime. On the other hand, some games are free-to-play except for the starter game-box required to obtain an account. The game **Guild Wars** [2] is a popular representative of this payment model.

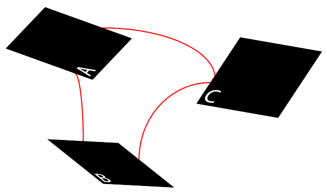
The second genre commonly offers fast-paced fighting in a first-person style. The game type is similar to common first person shooters played via networks. However, in an MMOFPS, it is likely that a lot more players are involved. Note, that this genre is not as successful as its counterpart described above. One of its representatives is the game **Planetside** [3].

On the other hand, there exist other purposes for using MMVEs. One is the creation of a so-called *metaverse*. This can be thought of as a parallel society in which users can interact using their avatars. **Second Life** [4] represents this kind of virtual environments. In Second Life, it is possible to buy plots of land and build own houses or other architectural structures upon it. Virtual game money can be bought with real money. Second life is even used by a number of companies to represent themselves.

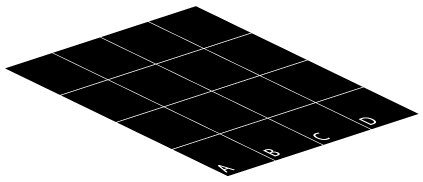
The following sections contain fundamental technical information required for developing MMVEs.

2.1.1 World Type

Game worlds of MMVEs can be of arbitrary size. The game world is partitioned into so-called *countries*. These countries differ in properties like climate, vegetation and population. This type of game world can be observed in **World of Warcraft**. It is possible to move freely across country frontiers without loading times. Such a game world is called a *seamless world*. On the other hand, there exist games that disallow unimpeded travelling across countries. Such worlds are called *chunked* or *zoned* [5] worlds.



(a) Chunked World



(b) Seamless world

Figure 2.1: Comparison of chunked and seamless worlds.

The efficient management of objects in these game worlds is crucial and will be explained in the two following sections.

2.1.2 Chunked Worlds

Using a chunked world means partitioning the complete world into isolated chunks equal to the countries. Furthermore, direct transition between chunks is disallowed. There is no direct geographical relation between chunks and the only way to move to another across each other is by using designated objects, such as a portal. Additionally, direct interaction is only possible with objects and players that reside in the same chunk. Figure 2.1(a) shows this relation in an example with three chunks. The red connections between the chunks visualize a linkage and possible transition path. With chunked worlds it is possible to assign each chunk a server. Servers only hold a share of the game world. Therefore, this approach scales very well and is easy to develop. Furthermore, servers are independent of each other, so that no synchronization is required. Concluding, the chunked worlds approach is very efficient and easy to develop. However, it comes with a major drawback: a loss of immersion. The transition between chunks comes along with a break of visual flow, cause by loading screens. This diminishes the illusion of being in a huge, coherent game world.

2.1.3 Seamless Worlds

The strive towards realism and immersion is ubiquitous. However, this conflicts with performance and management overhead. One possibility to reach this goal is using a *seamless world*. When providing a seamless world, there exist no loading times for transitions across country frontiers. Despite creating a seamless world, the world is partitioned into *zones*. This allows for efficient object management and detection through range queries. See figure 2.1(b) for an illustration for a seamless world with zones.

Discerning from the chunked world representation, smooth transitions are possible across zones. Seamless worlds can be operated using more than a single server equally to the chunked world approach. Therefore, each zone is processed by a dedicated server. However, for a consistent view at zone frontiers these servers need to be synchronized. As an example, players situated on one side of the frontier want to perceive actions performed behind the frontier as well. If they are not able to see behind frontiers, the world type would degenerate to a chunked world. To enable this, servers have a common threshold area in which they permanently need to synchronize the game state. For more information on distribution a seamless world over a number of servers, see [5].

2.1.4 Object Management

Objects exist in every type of game world. Naturally, the number of objects increases with the world or chunk size. And with an increasing number of objects, the object management has to be more efficient. This is emphasized by the frequency in which objects need to be detected and manipulated. The area inside which objects need to be detected is called the *area of interest*, or *AOI* in short. In games, players have a limited viewing range that is mathematically expressed by the AOI. This is particularly important to increase the performance; it is e.g. impracticable to render every object of the world. To decrease the amount of objects rendered, the AOI is used to identify objects that are close to the avatar. In our case, the area of interest is represented by a two-dimensional sphere.

This section outlines a technique to manage an arbitrary amount of objects on an arbitrarily sized world.

```
function DetectObjects(Sphere aoi)
{
    // get a list of intersecting grid squares
    List<GridSquare> squares = GetIntersectingSquares(aoi);

    // gather all objects from these squares if they are inside the aoi
    List<Object> objs = new List<Object>();
    foreach (GridSquare square in squares)
    {
        // loop through all objects
        foreach (Object obj in square.GetObjects())
        {
            // check intersection
            if (aoi.Intersects(obj))
            {
                // add to list
                objs.Add(obj);
            }
        }
    }

    // return the detected objects
    return objs;
}
```

Listing 2.1: Object detection using a geographic grid

There exist two types of objects: *doodads* and *dynamic entities*. Doodads are immutable and static objects that are tied to a fixed position. An example for doodads are houses. All movable or animated objects such as player avatars or non-player avatars are dynamic entities. A common task for the object management of game worlds is a region query; the detection of objects in the area of interest. To handle these queries efficiently, a geographic grid [6] is used. The grid is used as an overlay on the game world. Its cells represent the zones. A game object registers itself into the zones that surrounds his current position. Listing 2.1 shows how to detect a list of objects in the area of interest in pseudo-code. Figure 2.2 illustrates an example for this procedure. Note, that dynamic entities need to unregister themselves from zones they leave.

For optimal performance, it is important to choose an appropriate zone size. If the zones are chosen too small, too many zones need to be traversed for detecting potentially visible objects. On the other hand, if the zones are too big, there are too many objects in each zone. This causes too many intersection calculations.

Note, that the geographic grid presented in this section is used in the implemen-

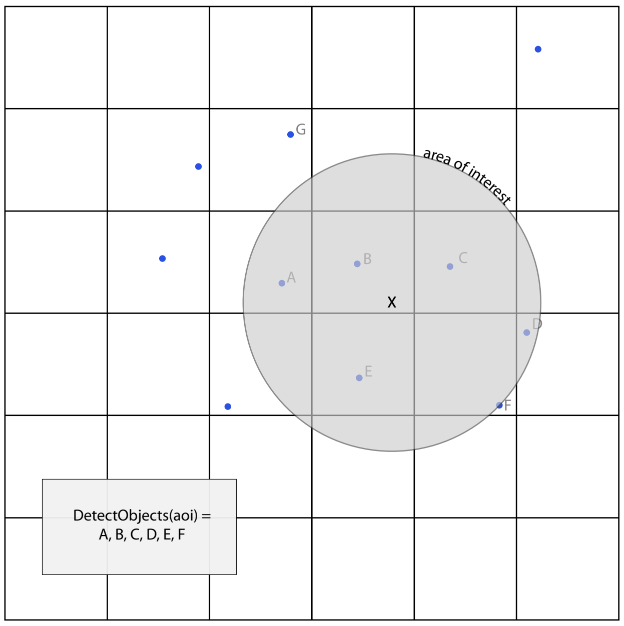


Figure 2.2: Example: Object detection using a geographic grid

tation for this thesis. It provides an efficient avatar detection by storing avatars into the object list of the zone they are situated in.

2.1.5 Instances

Instances are copies of a part of the game world accessed by a portal. Players can enter them alone or in groups without interference of other players. This way, complete campaigns can be played similar to single player games. This technique is used by the majority of MMORPGs, where instances mostly cover dungeons like caves and strongholds. In **World of Warcraft**, there exist instances for up to 40 players. However, the usual number of players is five to ten.

Instancing is not only used for dungeons, but also for countries such as in **Guild Wars**. In **Guild Wars**, the country is accessed by up to five players, similar to the dungeons in **World of Warcraft**. Only towns and cities are open to some extent: players are distributed among so-called *channels*. Each of these channels tolerates a maximum number of players. Whenever the number of player exceeds the maximum, a new channel can be created dynamically. The game **Aion: The Tower of Eternity** [7] uses channelling throughout the game world. This protects the game environment from becoming too crowded commonly perceived as irritating.

Furthermore, channelling is compatible to both chunked and seamless worlds. When combining channelling with chunked worlds, each chunk consists of a number of channels. The chunks are accessible by portals; entering a chunk will automatically join the player into the least crowded channel. Seamless worlds can profit from channelling as well. However, the complete game world is represented by each channel.

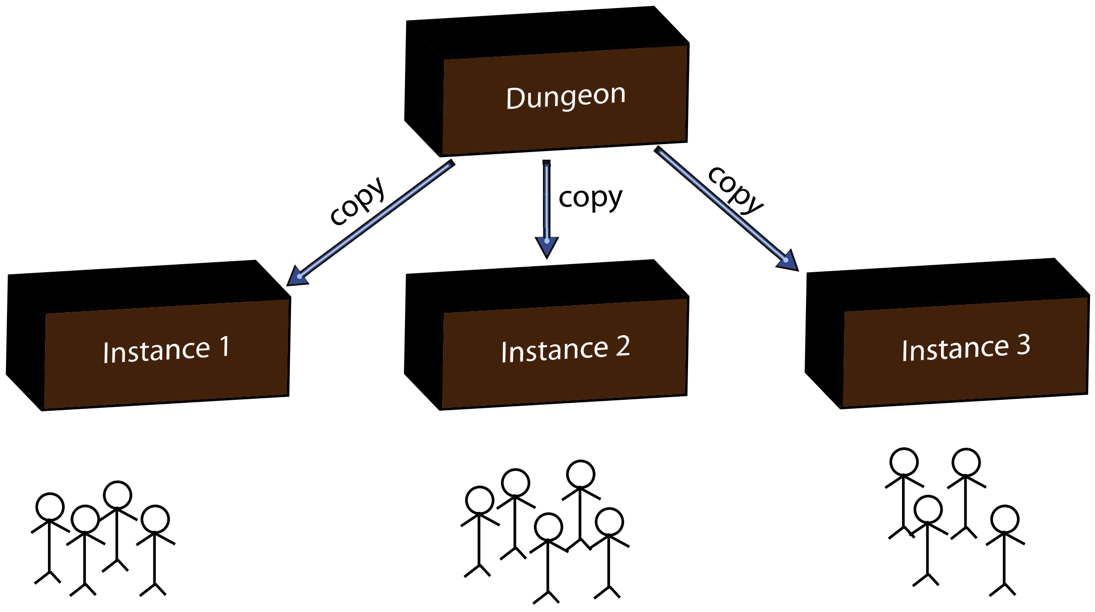


Figure 2.3: Instancing of dungeons

2.1.6 Synchronization

Every time a player performs an action, the following procedure has to be passed through: first, the client informs the server about the performed action. Aided by a rule set provided by the game operator, the server validates this action. Assume, that a player maliciously teleports its avatar to a distant place. If teleportation is not intended by the developers, the game rules will prohibit this action and the server will deny it. Another example for an invalid action would e.g. be an irregular increase of running speed. Especially in player versus player (PVP) environments, such behaviour would lead to an unfair advantage. This is called *cheating*. If the performed action has been valid, the server distributes an action message to a set of players. This set consists of all players that are inside the area of interest. Action messages arrive at interested players with a delay. This delay is introduced by the 2-hop delay to the server and by the validation of actions. As a result, the perception of actions is delayed. Furthermore, the validation of actions takes some time, additionally increasing the delay. Unfortunately, this delay is noticeable by the players.

There exists a practical approach for hiding general latencies in networking games. The key is to only send incremental updates and interpolate in between. As a nice side-effect, this decreases the bandwidth consumption required by the game. Assume, that a player moves along a direction vector v_1. At some time t_2 the player changes course. This induces the game client to issue an update to the server, solely with the information of the current position and the new direction vector v_2. In between these update messages, the position of the avatar is interpolated. This method of positional interpolation is called *dead reckoning* [8]. One crucial assumption required for this technique is, that the speed and the direction between update messages is constant. However, in most MMORPGs the speed is constant by default and can only be changed by an additional action, e.g. *sprint*. These actions are sent to the server anyway, so that this technique is perfectly well applicable to online games. It can furthermore be used to decrease the irritating delay of the game client in that actions become visible after they have been validated by the server. By assuming the validity and interpolating positions, a fair player will not sense any latency between his physical action and the visualization. Note, that dead reckoning cannot hide the 2-hop delay introduced by the server.

The process of dead reckoning is illustrated in figure 2.4. It shows four time-steps with their associated positions and direction vectors. The player changes his course on every shown time-step. This would lead to a total of three messages sent regardless of the elapsed time.

2.2 Peer-To-Peer Networks

A network consists of an arbitrary number of interconnected computers and enables communication between them. In many cases, such networks are driven by two

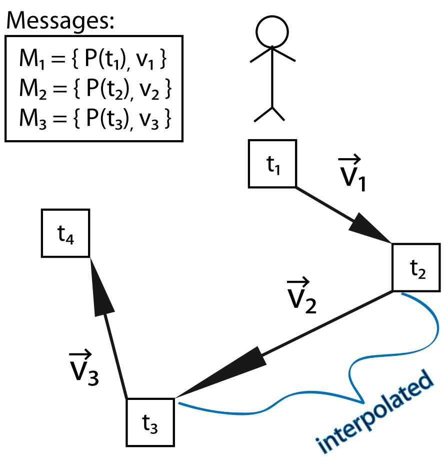

Messages:
$M_1 = \{ P(t_1), v_1 \}$
$M_2 = \{ P(t_2), v_2 \}$
$M_3 = \{ P(t_3), v_3 \}$
t_1
t_2
t_3
t_4
$\vec{v}_1$
$\vec{v}_2$
$\vec{v}_3$
interpolated

Figure 2.4: Dead reckoning in network games

types of participants: *servers* and *clients*. A server is responsible for providing services to other participants of a network. Web-servers for instance, provide access to a subset of web-pages, that are hosted on it. To serve high numbers of clients, a server requires to be computationally powerful and to be equipped with high available network bandwidth. These properties are limited and expensive. On the other hand, clients consume these services. They are not liable to provide any service to any other participant of the network. This network composition is called a *client/server* architecture. In this architecture, every client relies on the server. If the server fails, all clients requiring services from it cannot be served: a server is a *single point of failure*.

As an alternative of using a client/server architecture, it is possible to omit servers in a network. Instead, all clients are liable to provide required services. This network composition is called a *peer-to-peer architecture*. Clients in such networks are called *peers* and have have equal responsibilities and privileges. Furthermore, peers are not expected to remain in the network eternally. They may join and leave it at their will any time. This behaviour is referred to as *churn*. Because there is no server in this architecture, it does not have a single point of failure. If one peer fails, other peers take over its work. Furthermore, peer-to-peer architectures enable the minimisation of acquisition and maintenance costs for network operators. Figure 2.2 illustrates the two different architectures.

However, peer-to-peer systems have well-known drawbacks. Connection establishment, e.g. turns out to be difficult in some cases. To defend themselves from

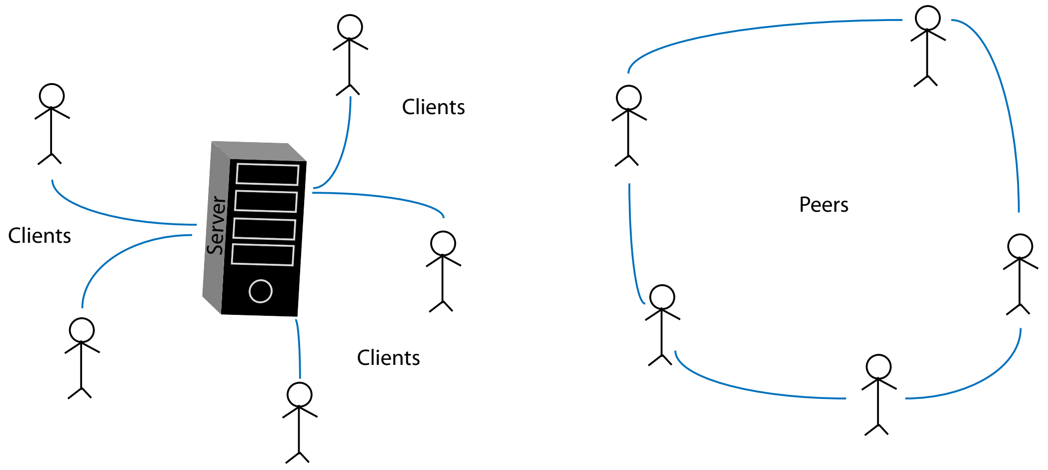


Figure 2.5: Architecture comparison of client/server and peer-to-peer.

intruders and malware, peers are usually situated behind a firewall. Many firewalls disallow a connection establishment request from outside, so peers behind strong firewalls are not accessible. For information on how to trick specific firewall types, see section 2.3.2. Furthermore, peer-to-peer networks lack of a well-known entry point. On start-up, a method is required to find already integrated peers. This method is called **bootstrapping**.

Common tasks of a network are data lookup, the addressing of peers and data exchange. The approach for addressing depends on the chosen network type. There exist two types of peer-to-peer networks: *structured* and *unstructured* networks. In unstructured networks, all participants are arranged without using a certain scheme; they are just mounted into the network at an arbitrary place. Therefore, the only way to address peers is by flooding. When flooding, a peer sends a message to all its connections. This message comprises of the payload and an integer value called *time-to-live*. Upon receipt of a flooding message, each node decrements this value. If it is greater than zero afterwards, the flooding continues recursively. For applications like MMVEs however, this approach is not applicable, because it consumes a lot of bandwidth. Structured peer-to-peer systems offer a more efficient alternative for addressing. They arrange peers after a strict ordering. Each peer is assigned a unique identifier used to define the order. This arrangement allows for logarithmic routing complexity: a message sent to a peer of the system will require up to $log(n)$ hops, where n is the number of peers of the network. Structured peer-to-peer systems are more complex than unstructured ones. They need a lot of computations for joining, leaving and maintenance. However, this cost is amortized by the logarithmic routing complexity. Peer-to-peer networks are also called *overlays*. In the following, we refer to overlays as structured networks. For more information on overlays, see section 2.3.4.

The majority of networking applications need a shared, persistent storage. In client/server architectures, this storage is provided by the server. In peer-to-peer architectures, it needs to be distributed equally among all nodes. The storage builds upon the overlay and shares a key-space with it. The most popular distributed storage systems all fall under the category of *distributed hash-tables* or *DHTs*. For more information on storage systems, see 2.3.5.

2.3 peers@play Architecture

The peers@play framework consists of layers of software, each accessing only neighbouring layers via interfaces. This enables the operator to exchange layer implementations dynamically.

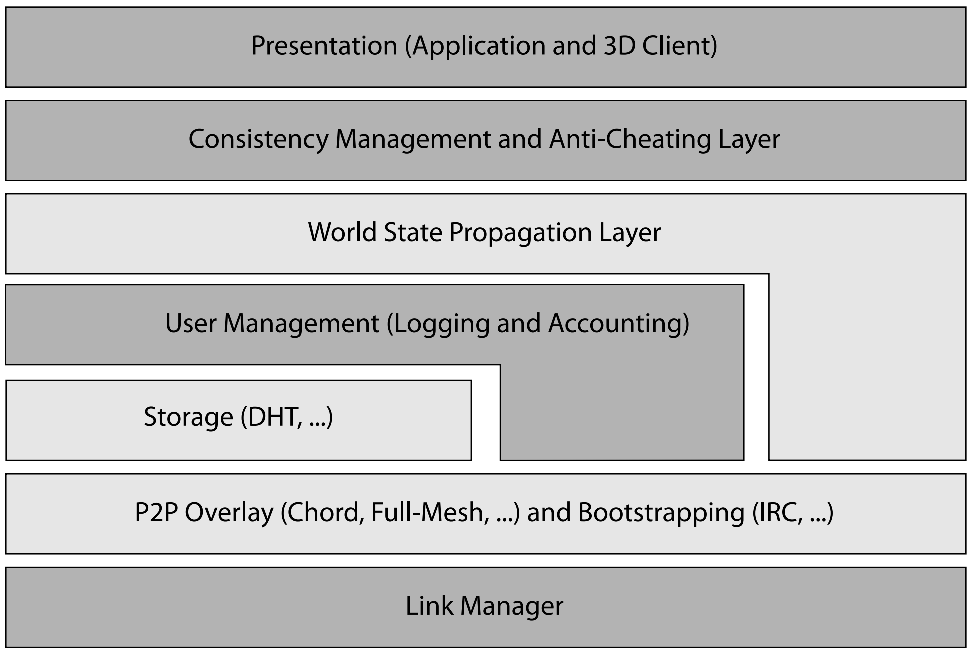

Presentation (Application and 3D Client)
Consistency Management and Anti-Cheating Layer
World State Propagation Layer
User Management (Logging and Accounting)
Storage (DHT, ...)
P2P Overlay (Chord, Full-Mesh, ...) and Bootstrapping (IRC, ...)
Link Manager

Figure 2.6: peers@play layer architecture

Figure 2.6 illustrates the layer architecture used in the peers@play network. The light-grey layer color indicates which layers are covered by this thesis. On the bottom of the architecture resides the **Link Manager**. Its task is to establish and maintain connections required by the upper layers. The current implementation of such a service is called **Secure Network Abstraction Layer** or **SNAL** and will be outlined in section 2.3.2. The next layer consists of the **P2P Overlay** and the **Bootstrapping**, explained in sections 2.3.4 and 2.3.3, respectively. The **Storage** is sitting on top of the overlay and provides a persistent database. Further information on the storage is given in section 2.3.5. The **User Management** sits on top of the storage and the overlay. Note, that there exists no implementation of this layer in the current version of the project. Therefore, we always assume that user credentials are valid. The **WSPL** uses the user management, the storage and the overlay. It will be detailed in section 2.3.6. The two topmost layers **Presentation** and **Consistency Management** are not completely implemented as well and will thus not be discussed in this thesis.

Although the user management is not implemented yet, a user can login into a distinct *world* by passing an arbitrary *user-name* / *password* combination and the identifier for the desired world, the *world-name*. By allowing disjoint worlds, it is possible to operate an arbitrary number of virtual environments using the same framework, in parallel.

Throughout the peers@play project, an asynchronous programming model is incorporated. This programming model is aided by the use of a message passing framework called **gears4net**. The upcoming section will screen its usage to provide an abstract understanding of the algorithms in chapter 4.

2.3.1 Gears4Net

The gears4net software resembles a message passing API, using a special syntax for wait conditions. Systems using gears4net implement a protocol. A protocol uses a scheduler processing all passed messages and consists of handlers responsible for the execution of routines associated with the retrieval of a message. Wait conditions are expressed using the *C# yield-operator*. The yield operator stems from the iterator programming concept. A simple usage of an iterator is the *for-each-loop*. To enable the for-each traversal over a data structure, the definition has to supply the iterator interface. Algorithm 1 shows some simplified, but common statements to clarify the usage of this API.

Algorithm 1 Simplified examples of gears4net wait conditions

```
1: yield return Receive<T>();
2: yield return Timeout(1000);
3: yield return Interval(1000, method);
4: otherProtocol.BroadcastMessage(new T('MyMessage'));
```

Statement *1* waits for the receiving of a message of type *T*. The second one waits for 1000 milliseconds until it is able to continue. The third statement regularly calls *method* in an interval of 1000 milliseconds. Using these statements, it is possible to drastically simplify the program code. Furthermore, while some methods are waiting for input or time-outs, other methods can be processed. Without gears4net several threads run in parallel, rendering the communication and synchronization of these very difficult. In gears4net, the scheduler is run in a single thread, so that synchronization is not needed. The communication between different and inside protocols is done by message passing (see statement *4* as an example on how to pass messages to another protocol). For detailed information on the gears4net message passing API, see [9].

2.3.2 Secure Network Abstraction Layer

The peers@play network contains a layer that handles networking tasks such as connection establishment and maintenance: the **secure network abstraction layer** or **SNAL**. Using the SNAL and the so-called **SNAL-addresses**, upper layers can establish and close connections. Connecting two peers can be difficult depending on their *firewalls* and *NAT-boxes*. Firewalls control the data transfer between clients and drop potentially malicious or unwanted packets for security reasons. Network Address Translation (NAT), is a technique to map IP/port-combinations to clients of the local network. This way, more than one client can access the Internet via one external IP. For each connection, the external port is assigned to an internal IP/port-combination. There exist NAT-boxes that provide a high security, like symmetric firewalls. The downside of these firewalls is that connection establishment from outside the local network is very difficult, if not impossible. The SNAL provides a NAT traversal mechanism, that uses three techniques to establish a connection: *reversal*, *hole punching* and *relaying*. Each technique can be used in a specific situation. Let A and B be peers, where A wants to establish a connection to B.

Situation 1.
Let B be situated behind a symmetric firewall. A does either use no firewall, or a firewall that is easy to bypass. In this situation, it is possible to apply the *reversal* technique by using the overlay. Therefore, A routes a message to B with a connection request. Then, B can establish a connection to A.

Situation 2.
If both peers A and B are behind a symmetric firewall, the only chance to connect is to try *hole punching*. This punches a hole in the clients firewall to find a port that maps to the peers client.

Situation 3.
If connection establishment failed using reversal and/or hole punching, *relaying* is the only possibility left: Therefore, no connection is directly established between A and B. Instead, a third peer to that both A and B are connected to is used for data transaction: the traffic is relayed on it. This way, each peer in the overlay is able to communicate with each other.

For more information on the SNAL and its techniques, refer to [9]

2.3.3 Bootstrapping

The bootstrapping service aids in the joining process of peers. Without it, there would be no way to find a running network from scratch; peer-to-peer systems commonly do not provide a well known entry point.

The following requirements need to be met by such a bootstrapping service:

- **Availability:** A single server like those used in most online games such as **World of Warcraft** is a *single point of failure*. If they suffer from failure, the whole service will be unavailable until it is recovered. We require the bootstrapper to be available at any time without downtimes. Therefore, the traditional server approach cannot be used. Instead, we need a decentralized service to provide fail-safety.

- **Scalability and Efficiency:** The bootstrapper must work with any amount of participants. Because we want our peer-to-peer network to be scalable, the bootstrapper needs to provide scalability as well. Additionally, it requires to be efficient: the computational and bandwidth load may not exceed a small slice for integrated peers to leave enough power for application purposes.

- **Robustness:** The bootstrapper must be robust enough to avoid failures for both peers and the service itself. It may not be disturbed by external interference or security appliances like firewalls.

Within the scope of the peers@play project, a bootstrapper fulfilling these requirements has been presented in [10]. It is the entry point of all peer-to-peer applications in the peers@play framework. It utilizes an arbitrary *IRC network*, known to have a high availability. Because there exist a huge amount of IRC networks in the Internet nowadays, providers can utilize these and don't need to run an own network. It is worth mentioning, that the bootstrapper does only impose a minimal load on the IRC network. Otherwise, the risk of being banned would be too high.

The bootstrapper works the following way: integrated nodes intercept an IRC channel created for their world. Every-time a peer wants to join the network, it joins this channel before-hands. Once inside, it requests information on how to join the overlay by sending a query message into this channel. Interceptors are then replying with a list of online nodes in turn. To obtain scalability, a maximum of C_{bt} interceptors are servicing at the same time.

A major disadvantage of the IRC bootstrapper is that of a lack of security. Because the software uses public IRC networks, malicious peers can easily disturb the bootstrapping process. It is thereby, e.g. possible to create a parallel, malicious network. Security issues have not yet been covered by the authors.

2.3.4 Overlay

Peer-to-peer networks are built on top of the Internet. They enable applications to route messages to logical addresses instead of IP addresses. This logical network is called *Overlay*. Overlay networks consist of an arbitrary number of nodes. These nodes have an overlay address associated with their physical address. A common task for overlays is routing, where messages are forwarded by a subset of nodes until they reach their destination. Overlays have the following requirements:

- **Scalability:** The overlay is a logical network that strives for scalability to allow an arbitrary number of nodes to participate. Therefore, each node may only be responsible for the maintenance of a fraction of the network. This includes established connections; establishing a connection to each node of the overlay creates a *full-mesh*. To avoid too many connections, the majority of overlays provide a logarithmic amount of connections.

- **Self-Organisation:** The overlay is responsible for organising itself automatically. No manual interaction should be required to maintain the network.

- **Fault-Tolerance:** The overlay must be able to cope with node failures and the churn to remain consistent. The routing procedure must operate correctly under all circumstances.

To obtain scalability, modern overlays arrange nodes in a key space. Each node only connects to a subset of other nodes. The routing is achieved by converging at the destination key. The majority of overlays provide a routing complexity of $O(logn)$, where n is the total number of nodes in the network. Examples for popular overlays are **Chord** [11] and **CAN** [12].

2.3.5 Storage

Applications like MMVEs using peer-to-peer techniques require some kind of storage accessible to every peer. In client/server applications, this storage is provided by the server, where the server reflects a single point of truth. Every data set is only managed at a single place to avoid inconsistencies. Each client that wants to retrieve data sets does so by querying the server for it. In peer-to-peer applications, there is no server and hence there exists no single point of truth, incurring the possibility of inconsistencies. The common storage design for peer-to-peer systems is a hash-table[1], whereas all data sets are stored in a distributed manner to allow load balancing. Therewith, each peer of the network only holds a share of the entire data. This data structure is called a **distributed hash-table**, or **DHT**.

Basing on the overlay, it is possible to implement a DHT, that provides access to the functions *store*, *retrieve* and *remove*. However, there exist several requirements that an implementation has to fulfil:

- **Scalability:** To allow access to an arbitrary number of nodes at the same time, the storage implementation needs to be scalable. This means, that the implementation may not exceed linear complexity.

- **Self-Organisation:** Nodes in the network should not be required to manually interact or configure the DHT for the joining and maintenance.

- **Load-Balancing:** The entire data set needs to be distributed equally among all nodes of the network.

[1]Hash-tables are containers that enable to associate an object value using a unique key.

- **Fault-Tolerance:** The DHT must remain consistent regardless of errors or node failures.
- **Robustness:** The DHT must be robust enough to work correctly, even if several nodes try to interfere.

Fault-tolerance is very important to prevent data-loss. Often, this is achieved by *replication*: each data set is held by more than a single node at a time. Every-time a data set is updated or removed, each *replicator* (i.e. the node, that holds a copy of the data set) will be informed by the original owner. There exists a simple rule for the probability of data loss: assuming that the data is replicated on k nodes, then a maximum of k nodes may leave at the same time without a loss of data, because there is still one node left holding the data (the owner plus all replicators make up a total amount of $k + 1$ redundant local storages). Hence, with more replicators, data loss becomes less probable and therewith fault-tolerance increases. Of course, each replicator comes with an overhead both in CPU-usage and in bandwidth consumption. Particularly big data sets consume a lot of bandwidth, aggravated by the number of replicators. Apparently, there is a trade-off between fault-tolerance and efficiency.

The majority of peer-to-peer storage solutions are included in the routing software. For example Chord and CAN have a storage integrated already. However, the peers@play framework separates them into disjoint layers.

2.3.6 WSPL

The overlay and the storage provide functionality to build MMVEs using peer-to-peer techniques. For MMVEs it is crucial to obtain the position of a node in the game world. It is, for instance, required to detect the nodes that need to be rendered to the screen. Furthermore, nearby nodes need to have a consistent view of the game state. Therefore, a high update rate is required with a minimal delay. Update messages e.g. contain current position information about a node. Furthermore, MMVEs need a way to geographically index static and dynamic entities in the game world and dynamic objects, such as *non-player characters* need to be controlled by nodes.

In the peers@play project, the world state propagation layer is responsible for this task. For convenience, it separates the behaviour into two tasks: **proximity management** and **entity management**. Proximity management refers to reducing the bandwidth consumption of the network in total and the delay between geographically close peers in the virtual world. Besides its effect on the consistency, the delay potentially introduces lagging motions of avatars. Further information about the proximity management is given in section 3.3.

The second task of the WSPL is the entity management. In an MMVE, there exist objects and *non-player characters* (*NPCs*) (sometimes called *agents*). Examples for objects are doodads and objects that can be picked up by players like weapons, ammo or health kits. Both the objects and the NPCs are situated at a specific place of the game world. The entity management provides an interface for discovering, changing and maintaining these objects in the world. The computation of NPCs needs to be distributed among all nodes in an efficient manner as well. Note however, that the entity management is not part of this thesis.

3 Design

This chapter gives detailed information about the design of the implemented layers: the **Overlay**, the **Storage** and most importantly the **Proximity Management**.

3.1 Overlay

Structured peer-to-peer systems need a way to address every integrated peer for communication: the overlay. An overlay of a peer-to-peer network arranges participants in a way, so that they are uniquely addressable. Often, this is done by sorting them into a ring, as in **Chord** [11] and **Pastry** [13].

The peers@play project did not offer an overlay at the beginning of this thesis. Because the proximity manager requires one, a simple overlay is implemented: the *full-mesh overlay*. Naturally, a full-mesh is not scalable. However, the implementation of a scalable, sophisticated overlay such as **Chord** is beyond the scope of this thesis. This full-mesh overlay can then be used for simulation purposes and comparisons. To be able to swap the overlay for a scalable one, the full-mesh overlay implements the overlay interface declared by the peers@play project. The key methods of this interface comprise of the methods **Optimize** and **Send**. The optimize method can be used to optimize a connection for either bandwidth throughput or the delay. The send method forwards a message to an arbitrary node of the network.

This section shows how this simple overlay is structured first, then it gives an introspection of the bootstrapping phase and the routing procedure.

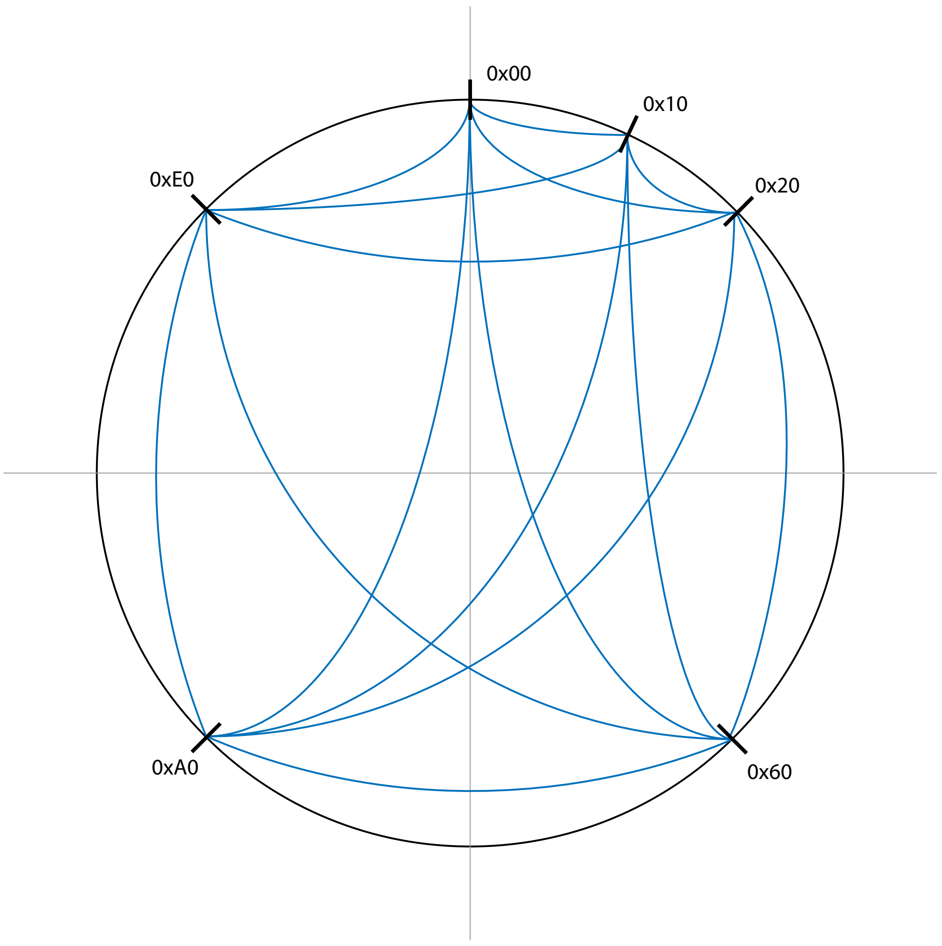


Figure 3.1: Full-Mesh Overlay: Structure

3.1.1 Structure

The structure of the full-mesh overlay resembles that of **Chord**. This means, that the nodes are arranged on a circle. Their IDs are elements of a warped, one-dimensional key space. They are calculated using the *SHA-1 algorithm* [14] that returns a 160-bit number. The ID of a node in the full-mesh overlay is called **OverlayID** in the following. Figure 3.1 illustrates this structure. The blue lines between nodes show their connection.

Despite the full-mesh, this structure has been chosen to enable efficient key space responsibility and homogeneous distribution of nodes. Each node has special neighbours: the *predecessor* and the *successor*. The predecessor is its immediate neighbour with a lower OverlayID. Symmetrically, the successor is the immediate neighbour with a higher OverlayID. The overlay does not need constructs like routing or finger tables. This simplifies the design to a large extend.

3.1.2 Integration

The overlay integration consists of two phases. In the first phase, the bootstrapper is used to find nodes, that are already integrated in the overlay. In peers@play, the **IRC-Bootstrapper** is used for this task. See section 2.3.3 for more information on how this bootstrapper works. There are two possible outcomes of the bootstrapper. If the desired world has been empty before, it returns an empty list. Otherwise, the list contains already integrated nodes. The overlay then connects to all conveyed nodes. To each connected node an integration request message is sent. Upon receipt of such a message, an integrated node replies with the list of all nodes of the network. This list is then used to re-establish the full-mesh.

3.1.3 Routing

Every node is responsible for the key space between itself and its predecessor. This segment is called the *responsibility interval*. Because of the full-mesh, the routing can always be achieved in a single hop. The destination is calculated the following way: Given a destination key k, the message is sent to the node with the closest ID to k. Because routing is only processed forward in the ring, the calculation of the distance is dependent on the order. If $k > n$, it can be calculated using $k - n$. If $k < n$, it is equal to $2^{160} - n + k$. If $k = n$, the node itself is the destination and no messages need to be sent. To send a message, the overlay compares the distance of the ID of each connected peer to k. The node with the closest distance is then chosen as destination.

3.2 Storage

Basing on the overlay, a storage system has to be implemented to provide a persistent database. In this thesis, a distributed hash-table is implemented. This implementation uses the overlay described above and is therefore called **Full-Mesh DHT**.

The peers@play project offers an interface for DHTs, which is incorporated by the Full-Mesh DHT. Distributed hash-tables commonly comprise of the methods **Store**, **Retrieve** and **Remove**. Their functions are self-explanatory. The storage uses the responsibility interval given by the overlay to decide the owner of a data set. This allows for efficient and simple algorithms: messages that are sent to an arbitrary address K will reach the node responsible for K. The data key can be used as address for DHT operations. The responsible node will be found using the responsibility interval. It will afterwards acknowledge the result of an operation.

When more than a single node wants to store data under the key K at the same time, lost updates can occur. If node A and B both store under the same key in parallel, one of the stores is overwritten. This depends on the order their messages arrive at the responsible node. If for instance the store message of node A arrives first, the store is processed and a success message is sent back to A. Afterwards, the store message of B is processed, overwriting the data stored by A. To prevent lost updates, the Full-Mesh DHT incorporates a version control system. Each data set has an associated real version, which needs to be conveyed for store and retrieve operations. Furthermore, each node has a local version of recently used data sets. The local version is included in each operation message. When storing, the local version is compared to the real version. If they are equal, the store can be performed. Otherwise, it is likely that the store overwrites previous changes. Therefore, the store is denied. Afterwards, the changes on the data have to be performed again on the current data. When retrieving, the local version is used to detect whether a node already has the data up-to-date. If so, it is not sent back again to save bandwidth.

To gain protection against data loss, replication is incorporated in the storage. A fixed number of nodes, called replicators, will therefore hold an inactive copy of each data set. These replicators are the immediate successors of the responsible node. The number of replicators is defined at compile-time. The more replicators

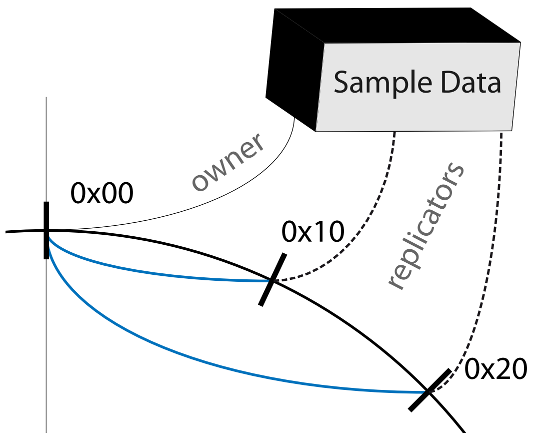


Figure 3.2: Full-Mesh DHT: Replication

there are, the higher is the fault-tolerance. On node failures, the data set can be reconstructed using the replicators. In practice, this procedure is very simple. The successor of the failed node will be responsible for its data. Because it already holds a copy of all this data, this process only involves a local copy from the replication to the responsible data pool. Figure 3.2 illustrates the replication using the full-mesh overlay.

3.3 Proximity Management

This section begins with a requirements analysis and continues with detailing the design decisions considered for the implementation.

3.3.1 Requirements

The requirements analysis helps to find a proper software architecture. The following requirements have been analysed in thesis:

- **Delay-Minimization:** The proximity manager must provide an infrastructure that allows minimal exposure of delay to update messages. These update messages need to be sent to all neighbouring nodes[1] in a timely manner. This shall reduce noticeable delay and lagging artefacts to a maximal extent and thus expose a maximal immersion to the players.

- **Minimal Bandwidth Footprint:** The proximity manager must be efficient in terms of bandwidth consumption. Regarding that hundreds of players might be nearby at the same time, it may never exceed common internet connection capabilities.

- **Self-Organization:** The proximity manager must function without manual configuration or interaction at runtime. This contains both the initialization process and the maintenance.

- **Locality-Awareness:** To reduce the total amount of connections, the proximity manager must be locality aware. In practice, this means that the manager only connects to other nodes in the immediate surrounding in the game world.

- **Scalability:** Scalability is very important in the majority of applications. This enables access to an arbitrary number of users. Therefore, the proximity manager shall be scalable in terms of required connections and bandwidth.

[1]The term neighbouring nodes refers to spatial coherence of players in the virtual game world.

3.3.2 Solution Approach

This section describes the chosen approach for the implemented proximity manager. The storage layer delivers a persistent storage that can be used to aid in this task. Furthermore, the peer-to-peer overlay is used for message transmission.

The implemented proximity manager subdivides the task into three smaller ones. First of all, each node has a position associated to the game world. Additionally, it possesses a bounding circle. This bounding circle comprises of the nodes' position and a radius. This circle resembles the area of interest of a node. The area of interest is used to localize other nodes. With its help, the proximity manager knows which nodes are potentially visible. Because they are visible, it is important to provide high update rates and a minimal delay. Other nodes, that are inside of a nodes' area of interest are called *near nodes*. Further information on the AOI is given in section 3.3.4.

The detection of near nodes does however not suffice to provide consistency at all times. Due to constant movement of nodes, the set of near nodes is expected to change constantly, as well. To detect nodes that are about to enter the AOI, a second threshold area is declared: the *far region*. The peers@play projects aims at operating seamless world MMVEs. Therefore, the world is partitioned into zones of a fixed size and position. The far region complies exactly one of these zones. Consult section 3.3.4 for further information on the far region.

Still, both presented regions do not suffice. The third one is called the *adjacency region* and comprises of all surrounding zones in the eight-neighbour topology. The adjacency region is used to keep track of nodes in the hinterland. This is of importance, because of the constant movement. Assume, that a nodes AOI scratches the frontier to a neighbouring zone. Although we know both the nodes inside the AOI and the nodes inside our zone, popping[2] might occur. Furthermore, if the AOI overlaps with the frontier, it is possible that nodes behind it ought to be seen. This phenomenon is handled by the adjacency region and the **HandleFrontierDiscovery** method. The adjacency region is described in section 3.3.5.

When a node wants to join the world, it first retrieves information about its avatar from the storage. This includes the current position of the avatar. Using this

[2]Popping means that something previously unseen *pops* into being without a smooth gradient. This usually is perceived as irritating.

position, its far zone can be calculated. Afterwards, the node connects to all nodes inside this zone. Following, it connects to a fixed number of nodes of its adjacent zones.

The following sections further describe the world partitioning and the accosted regions and their discovery in detail. The chapter ends with a summary on the covered design issues.

3.3.3 World Partitioning

The discovery of nodes in a given area of the game world can be done using the storage layer. The game world is supposed to be seamless. Therefore, the game world is partitioned into zones. Each zone is then represented by a single entry in the storage. Figure 3.3 illustrates this process with a sample game world consisting of 16 zones. This entry consists of the zone coordinates and a list of potentially contained nodes. When a node joins a zone, it writes its overlay-address into this list. Upon leaving the zone, it unregisters itself in turn. Due to node failures and delays, we cannot be sure whether each entry of this list is valid. As a result of a failure, a node is not able to unregister himself from the list. To invalidate such failed nodes, the current time-stamp is added to the list upon registration. After fetching the list from the storage, each node cleans up the list by removing old entries. On the downside, this means at the same time, that each peer needs to refresh its entry regularly. This kind of data structure is called a *soft-state*. The time after which an entry expires is determined by the parameter T_{expiry}. Furthermore, nodes refresh their entries with an interval of $T_{refresh}$. The concrete values for these parameters have to be chosen wisely. In exert, the longer an entry is valid, the longer invalid entries remain in the list. On the other hand, if it expires too quick, the refresh rate is higher, exposing a higher load on the network.

3.3.4 Far Region and AOI

The partitioning of the seamless world into zones allows for efficient node discovery. Furthermore, the zone a node resides in is its far region at the same time. Unlike the area of interest however, the zones have a fixed position. Inside the far region the nodes are fully-meshed. Because the area of interest lies inside the far region

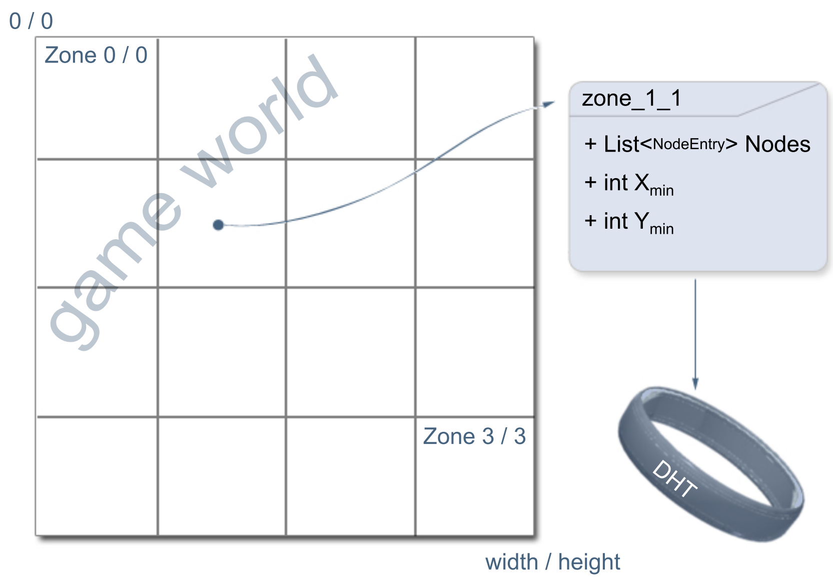


Figure 3.3: Geographic grid and its storage entries

to a certain extent, nodes that are inside the AOI are fully-meshed as well. This reduces the delay with the cost of additional connections.

When a node moves towards a zone frontier, the area of interest potentially overlaps with other zones. This means, that the node needs to detect nodes that are inside its area of interest, without being in the same zone. Figure 3.4 illustrates this problem. On the left, the AOI lies completely inside. In the right image, it overlaps with other zones. How to detect the two indicated nodes is explained in section 3.3.6.

To accommodate the required bandwidth, message exchange between nodes is dependent on the region they lie in. Nodes that lie inside the area of interest are potentially visible, requiring a high update rate using the interval T_{AOI}. Nodes outside the AOI are not visible, so that a lower interval suffices: $T_{AOI} > T_{Zone}$.

In the course of time, nodes join and leave the area of interest and the zone. Whenever a node leaves the AOI, it is handed-off to either the far region or the adjacency region. Likewise, a node that leaves the far region is handed-off to the adjacency region described in the next section.

Note, that the full-mesh inside the far region renders the approach as a whole as

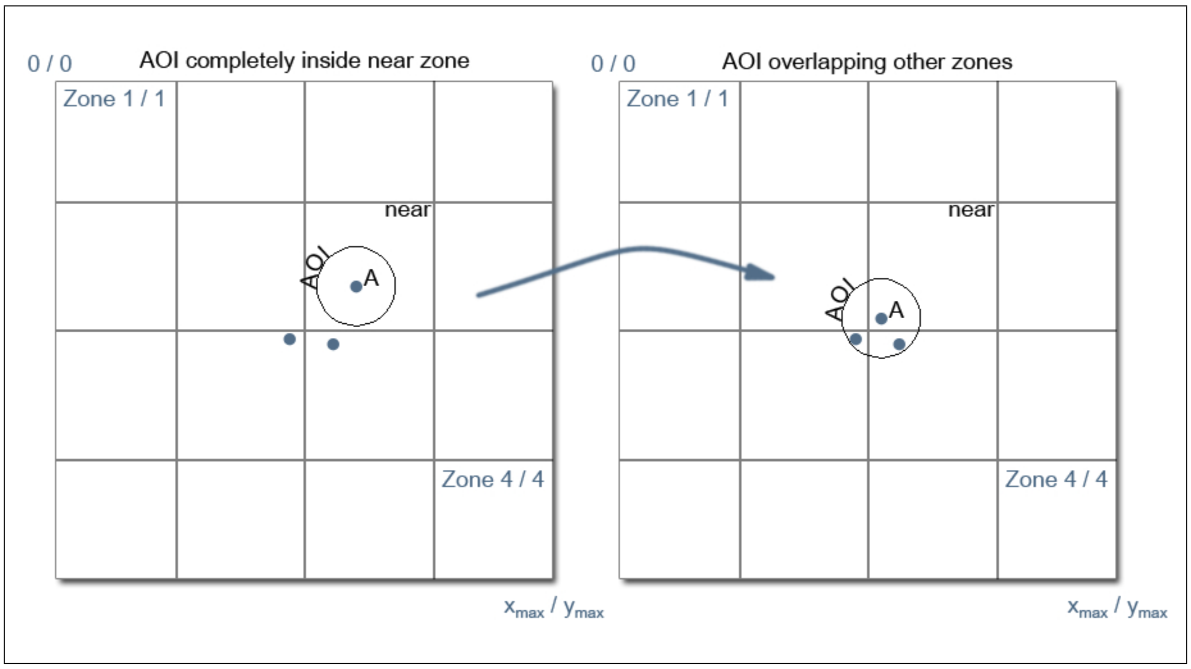


Figure 3.4: AOI moves with the player

not scalable. Because scalability is required, it is worthwhile to consider further possibilities for the connectivity inside the far region. A reduction of connections however, arises the problem of an increasing average delay. A possible solution for this problem might be the use of a *k-degree network* [15]. In a k-degree network, every node holds a minimum of s and a maximum of k connections, with $k/geq2s + 1$. The higher k is chosen, the higher is the resilience of such a network against partition. On the downside, it needs to be balanced. Therefore, the implementation of k-degree network is beyond the scope of this thesis.

3.3.5 Adjacency Region

Besides the area of interest and the far region a third region is created: the *adjacency region*. The adjacency region consists of all surrounding zones of a node; because the eight-neighbour topology is chosen, the adjacency region consists of a maximum of eight zones. To each of these zones, a number of connections is established. The parameter for this number is C_{far} and can be adjusted to a desired value. In practice, this parameter is chosen as $2 <= C_{far} <= 4$.

The connections established in the adjacency region have a twofold purpose: first, the transition into neighbouring zones is simplified and thus more efficient. Therefore, a third interval T_{adj} is used, after which the connected adjacent nodes are informed about position updates. Secondly, they are used to detect margin-nodes that lie inside the area of interest. Note, that we cannot guarantee that each node has connections to each neighbouring zone. This effect is introduced by the connection limit, but does not pose a problem for the frontier discovery described in the next section.

3.3.6 Frontier Discovery

The frontier discovery detects nodes that are inside the AOI, but outside the far region. This occurs when a node moves towards the zone frontier, so that the AOI sphere overlaps with it. In the worst case, the AOI overlaps with a total of three neighbouring zones. To detect nodes that lie in the intersection of the neighbouring zone and the AOI, the adjacency connections are used. Therefore, a frontier discovery request is sent to all connected nodes of the target zone. This message contains the own position. Upon receipt of such a message, a node searches for all other nodes contained in its zone, that lie within the requesters area of interest. This procedure is executed regularly by each node that is near a zone frontier.

The described frontier discovery mechanism works despite of lacking connections to an intersecting zone. Figure 3.5(a) shows such a situation, where node A does not have any connection to the left neighbouring zone. All nodes except for A remain inert while A moves to the frontier in question as shown in figure 3.5(b). Both nodes A and B regularly execute the frontier discovery procedure, if their area of interest overlaps with at least one neighbouring zone. In this case however, A has no chance of detecting B. Instead, B detects A in his next frontier discovery, because he has connections to the zone in which A resides. This mechanism only works if at least one of the nodes has a connection to the adjacent zone. However, this is always true. Two neighbouring zones z_1 and z_2 have m and n number of contained nodes, respectively. If $m > n$, then there exist nodes in z_1 without a connection to z_2. Therefore, when a new node enters z_2 it is always possible to connect to a node in z_1.

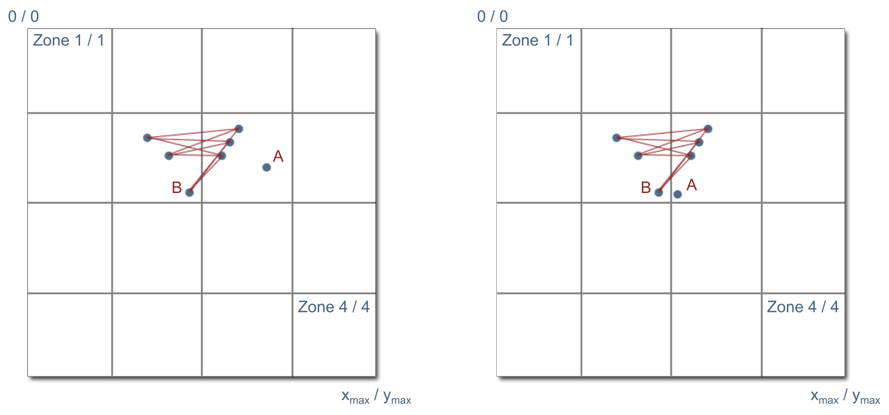


Figure 3.5: Discovery problems near frontiers.

3.3.7 Summary

The presented approach subdivides the world into equally sized zones. Nodes are registered for reference in the zone they lie in and establish a full-mesh inside their zone. The interval at which update messages are sent to connected nodes is dependent on the distance, categorized into three groups: the AOI, the far region and the adjacency region. Furthermore, the intervals are dependent on the maximal running speed of avatars. The area of interest is relative to the node position. The far region matches the bounding box of the current zone and the adjacency region consists of the neighbouring zones. Once a node approaches the zone frontier, the frontier discovery is processed regularly.

The presented approach is expected to meet most requirements stated in section 3.3.1. One exception to this is the scalability requirement. Nodes that are situated in the same zone are building a full-mesh, being not scalable. The delay-minimization is guaranteed by connecting to each node in the vicinity. It is expected, that minimal bandwidth is required to operate the system. This is accommodated by reducing the frequency in that messages are sent depending on the distance. The proposed approach organizes itself by the use of the storage layer. No manual interaction is required to operate the system. It keeps track of all nodes in a zone by managing a soft-state list. The last requirement, locality-awareness, is fulfilled by nature. Nodes are never supposed to connect to distant nodes. The concrete distance is dependent on two factors: the size of the zones and the diameter of the area of interest.

4 Implementation

This chapter gives detailed information on the implementation developed for the thesis. In the first part of this chapter, the architecture is described in detail and interfaces for the usage of each layer are introduced. The second part consists of a detailed introspection into the key algorithms of the proximity management layer as a crucial preparation for the theoretical analysis section of the following chapter 5.

The implementation has been developed using C# and Visual Studio 2008. The peers@play project completely relies on asynchronous programming using the **gears4net** message passing API. Therefore, no threading related code can be found throughout the project. Nevertheless, modern multi-core CPUs are supported, because each gears4net protocol runs in its own thread. The project only allows the usage of interfaces for inter-layer communication. This enables the independent exchange of a single layer. Furthermore, the project incorporates its own coding conventions:

Classes and methods are beginning with capital letters and method names are chosen to be imperative. **Properties** are beginning with capital letters as well and represent the only way to publicly present variable values. **Variables** are beginning with small letters and must be private.

In the peers@play project, messages that are to be sent over the network are assembled stack-wise. The message initiator creates an instance of the **ByteStack** class, that allocates a byte array with a default size of 256. Every layer pushes its required information onto the stack. The **LinkManager** can then send the used bytes over the network. This procedure was employed for high performance serialization. The C# serialization classes present an alternative. However, they serialize data with a very high memory overhead, because they are general-purpose. Therefore, a specialized serialization method has been chosen.

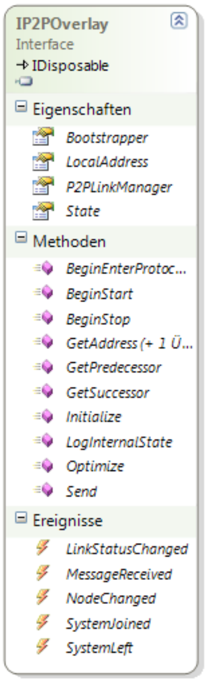


Figure 4.1: Interface: IP2POverlay

The following sections are structured this way: After presenting the layer interface, the class diagrams are visualized and the key procedures are detailed.

4.1 Overlay

The overlay features a preparatory task for this thesis and therefore will only be outlined briefly. Figure 4.1 shows the interface chosen for overlays in the peers@play project.

For unified resource disposal, it inherits the **IDisposable** interface. The properties **Bootstrapper** and **P2PLinkManager** give upper layers access to the instantiated bootstrapper and the link manager, respectively. The property **LocalAddress** gives reading access to the Overlay-address, whereas **State** expresses the state the overlay is in at an instant of time, e.g. whether its still bootstrapping or running.

The methods **BeginStart** and **BeginStop** initiate the bootstrapping and stop the execution of the overlay, respectively. Their names stem from the asynchronous

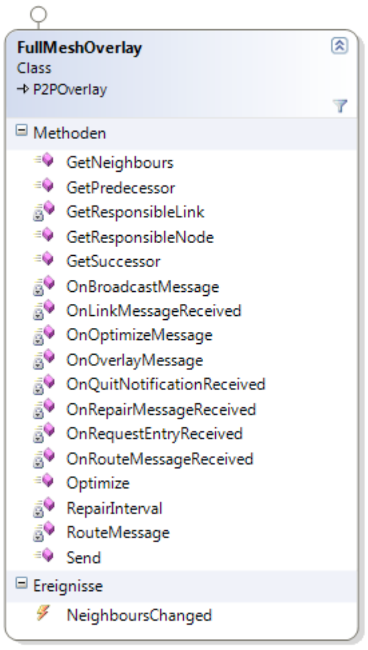


Figure 4.2: Class: FullMeshOverlay

programming model used throughout the project. Because the task has not been finished after the methods return, an asynchronous callback is used[1]. It is called upon finished joining or leaving, respectively. The method **Send** handles message sending to other nodes of the network, whereas the **MessageReceived**-event notifies registered classes of a message receipt. The **Optimize** method can be used to optimize a connection to another node. This optimization can either be delay-minimization or maximization of available bandwidth.

The methods **Send** and **Optimize** take an *overlay address* of a peer as parameter. This overlay address differs between overlay implementations. In the case of *Chord*, this overlay address would contain nothing else than the *Chord-ID*.

The implementation of an overlay used for this thesis is the previously-motivated **FullMeshOverlay**. Note, that this class does not directly inherit from **IP2POverlay**, but from an additional, abstract class called **P2POverlay**. This abstract class contains standard procedures for overlays to decrease code duplication: initialization and clean-up routines are defined centrally, while they call abstract methods, that are to be implemented by the **FullMeshOverlay**.

[1]A callback is a method or function, that is potentially called from another thread. Callbacks enable asynchronous notification when more than a single thread is utilized.

Figure 4.2 contains the class definition in question. Note, that this figure only shows important members to avoid bloating and that inherited members are not shown as well. For information on the structure and the organization of the **FullMeshOverlay**, see section 3.1.

Listing 4.1 shows pseudo-code of the joining procedure into the overlay. This method is called after the node is successfully integrated in the overlay and the storage. The description of methods can be found as in-line comments.

Listing 4.2 shows the procedure that is used for routing.

```
function JoinOverlay()
{
    // start the bootstrapper
    Bootstrapper.BeginGetPeers();

    // wait for the bootstrapping result
    yield return Receive<BootstrappingInfoMessage>();

    // get node list from the result
    List<Node> nodes = lastMessage.Nodes;
    if (nodes.Empty())
    {
        // node alone! join succeeded.
    }
    else
    {
        // traverse all found bootstrapping nodes
        foreach (Node node in nodes)
        {
            // connect to the node
            ConnectTo(node);

            // request the entry into the network
            RequestEntry(node);
        }

        // wait for any reply
        yield return Receive<AcceptEntry>();

        // connect to each supplied nodes
        ConnectToAll(lastMessage.Nodes);

        // notify each node, that we're in
        NotifyAllNodes();
    }
}
```

Listing 4.1: FullMeshOverlay Joining Procedure

```
function RouteMessage(OverlayMessage msg)
{
    // get the peer that is responsible for the key
    LinkAddress responsible = null;
    if (GetResponsibleLink(msg.Destination, out responsible))
    {
        // localnode is responsible for this message!
        InvokeMessageReceived(msg);
    }
    else
    {
        // responsible node found, continue routing
        SendMessage(responsible, msg);
    }
}
```

Listing 4.2: FullMeshOverlay Routing Procedure

The method **GetResponsibleLink** calculates which connected node is nearest to the destination. In a fully-meshed overlay, however, the destination node is directly found, so that only one hop is necessary. The return value is a boolean indicating whether oneself is the receiver of the message.

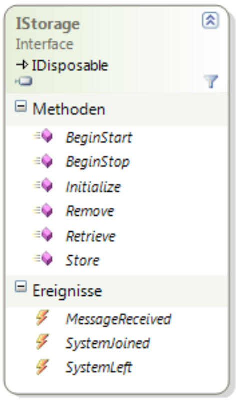


Figure 4.3: Interface: IStorage

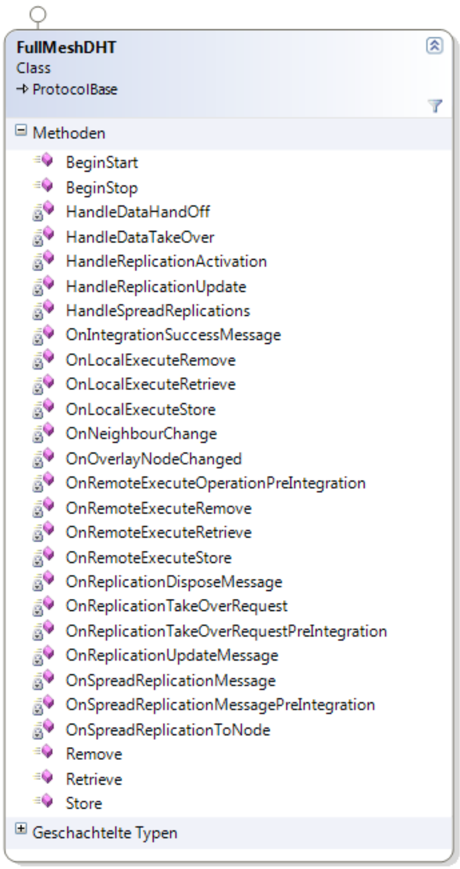
4.2 Storage

The storage system bases upon the overlay. To function correctly, it requires some additional information from the overlay. For example, it needs to be able to query its neighbours in the network. In the **Full-Mesh DHT**, as well as in **Chord**, the neighbours are the nodes with the minimal distance in the key space.

Figure 4.3 shows the **IStorage** interface. The three methods **Store**, **Retrieve** and **Remove** resemble the typical DHT operations. Because of the asynchronous programming model used in the project, they do not block until the result is available. They rather return immediately and invoke a callback when finished. To be able to distinguish an arbitrary number of operations, these methods return a **Guid**[2].
The class **FullMeshDHT** is partly illustrated in figure 4.4(a).

In addition to the interface methods required, there exist several groups of message handlers. The processing of DHT queries is done by *local* and *remote* handlers. The local handler is executed by the initiator, whereas the remote handler is executed by the owner of the queried data set. Furthermore, for these handlers, there exist *pre-integration* variations. These are used instead, before the node is completely integrated into the network. They forward possible mislead messages to their original destination, the nodes' successor. This is required, because of the detachment of overlay and storage. When the overlay integration succeeded, the

[2]A Guid is a globally unique identifier. It is provided by the C# framework.

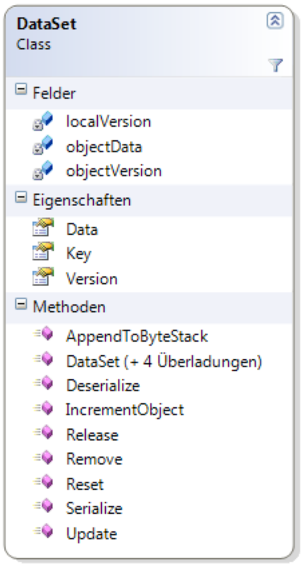


FullMeshDHT
Class
→ ProtocolBase

Methoden
- BeginStart
- BeginStop
- HandleDataHandOff
- HandleDataTakeOver
- HandleReplicationActivation
- HandleReplicationUpdate
- HandleSpreadReplications
- OnIntegrationSuccessMessage
- OnLocalExecuteRemove
- OnLocalExecuteRetrieve
- OnLocalExecuteStore
- OnNeighbourChange
- OnOverlayNodeChanged
- OnRemoteExecuteOperationPreIntegration
- OnRemoteExecuteRemove
- OnRemoteExecuteRetrieve
- OnRemoteExecuteStore
- OnReplicationDisposeMessage
- OnReplicationTakeOverRequest
- OnReplicationTakeOverRequestPreIntegration
- OnReplicationUpdateMessage
- OnSpreadReplicationMessage
- OnSpreadReplicationMessagePreIntegration
- OnSpreadReplicationToNode
- Remove
- Retrieve
- Store

Geschachtelte Typen

(a)
FullMeshDHT



DataSet
Class

Felder
- localVersion
- objectData
- objectVersion

Eigenschaften
- Data
- Key
- Version

Methoden
- AppendToByteStack
- DataSet (+ 4 Überladungen)
- Deserialize
- IncrementObject
- Release
- Remove
- Reset
- Serialize
- Update

(b)
DataSet

Figure 4.4: Classes: FullMeshDHT and DataSet

node is not yet integrated into the storage.
The data stored in the DHT is stored using the **DataSet** class, illustrated in figure 4.4(b).

This class stores both a binary and a de-serialized copy of the object itself. Additionally, each data is version-controlled. To update a dataset, it is required to know its current version. The store is rejected by the responsible node if a deprecated version number was submitted. This simple trick eradicates the chance of lost updates.

The method **AppendToByteStack** appends the serialized data including the version and the key onto a given **ByteStack** and by the use of **IncrementObject**, the version number is incremented. The **DataSet** class is used in all DHT queries. For retrieve and remove operations, only a small fraction of the class is sent, because the payload is irrelevant. It is only relevant for store operations. These small fractions are also called *headers*. Headers are cached locally to increase efficiency and reduce the memory consumption.

4.3 Proximity Management

The proximity management consists of two projects. The first one is the core component called **ProximityManager**. The second one is a GUI for testing and evaluation.

This section will first show some helper classes used by the proximity manager. Then it will outline the provided interface for upper layers followed by implementation details on the core component and its key methods.

The interface of the proximity manager is relatively small. In this first version only one method exists: **Move**. Each time a player moves around the game world, this method needs to be called. For further ideas on how to improve the usability of the layer, refer to section 6.2.

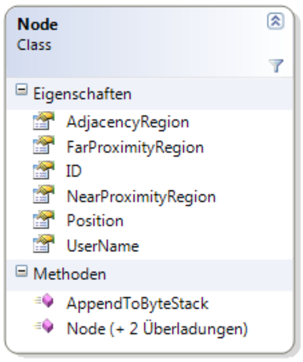


Figure 4.5: Class: Node

The class **Node** is used to internally represent a node. Figure 4.5 illustrates it. Nodes are distinguished by their **OverlayAddress** in the property **ID**. Additionally, they have an associated position represented by a vector. By the use of the position, its area of interest, its far region and its adjacency region can be calculated. To save bandwidth it is convenient to omit the three regions, because they can be calculated using the position.

For simulation purposes, the proximity manager uses an intermediary class called **Communicator** for network access. This additional layer can be operated in two modes: *local* and *online*. When operating locally, only instances inside the same process can communicate with each other without the use of a peer-to-peer network. This brings a crucial performance increase for simulation and bug-hunting. The online mode uses the underlying storage and overlay to send messages and database access. This mode is used for distributed testing using a number of different computers. In addition to the mode switching, the communicator can collect statistical data, such as the amount of messages sent in total or the bandwidth consumption of a peer. This data is then send to an evaluation server if desired.

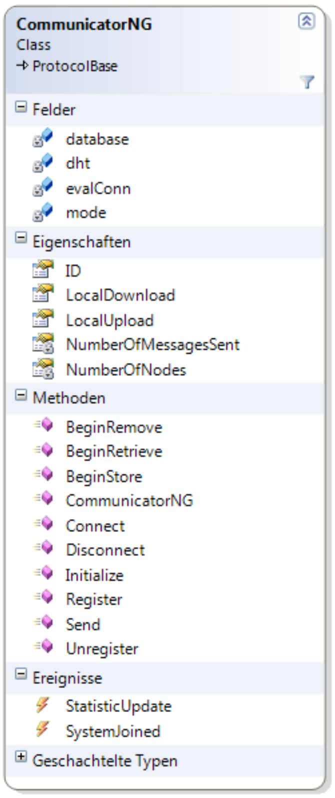
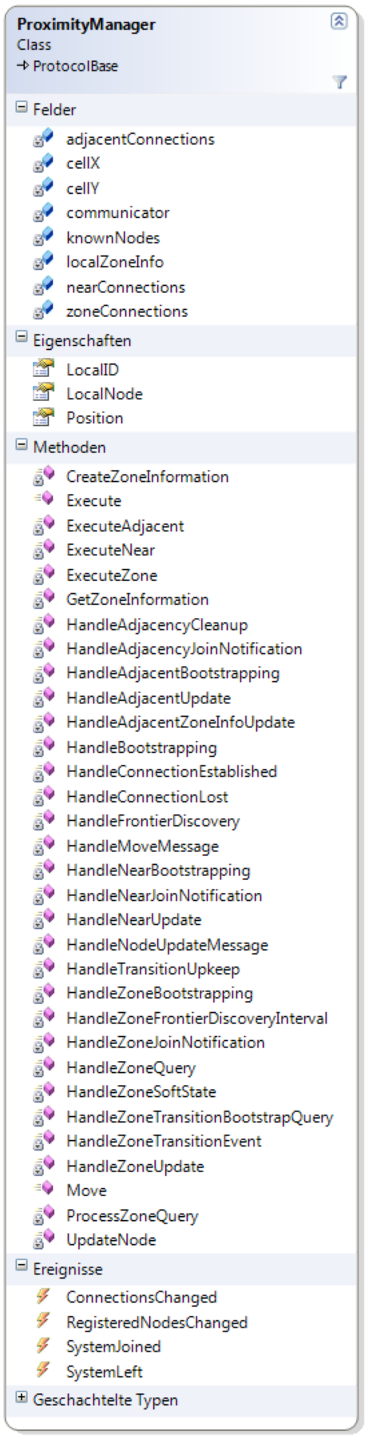


Figure 4.6: Class: Communicator

Figure 4.6 contains its class diagram. The variable **database** contains the local implementation of the storage, whereas the **dht** variable is of type **IStorage** and contains the online variant. The connection to the evaluation server is processed by the **evalConn** object, which is a gears4net protocol built using TCP.

The class contains the core methods of the overlay and storage interface to pass through the requests depending on the mode. The methods **Register** and **Unregister** are used for local mode where each node has to register itself with the same communicator.

The most important class, however, is the **ProximityManager** itself. It holds



Felder: adjacentConnections, cellX, cellY, communicator, knownNodes, localZoneInfo, nearConnections, zoneConnections

Eigenschaften: LocalID, LocalNode, Position

Methoden: CreateZoneInformation, Execute, ExecuteAdjacent, ExecuteNear, ExecuteZone, GetZoneInformation, HandleAdjacencyCleanup, HandleAdjacencyJoinNotification, HandleAdjacentBootstrapping, HandleAdjacentUpdate, HandleAdjacentZoneInfoUpdate, HandleBootstrapping, HandleConnectionEstablished, HandleConnectionLost, HandleFrontierDiscovery, HandleMoveMessage, HandleNearBootstrapping, HandleNearJoinNotification, HandleNearUpdate, HandleNodeUpdateMessage, HandleTransitionUpkeep, HandleZoneBootstrapping, HandleZoneFrontierDiscoveryInterval, HandleZoneJoinNotification, HandleZoneQuery, HandleZoneSoftState, HandleZoneTransitionBootstrapQuery, HandleZoneTransitionEvent, HandleZoneUpdate, Move, ProcessZoneQuery, UpdateNode

Ereignisse: ConnectionsChanged, RegisteredNodesChanged, SystemJoined, SystemLeft

Figure 4.7: Class: ProximityManager

information about all known nodes of the network and manages their updates and positions and uses the **Communicator** class described previously to process networking tasks. Most importantly, however, the class tracks the movement of nodes in the vicinity in the method **HandleNodeUpdateMessage**. Depending on their current distance, their update interval is adjusted. This is done by pushing the nodes into the different zones. The updates are then sent by the three methods **HandleNearUpdate**, **HandleZoneUpdate** and **HandleAdjacencyUpdate**. When a node oversteps the frontier of its current zone, a zone transition event is fired. This is handled by **HandleZoneTransitionEvent**.

Currently, the only interface to upper layers is the **Move** method. By calling the move method, one can signalize the proximity manager that the avatar shall move by a certain offset in space. Using the move instruction, the proximity manager can maintain the AOI and near regions and eventually transition the node to an adjacent zone. The **HandleMoveMessage**-method will be detailed in the following section.

4.3.1 Key Methods

This section elaborates on key methods used in the proximity manager, namely **HandleBootstrapping**, **HandleMoveMessage**, **HandleZoneTransitionEvent** and **HandleFrontierDiscovery**. Beginning with the bootstrapping, this section first outlines the application of a method, followed by pseudo-code and its description.

HandleBootstrapping.
The life-cycle of a peer-to-peer application begins with bootstrapping, where a new peer strives towards the integration into the network. In analogy, the proximity manager needs to bootstrap as well. The procedure begins with retrieving the data for the desired avatar from the storage. Afterwards, the zone in which the avatar is situated is calculated. Then, all nodes lying in this zone are detected. Afterwards, the full-mesh is established inside the far region. Listing 4.3 contains pseudo-code for this method. All operations are described as comments.

```
function HandleBootstrapping()
{
    // retrieve avatar information (position, ...)
    Avatar localAvatar = RetrieveAvatarInformation();

    // calculate the zone the avatar is situated in
    int xZone = (int)(localAvatar.Position.X / Settings.UnitsPerRegionX)
        - 1;
    int yZone = (int)(localAvatar.Position.y / Settings.UnitsPerRegionY)
        - 1;

    // retrieve the zone object from the DHT
    Zone localZone = GetZoneObject(xZone, yZone);

    // start the zone bootstrapping
    HandleZoneBootstrapping(localZone);

    // bootstrap adjacent zones
    HandleAdjacencyBootstrapping();

    // bootstrap the area of interest
    HandleNearBootstrapping();
}

function HandleZoneBootstrapping(Zone localZone)
{
    // get a list of potential nodes of the current zone
    List<Node> ZoneNodes = localZone.Nodes;

    // try to get a reliable list of nodes by consulting
    // potential nodes
    while(!ZoneNodes.Empty())
    {
        // get one node and connect to it
        Node n = ZoneNodes.Pop();
        communicator.Connect(n);

        // request further information by sending a ZoneQuery
        communicator.Send(new ZoneQuery());

        // wait for reply
        yield return Receive<ZoneQueryResult>();

        // if the reply is positive, stop the loop
        if (ZoneQueryResult.Success)
            break;
    }

    // establish a full mesh
    ConnectToAll(ZoneQueryResult.Nodes);

    // inform everybody of our joining
    SendToAll(ZoneQueryResults.Nodes, new ZoneJoinNotification());
}

```

```
    function HandleAdjacencyBootstrapping()
    {
        // for each adjacent zone
        for (uint y = 0; y < 3; y++)
        {
            for (uint x = 0; x < 3; x++)
            {
                // if the zone is valid (i.e. not beyond the world boundaries
                // and not equal to the near zone)
                if (Zone[x, y] != null && !(x == 1 && y == 1))
                {
                    // and if we still need connections in this zone
                    if (Zone[x, y].Connection < Settings.
                        AdjacentZoneConnectionsMin)
                    {
                        // then connect to some nodes
                        communicator.Connect(GetZoneNodes(Zone[x, y]));
                    }
                }
            }
        }

        // inform everybody of the established adjacency connection
        SendToAll(AdjacencyNodes, new ZoneAdjacencyNotification());
    }

    function HandleNearBootstrapping()
    {
        // nothing to do here; the zone bootstrapping
        // does all necessary steps
    }
```

Listing 4.3: The HandleBootstrapping implementation

HandleMoveMessage.
An input by the user potentially initiates the movement of the avatar. Therefore, the game can instruct the proximity manager to move the avatar around in the game world. Listing 4.4 shows the procedure triggered in the event of such a move instruction.

```
1    function HandleMoveMessage(MoveMessage msg)
2    {
3        // update the avatar position
4        localAvatar.Position += msg.Delta;
5
6        // calculate the zone in that the avatar resides after the movement
7        int localX = Math.Max(Math.Min((int)(localAvatar.Position.X /
             Settings.UnitsPerRegionX)));
8        int localY = Math.Max(Math.Min((int)(localAvatar.Position.Y /
             Settings.UnitsPerRegionY)));
9
10       // check whether the avatar transitioned to another zone
11       if (localX != localZone.X || localY != localZone.Y)
12       {
13           // if it has, initiate a transition event
14           InitiateTransitionEvent(localX, localY);
15       }
16   }
```

Listing 4.4: The HandleMoveMessage implementation

HandleZoneTransitionEvent.
Line 14 in listing 4.4 initiates a zone transition event. This event is processed by the method **HandleZoneTransitionEvent** shown in listing 4.5.

```
function HandleZoneTransitionEvent(ZoneTransitionEvent msg)
{
    // clean up the old zone information (i.e. remove own adress from
    // the node list)
    CleanUpZoneList(msg.OldZone);

    // add own address to the new zones' node list
    UpdateZoneList(msg.NewZone);

    // recalculate the distance to each known node and move them
    // into the correct region handler:
    //  - previous zone nodes are moved to the adjacency region
    //  - adjacent nodes inside the new zone are moved to the far region
    UpdateKnownNodes();

    // connect to all nodes in the new zone corresponding to the DHT
    // entry of that zone
    ConnectToAll(msg.NewZone.Nodes);

    // call the adjacency bootstrapping again to re-establish adjacency
    // connections for each adjacent zone
    HandleAdjacencyBootstrapping();
}
```

Listing 4.5: The HandleZoneTransitionEvent implementation

HandleFrontierDiscovery.
Once a node approaches zone frontiers, it is crucial to find the nodes that are beyond the frontiers and nevertheless inside the area of interest. Therefore, the **HandleFrontierDiscovery** method shown in listing 4.6 is called regularly.

```
function HandleFrontierDiscovery()
{
    // for each near and adjacency node
    foreach (Node node in nearNodes or adjacentNodes)
    {
        // if the nodes' area of interest intersects the own zone
        if (node.OverlapsFrontiers())
        {
            // send node a message containing all nodes in his vicinity
            // we know of
            communicator.Send(node, new ZoneFrontierDiscoveryMessage());
        }
    }
}
```

Listing 4.6: The HandleFrontierDiscovery implementation

5 Evaluation

The second part of the thesis is a comprehensive evaluation of the developed proximity management layer implementation. Note, that other layers that are described throughout this thesis are not evaluated on their own. They might, however influence the evaluation to some extent.
An evaluation for the bootstrapper can be found in [10], whereas an evaluation for the Full-Mesh Overlay does not make sense by its nature, because it fails to meet the scalability requirement.

This chapter is structured the following way. First, the algorithms will be theoretically analysed for their complexity. Afterwards, the proximity management is simulated to verify the theoretical results. At the end, a conclusion will be presented with major advantages and drawbacks of the approach in question.

5.1 Theoretical Analysis

The first approach analysed is routing update messages using an overlay. This shows the bandwidth consumption for the task by using common overlays. The results are then compared with the bandwidth consumption of the proximity management. Because the full-mesh overlay implemented for this thesis only requires a single hop for routing, this analysis uses the complexity of common overlays. In Chord for instance, routing involves a worst case of $O(logn)$ nodes for each message, where n is the total number of peers in the overlay.

The bandwidth capabilities of common internet connections can be used for comparisons. It is crucial, that the bandwidth consumption of the application does not exceed these capabilities. A list of common internet connections is shown in table 5.1. These numbers are taken from an offer of the *Deutsche Telekom* [16].

type	download in kb/s	upload in kb/s
DSL 2.000	256	24
DSL 6.000	752	72
DSL 16.000	2.000	128

Table 5.1: Bandwidth capabilities of common internet connections

For this analysis, we assume that update messages are only sent to nodes in the area of interest. Furthermore, other messages like maintenance messages of the overlay are not included in the analysis. The number of nodes in the area of interest is denoted with c_{aoi}. The total number of nodes is denoted with n. Updates are sent every $50ms$. Each update message has a size of 30 bytes only for the proximity manager. When using a network connection for communication, the underlying layers add their information and increase the size to a total of 118 bytes. Concluding, each second $20 * 118$[1] bytes will be sent to each near node. When using the overlay to route the message, a single update message will be sent over up to $log(n)$ other nodes in the worst case. This sums up to an amount of messages sent by each node each second of $c_{aoi} * log(n) * 20$. To obtain the total amount of messages throughout the network each second, this value is multiplied by the total number of nodes. Table 5.1 enumerates the total amount of messages per second and the associated *upload*[2] bandwidth consumption per second. The number of nodes c_{aoi} in the area of interest is constantly equal to 10 for each node. This means, that each node sends updates to 10 other nodes.

number of nodes n	messages per second	bandwidth in total kb/s	bandwidth in total in Mb/s	average bandwidth per node in kb/s
10	2.000	230,5	0,23	23,1
100	40.000	4609,4	4,5	46,1
1.000	600.000	69140,6	67,52	69,1
10.000	8.000.000	921875,0	900,27	92,2
100.000	100.000.000	11523437,5	11253,36	115,2

Table 5.2: Upload bandwidth consumption with constant number of near nodes and without proximity manager ($c_{aoi} = 10$)

[1]The factor 20 is the frequency in which update messages are sent. It is calculated by $1000ms \div 50ms = 20$.

[2]Upload and download bandwidth consumption are always identical. However, the available upload bandwidth usually is much lower than the download bandwidth. Therefore, we only regard the upload in the following.

The third and fourth column represent the total bandwidth consumption throughout the network. The last column represents the average bandwidth consumption for a single node. This value increases logarithmically with the number of nodes, which is acceptable. However, they quickly exceed the common upload bandwidth capabilities as shown in table 5.1. Therefore, this approach is unacceptable. Moreover, the delay is expected to increase by the required amount of hops as well. And these results are only for the benign case, in that the number of near nodes remains constant.

Going a step further, the performance for an increasing number of near nodes is analysed. Note, that the total number of nodes is equal to the number of near nodes in this scenario. This represents the worst-case, because all nodes see each other. Table 5.1 illustrates this. For convenience, the fourth and last column values are now displayed in megabytes per second.

number of near nodes c_{aoi}	messages per second	bandwidth in total in Mb/s	average bandwidth per node in Mb/s
10	2.000	0,23	0,02
100	400.000	45,01	0,45
1.000	60.000.000	6.752,01	6,75
10.000	8.000.000.000	900.268,55	90,03
100.000	1.000.000.000.000	112.533.569,34	1125,34

Table 5.3: Upload bandwidth consumption with increasing number of near nodes without proximity manager ($n = c_{aoi}$)

The table shows that an increase of near nodes has a huge impact on the maximum total bandwidth. In fact, it increases quadratically by $O(n^2)$. The average bandwidth per node only increases quasi-linearly by $O(n * log(n))$ and quickly exceeds the megabyte level. However, the number of visible nodes is restricted in a game, since the rendering of avatars is expensive. Therefore, it is improbable to have more than some hundreds of near nodes.

The same analysis follows for the proximity manager. This enables to compare the naive approach with the solution proposed in chapter 3.3. Again, each near node is sent an update every $50ms$. In contrast to the routing approach, the proximity manager establishes direct connections to near nodes. This eliminates the requirement for routing. Without routing, the total amount of messages sent each second is calculated by $c_{aoi} * n * 20$. The first scenario with a constant number of near nodes is illustrated in table 5.1.

number of nodes n	messages per second	bandwidth in total in Mb/s	average bandwidth per node kb/s
10	2.000	0,23	23,05
100	20.000	2,25	23,05
1.000	200.000	22,51	23,05
10.000	2.000.000	225,07	23,05
100.000	20.000.000	2.250,67	23,05

Table 5.4: Upload bandwidth consumption with constant number of near nodes using the proximity manager ($c_{aoi} = 10$)

In this scenario, the average bandwidth per node is constant. This indicates, that an increase of nodes of the network does not have a direct impact on it. But still, it comes close to the physical restriction of the slowest internet connection. A hint on how to leverage this is given in chapter 6.2.

The second scenario consists of increasing the number of near nodes with the results shown in table 5.1. Again, this represents the worst case: the total number of nodes is equal to the number of near nodes.

number of near nodes c_{aoi}	messages per second	bandwidth in total in Mb/s	average bandwidth per node in Mb/s
10	2.000	0,23	0,02
100	200.000	22,51	0,23
1.000	20.000.000	2250,67	2,25
10.000	2.000.000.000	225067,14	22,5
100.000	200.000.000.000	22506713,87	225,1

Table 5.5: Upload bandwidth consumption with increasing number of near nodes using the proximity manager

The numbers show, that the average bandwidth per node increases linearly with the number of near nodes. Figure 5.1 illustrates a comparison between the complexities of both approaches, the naive routing approach and the proximity manager approach. It is visible, that the routing approach exposes a quasilinear increase of required bandwidth, while the proximity manager exposes a linear increase. Despite the lower increase however, the proximity manager requires a lot more connections than the routing approach.

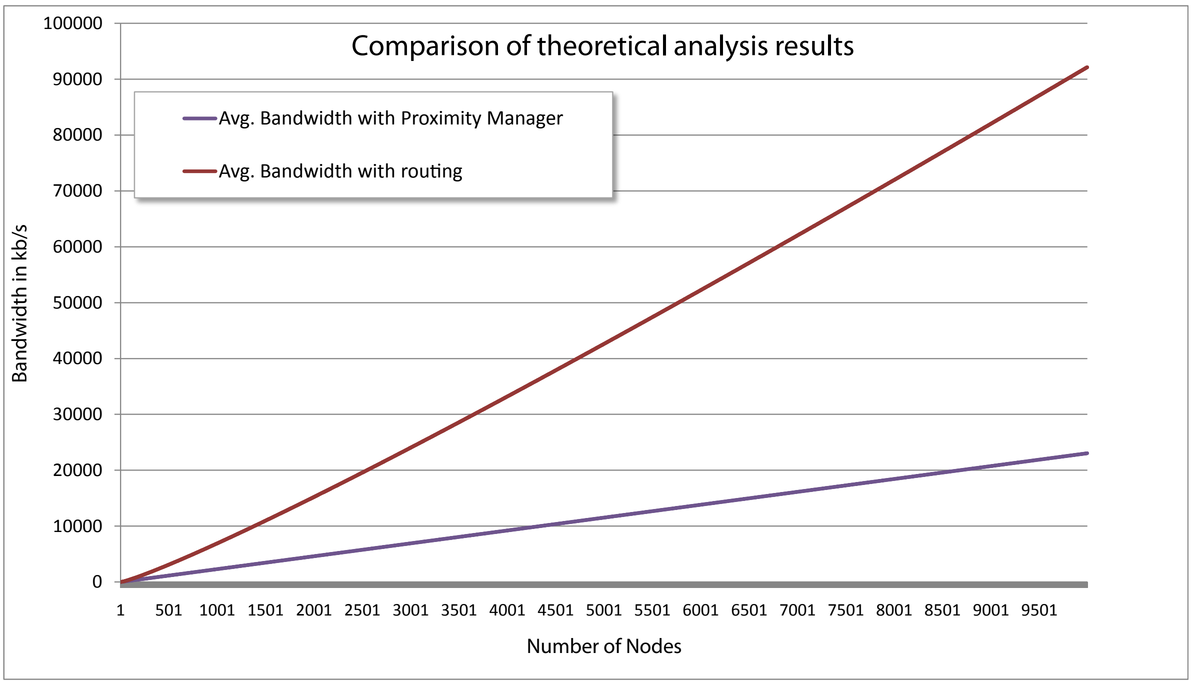
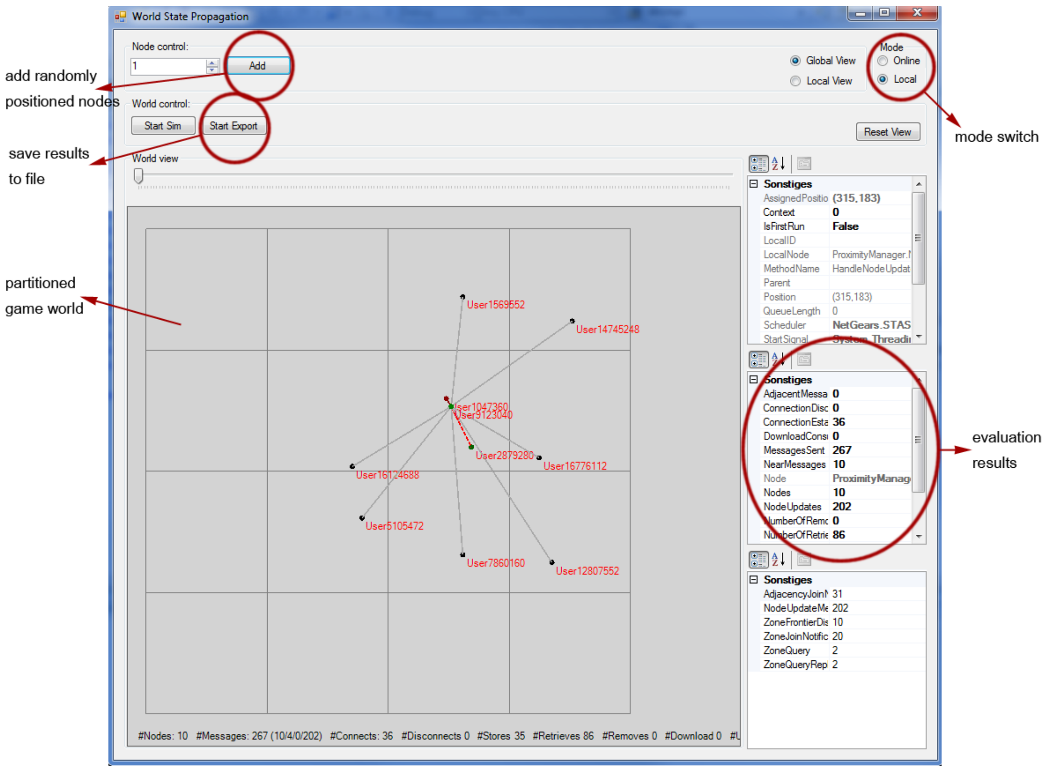


Figure 5.1: Comparison of theoretical analysis results

5.2 Measurements

This section shows the results of undertaken performance measurements of the proximity manager. A custom built simulator aided the process of measuring and is shown in figure 5.2.

Using this tool, it is possible to simulate the proximity manager with certain conditions. It is possible to manually position the nodes and to measure the performance in this situation. We call a certain positioning of nodes a *scenario*. For the evaluation, the world is partitioned into 16 seamlessly linked zones. The simulator allows to assign a random movement to each node. Additionally, one can group nodes together, so that they move together. This is used to approximate the common grouping behaviour of MMORPG players. Each proximity manager instance sends local evaluation results to an evaluation server. The server stores the average results into a file with comma separated values. These files are then



Node control: 1

Mode: Online [✓ Local]

Sonstiges	
AssignedPositio	(315,183)
Context	0
IsFirstRun	False
LocalID	
LocalNode	ProximityManager.!
MethodName	HandleNodeUpdat
Parent	
Position	(315,183)
QueueLength	0
Scheduler	NetGears.STAS
StartSignal	System.Threadi

Sonstiges	
AdjacentMessa	0
ConnectionDisc	0
ConnectionEsta	36
DownloadCons	0
MessagesSent	267
NearMessages	10
Node	ProximityManag
Nodes	10
NodeUpdates	202
NumberOfRemc	0
NumberOfRetrie	86

Sonstiges	
AdjacencyJoinN	31
NodeUpdateMe	202
ZoneFrontierDis	10
ZoneJoinNotific	20
ZoneQuery	2
ZoneQueryRep	2

#Nodes: 10 #Messages: 267 (10/4/0/202) #Connects: 36 #Disconnects 0 #Stores 35 #Retrieves 86 #Removes 0 #Download 0 #U

Figure 5.2: The proximity management simulator

imported into an arbitrary calculation tool and can then be transformed into diagrams. Note, that for performance reasons the simulation is running slower than in the real world. The simulation time is advanced every 200ms. In practice, it is reasonable to advance the time every 50ms. This means, that the bandwidth consumption needs to be multiplied by four to retrieve comparable results. The connection between nodes is indicated by a line. Nodes that are in the area of interest share a continuous, dark-red line. Nodes situated in the same zone share a dashed, light-red line. Connections between nodes of adjacent zones are shown in grey.

In the following, different scenarios are presented and evaluated, concluding with highlighting the most important perceptions gained during evaluation. These scenarios are simulated without the use of a real network and will thus only show the behaviour and complexity of the proximity manager itself. The bandwidth value will always refer to the right vertical axis. Its unit is given in average bytes per second. The number of nodes and the average number of connections will refer to the left vertical axis. The time is given in seconds.

Scenario 1:

The first scenario presented is fairly simple: each zone is assigned a single immutable node in the center. Hence, each node has got an empty area of interest and exactly one connection to each adjacent zone. In theory, the required bandwidth should be very low in this scenario with a maximal number of connections of 8. Figure 5.3 shows the distribution of nodes in the simulator. Besides, figure 5.4 illustrates the results for this scenario.

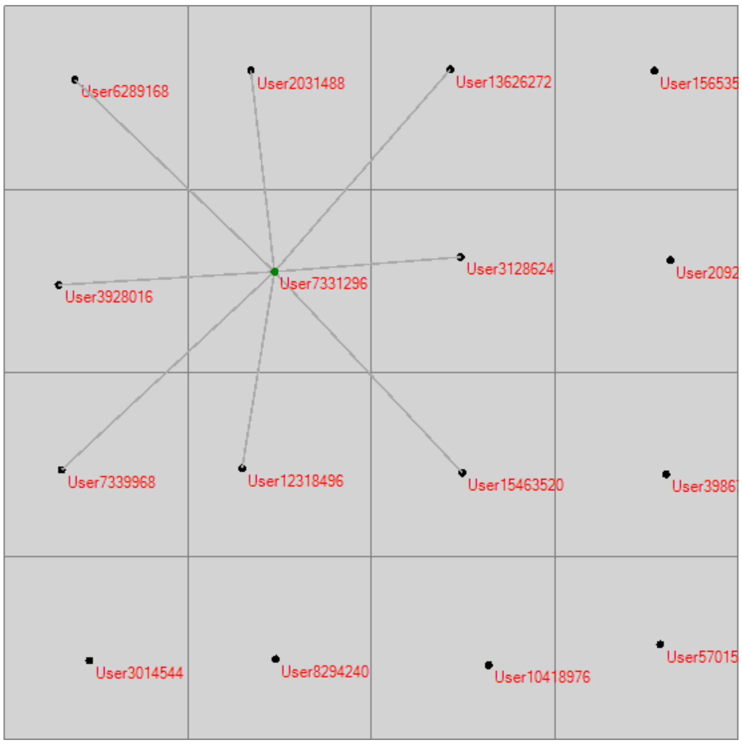


Figure 5.3: Scenario 1: Node distribution

The figure shows that the average bandwidth per node is not proportional to the number of nodes in an homogeneously distributed world. The bandwidth consumption never exceeds the capabilities of common internet connections. The average number of connections increases with the number of nodes; each time a node joins, it will establish one connection to each of its adjacent zones.

Scenario 2a:

The second scenario is split up into two versions a and b. Both versions consist of 20 nodes. They are joined in four groups of five nodes each and are situated nearby. In version a, the four groups are placed into adjacent zones at the center of the

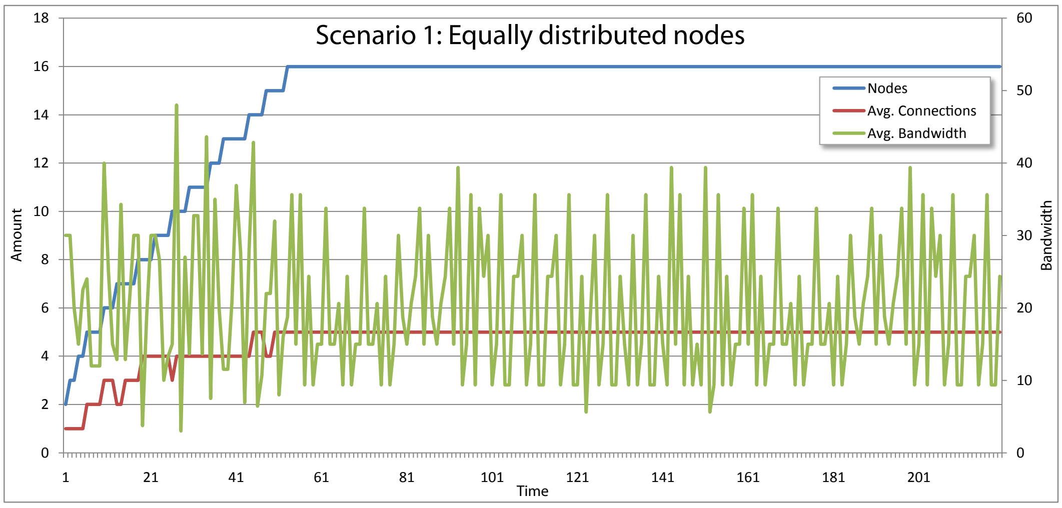


Figure 5.4: Scenario 1: Evaluation results

world. Figure 5.5 illustrates this distribution. This results in five near connections for each node and an additional maximum of nine adjacent connections. The results are shown in figure 5.6.

Compared to the first scenario, the bandwidth consumption is higher. Each group of five nodes exchanges updates highly frequent. The average bandwidth per node for the real-world application is approximately 730 bytes per second, which is still acceptable for common internet connections. The number of connections does not reach the theoretical maximum of 14.

Scenario 2b:

Version b of scenario 2 distributes the groups in different manner; two are positioned at the left world boundary and the other ones at the right world boundary. This is shown in figure 5.7 with the results following in figure 5.8.

Despite the fact, that the two left groups do not connect to the two right groups, the bandwidth is more or less equal to version a. This shows, that adjacent connections incur only little bandwidth overhead. The geographical independence can be seen in the value of average connections: it is always lower to the value in version a. In essence, this means that crowds of nodes do not communicate to each other as long as their distance is high enough. This distance is dependent on the relation of the area of interest to the zone sizes.

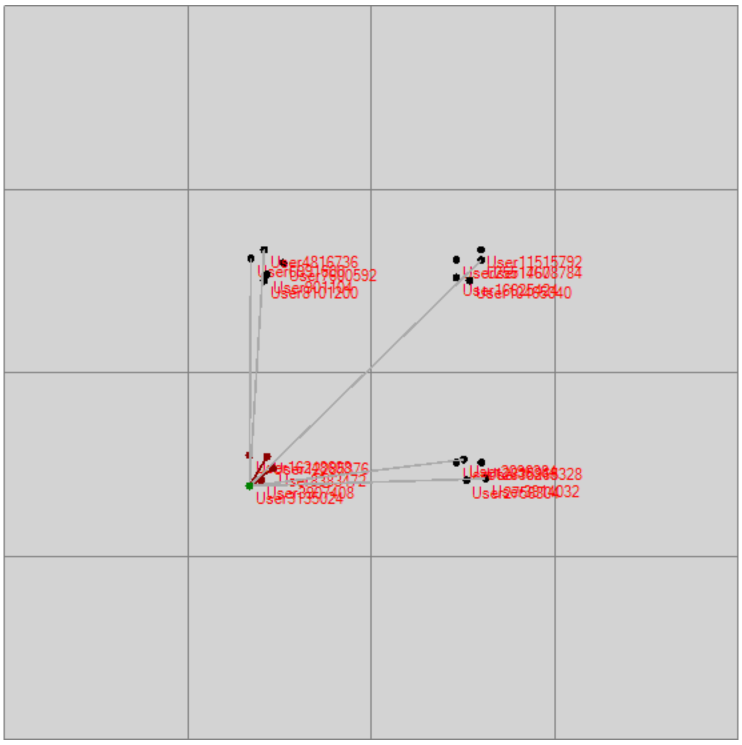


Figure 5.5: Scenario 2a: Node distribution

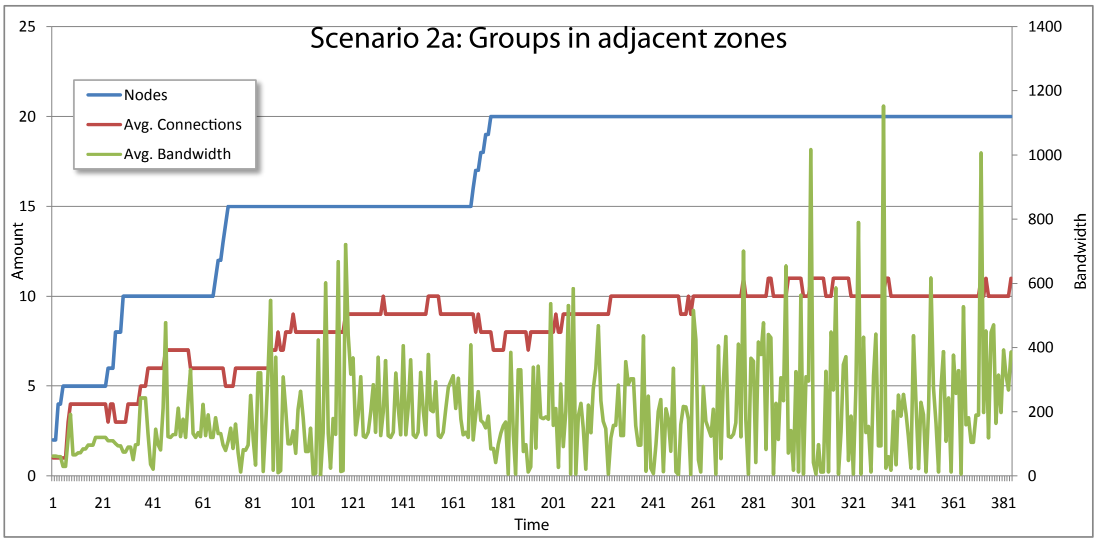


Figure 5.6: Scenario 2a: Evaluation results

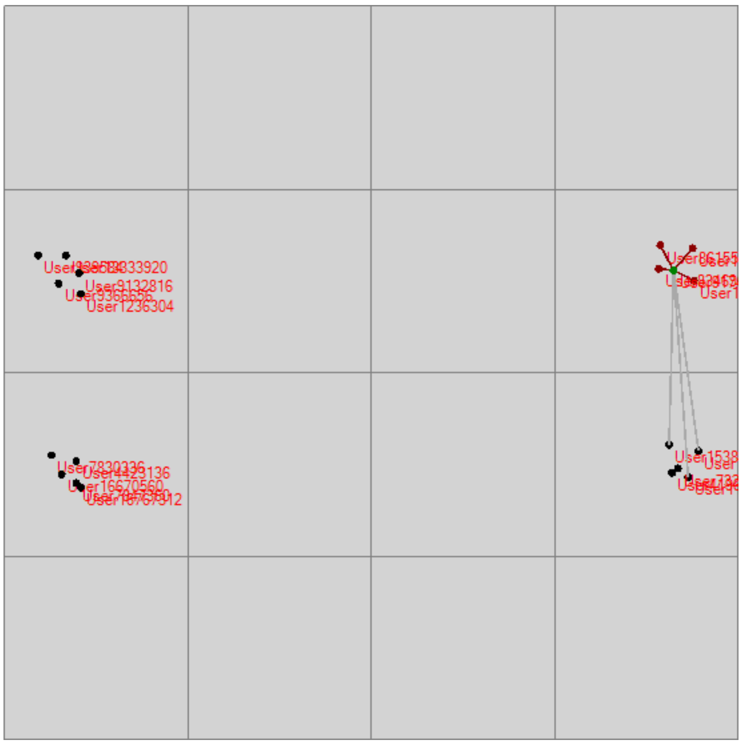


Figure 5.7: Scenario 2b: Node distribution

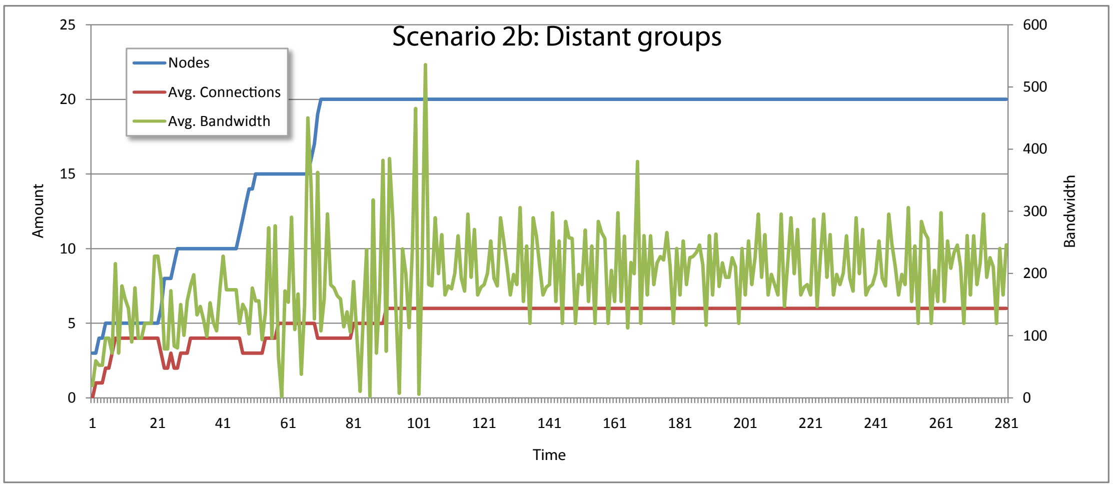


Figure 5.8: Scenario 2b: Evaluation results

Scenario 3:

The third scenario resembles a worst case: 20 nodes are positioned directly besides each other. This yields a theoretical number of connections of 19 and a high bandwidth. The motivation for this scenario is simple: many MMORPGs incorporate the concept of capital cities. Cities are usually not instanced and therefore all players inside the city are visible. For the proximity manager this means, that many players are in the area of interest, increasing the bandwidth consumption. Figure 5.9 shows the dense distribution for this scenario and figure 5.10 illustrates the results.

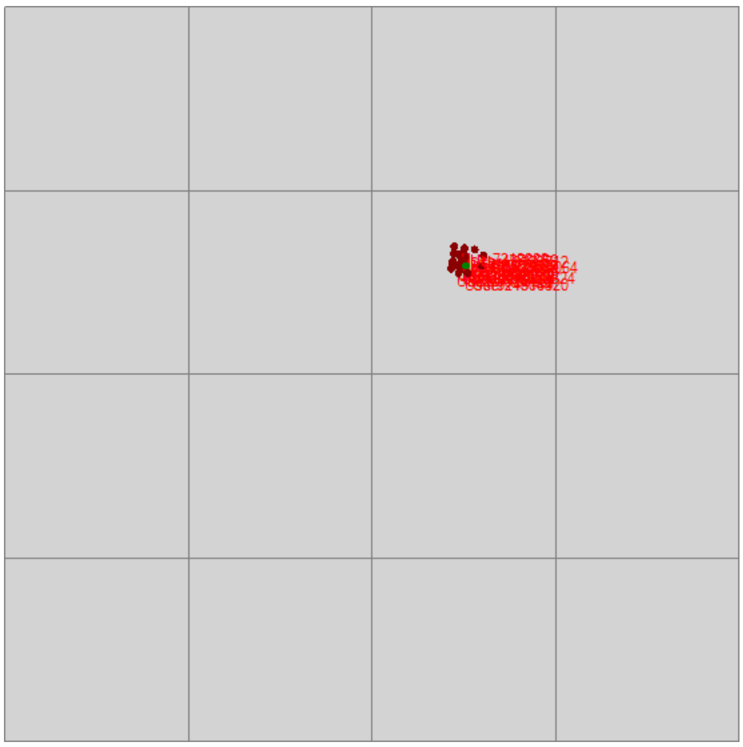


Figure 5.9: Scenario 3: Node distribution

As expected, the number of connections is at its maximum. Additionally, the bandwidth consumption is distinctly higher than in the previous scenarios. Although the bandwidth consumption is still in the capability bounds, such crowds are problematic in this approach, because population numbers of cities in modern MMORGPs easily exceed hundreds of players.

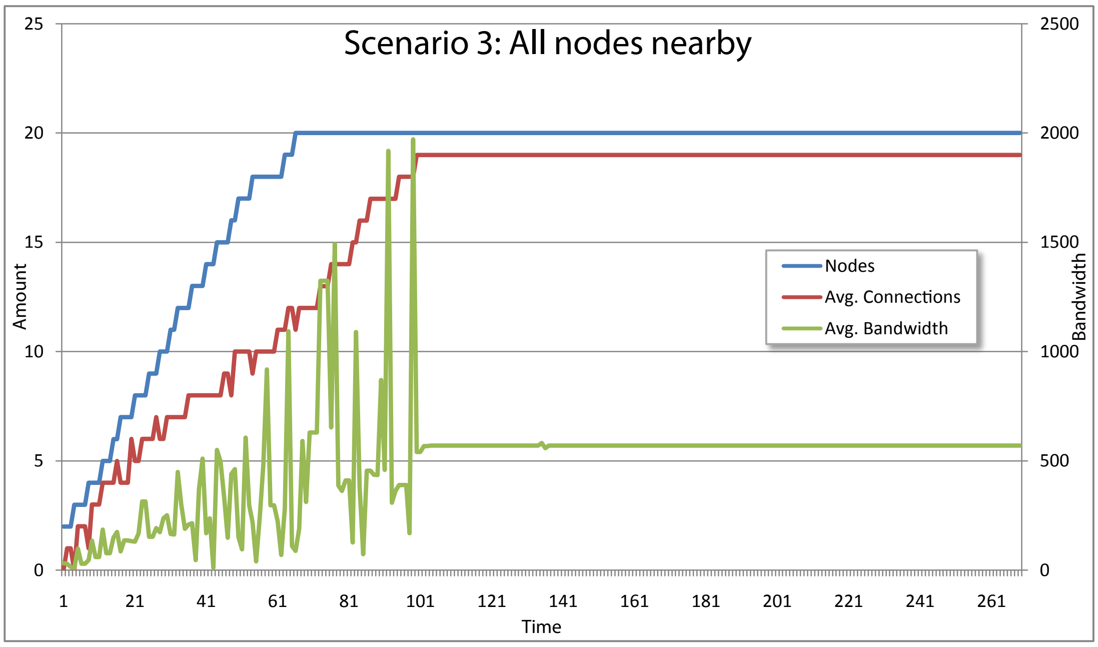


Figure 5.10: Scenario 3: Evaluation results

Scenario 4:
The last scenario simulates groups with movement. Therefore, five different randomly positioned groups consisting of five nodes each are introduced. After they are all logged in, the movement commences. This scenario has been designed to resemble the common behaviour of players in a MMORPG. Again, the two figures 5.11 and 5.12 illustrate the distribution and results.

The results show, that the approach renders itself as useful in such situations. Of course, in real games there exist a lot more than five groups. As a result it can be stated that this approach works well in games with a limited amount of groups in a single surrounding area.

5.3 Evaluation Results

In the previous two section the evaluation of the proximity management was discussed. This section will combine the results and present the major advantages and drawbacks.

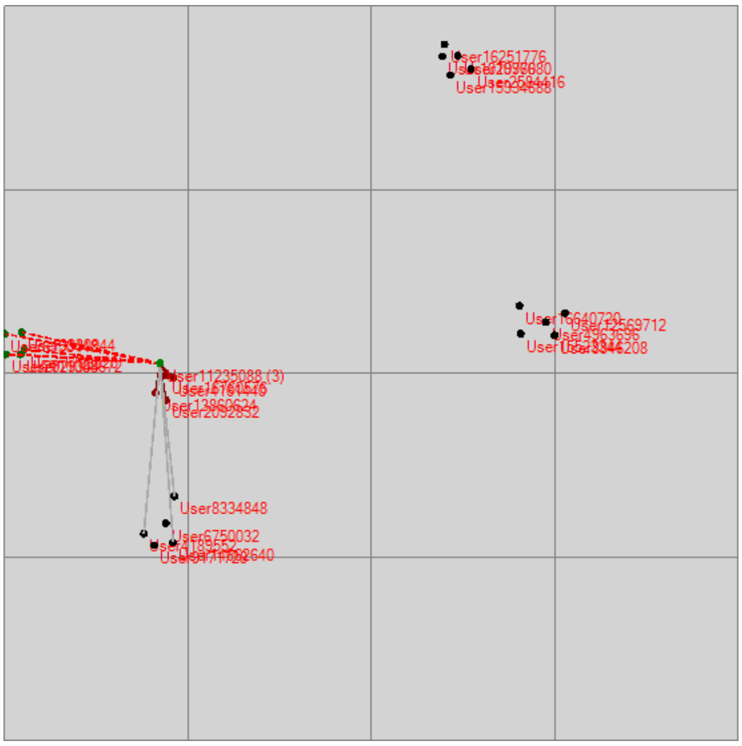


Figure 5.11: Scenario 4: Node distribution

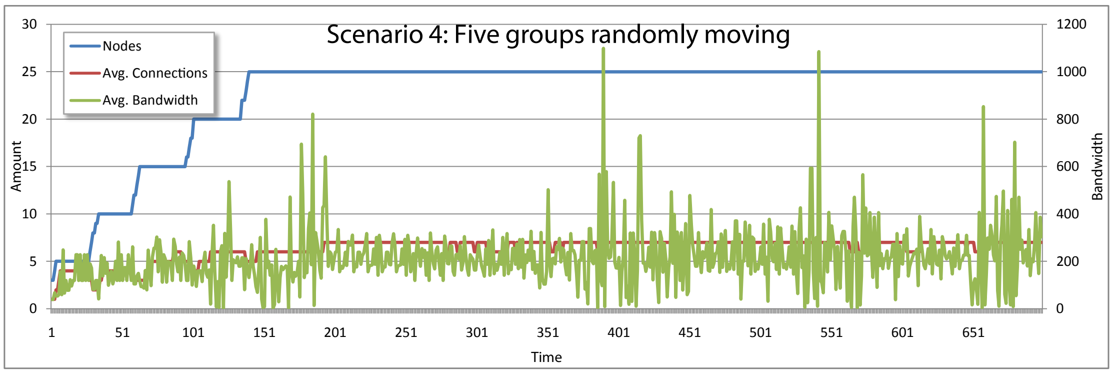


Figure 5.12: Scenario 4: Evaluation results

The theoretical analysis proved that the bandwidth consumption in the presented approach is lower compared to the naive routing approach. The measurements indicate, that the chosen approach is well applicable to environments with a limited number of nodes in the ultimate surrounding. High numbers of near nodes raise high connection numbers and high bandwidth consumption. The amount of sent messages increases with the number of near nodes.

Unfortunately, the evaluation did not give any hint about the desired delay-minimization. It is obvious, that the delay between two connected nodes is at a certain minimum. There may be other paths through the network with lower delays. However, they are hard to detect. Therefore, the delay-minimization is guaranteed by connecting to nearby nodes. To these nodes a high update rate with a minimal delay is required.

Although the best performance is achieved in environments with limited players, the approach is applicable for some existing games. Modern MMORPGs incorporate group systems for up to six players. Additionally, games like World of Warcraft use instancing to enable an arbitrary number of groups to play the same dungeon. This means, that the number of nodes in these dungeons is limited anyway. Other games use instancing even for the open world. A popular representative of this is Guild Wars. Additionally, it uses channelling in cities and other assembly places (where channels are called districts). In Guild Wars, there exists a player maximum of 100 in each district. Upon exceeding this number, a new district is generated. Table 5.1 states, that 100 nodes incur an upload bandwidth consumption of $0,23$ megabytes per second per node (including overlay overhead). This value exceeds the capabilities of common internet connections. To prevent this, the number of nodes or the number of connections can be restricted. For the latter case, one strategy can be a backward trade-off: paying the cost of an increased delay to reduce the required bandwidth. In some cases, this would work well. In cities, for example, fighting is often disallowed and for transactions like trading and chatting, the delay is usually not equally crucial.

6 Summary

This chapter concludes the thesis and shows possible enhancements to incorporate in the future.

6.1 Conclusion

In this diploma thesis, a technique for managing connections in a peer-to-peer based massively multi-player online game is described. Its main remit is the minimization of delay without a substantial increase of bandwidth consumption and required connections. To yield the best results concerning these requirements the approach uses geographical coherence. This is well suited, because the guarantee for a minimal delay only needs to apply to players that are visible, i.e. are situated nearby in the game world. The approach is subdivided into three areas of responsibility: the *area of interest*, the *far region* and the *adjacency region*. These regions are used to categorize nodes. Dependent on the region to that a node is assigned, the frequency for sending update messages is determined.
The evaluation shows, that the presented technique is applicable only for some applications of MMOs. The area of application can still be increased by some means of improvement, outlined in section 6.2. In general, it became clear that the proximity manager works best for environments with a limited amount of nodes in the vicinity. For large, open worlds where a high number of nodes are expected, the full-mesh inside the area of interest does not work, because it does not scale. A high number of visible nodes is generally not scalable. Assume that hundreds of players need to be rendered by a graphics card. This would degenerate the frame rate to a minimum, if not to zero.
As a conclusion, the proximity manager proposed in this thesis works for some scenarios and applications concerning online games. However, there is still need to improve the performance and scalability of the presented technique.

6.2 Future Work

This section gives information on how to overcome some of the drawbacks of the presented proximity management. The evaluation showed, that improvements are required for bandwidth consumption and number of connections. Unfortunately, these requirements are contradictory. A decrease of connections usually means an increase of overall required bandwidth and vice versa. Nevertheless, there exist several methods for further improvements.

Dead reckoning.
The most important improvement that needs to be undertaken in the future is the incorporation of dead reckoning. For an explanation of this technique, refer to section 2.1.6. Currently, each single position update is distributed among all nodes requiring this information. This constitutes the worst case, which can be reached by using dead reckoning as well. However, the bandwidth consumption is expected to be a lot lower. Assume, that a player changes its direction each second. This would engender exactly one message per second to each near node in contrast to the 20 messages required by the current approach. Note, that dead reckoning is only sensible for nodes inside the area of interest.

Gradient frequency adaption.
Assuming two nodes of the same zone, the implementation has exactly two possible frequencies after which node updates are transferred. Instead, it is possible to use gradient values for the frequency. The gradient is equal to the distance metric of two nodes. In practice that means, that the lower the distance, the higher the frequency will be.

Avoiding the zone full-mesh.
The performance of the implementation can be improved by getting rid of the full-mesh of the nodes residing in a zone. Unfortunately, this comes with a cost: the delay is expected to increase, because messages may require more than a single hop to reach its destination. However, in games that allow hundreds of players to be at the same place, a scalable approach is required. One possible approach is the use of a k-degree network. Refer to section 3.3.4 for an explanation of this network type. Besides the increase of delay, a further drawback is that of maintenance. A k-degree network needs to be balanced such that each node meets the consistency requirement. Nevertheless, this technique is very well suited, because the variable

k gives full control on how many connections are established. Thus, the higher k is, the lower is the delay.

Storage modifications.
The storage used for this thesis is a commonplace DHT. Unfortunately, game applications have different requirements towards the storage compared to usual peer-to-peer applications. An example is soft-state handling. The proximity manager uses a soft-state list to detect zone nodes. To refresh the own entry in this soft-state, the complete list has to be fetched from the storage and is afterwards written back to it. This incurs a fairly high bandwidth consumption. It would therefore be convenient to have a storage system, that provides functionality for such purposes. In the best case, to refresh an entry, the node only sends its ID to the responsible node. Upon receipt, the responsible node updates the time-stamp in the local list.

Bibliography

[1] Activision/Blizzard Entertainment. World of Warcraft. published in the WWW at http://www.worldofwarcraft.com, November 2004 (visited in Nov. 2009).

[2] ArenaNet & NCSoft. Guild Wars. published in the WWW at http://www.guildwars.com, April 2005 (visited in Nov. 2009).

[3] Sony Entertainment. Planetside. published in the WWW at http://planetside.station.sony.com, May 2003 (visited in Nov. 2009).

[4] Philipp Rosedale. Second Life. published in the WWW at http://www.secondlife.com, 2003 (visited in Nov. 2009).

[5] Tom-Christian Bjørlo Johannessen Frode Voll Aasen. Hybrid Peer-to-Peer Solution for MMORPGs. Master's thesis, Norwegian University of Science and Technology, 2009.

[6] Roger Smith. *Geographic Grid Registration*, volume 6 of *Game Programming Gems*, chapter Chapter 1, pages 39 – 47. Charles River Media, 2006.

[7] NCSoft. Aion: The Tower of Eternity. published in the WWW at http://www.aiononline.com, September 2009 (visited in Nov. 2009).

[8] Jesse Aronson. Dead Reckoning: Latency Hiding for Networked Games. published in the WWW at http://www.gamasutra.com/view/feature/3230/dead_reckoning_latency_hiding_for_.php, September 1997 (visited in Nov. 2009).

[9] Arno Wacker, Gregor Schiele, Sebastian Holzapfel, and Torben Weis. A nat traversal mechanism for peer-to-peer networks. In *The Eight International Conference on Peer-to-Peer Computing (P2P'08)*, Aachen, Germany, September 8th-11th 2008. IEEE.

[10] Mirko Knoll, Matthias Helling, Sebastian Holzapfel, Arno Wacker, and Torben Weis. Bootstrapping Peer-to-Peer Systems Using IRC. In *Infrastructures for Collaborative Enterprises, 2009*, 2009.

[11] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, San Diego, CA, USA, 2001. ACM Press.

[12] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 161–172, San Diego, CA, USA, 2001. ACM Press.

[13] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object loaction for routing for large-scale peer-to-peer systems. In *Proceedings IFIP/ACM Middleware 2001*, November 2001. Heidelberg, Germany.

[14] RFC 3174: US Secure Hash Algorithm 1 (SHA1). published in the WWW, November 2009.

[15] Arno Rüdiger Wacker. *Key Distribution Schemes for Resource-Constrained Devices in Wireless Sensor Networks.* PhD thesis, Institut für Parallele und Verteilte Systeme (IPVS) der Universität Stuttgart, 2007.

[16] Deutsche Telekom. Komplettpakete zum telefonieren und surfen. published in the WWW, November 2009.

List of Figures

List of Tables

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Duisburg, November 30, 2009 ___________________

Unterschrift