Implementation of a Peer-to-Peer Multiplayer Game with Realtime Requirements

Master-Thesis von Max Lehn
22.10.2009

TECHNISCHE UNIVERSITÄT DARMSTADT

DVS

Implementation of a Peer-to-Peer Multiplayer Game with Realtime Requirements

vorgelegte Master-Thesis von Max Lehn

1. Gutachten: Alejandro Buchmann
2. Gutachten:

Tag der Einreichung:

Abstract

Massively multiplayer online games (MMOGs) have become increasingly popular in the recent years, particularly in the form of online role-playing games (MMORPGs). These games support up to several ten thousand players interacting in a virtual game world. The current commercially successful games are client-server based, which is feasible for relatively slow role-playing games. Those have modest bandwidth and latency requirements and are paid for by their customers. For MMOGs with higher realtime requirements and/or a smaller number of customers willing to pay, peer-to-peer networking seems to be a serious alternative.

This work analyzes the implementation of both a client-server and a peer-to-peer networking model for the prototype shooter game Planet π4. Initially, a survey introduces recent academic approaches to peer-to-peer systems specifically designed for games. Of those, one system is selected for implementation with Planet π4. Planet π4 is improved in several aspects for the purpose of analyzing various network implementations. First, its architecture is restructured and cleaned to allow for an easy replacement of the networking component. Second, its core is modified to work in a completely event-based mode, supporting the execution of the game in an discrete-event-based network simulator. Third, a simple artificial intelligence player is developed for workload generation in large (and possibly simulated) networks. Furthermore, the newly developed transport protocol CUSP is applied for the network implementation, thus the game is the first real application using CUSP. Finally the game Planet π4 is integrated with the CUSP network simulator, allowing to run the whole game in a simulated network without the need for modification of its core components.

Kurzfassung

Massen-Mehrspieler-Onlinespiele (Massively multiplayer online games; MMOGs) wurden in den vergangenen Jahren zunehmend populär, insbesondere in Form von Online-Rollenspielen (MMORPGs). In diesen Spielen interagieren bis zu mehrere Zehntausend Spieler in einer virtuellen Spielwelt. Die zur Zeit kommerziell erfolgreichen Spiele sind Client-Server-basiert, was für vergleichsweise langsame Rollenspiele gut funktioniert. Solche Spiele haben begrenzte Anforderungen an Bandbreite und Latenzen, und ihre Kunden bezahlen für die Nutzung. Für MMOGs mit höheren Anforderungen bezüglich Echtzeitfähigkeiten und/oder einer kleineren Zahl Kunden, die bereit sind zu zahlen, scheint Peer-to-Peer als eine ernstzunehmende Alterative.

Diese Arbeit analysiert die Implementierungen sowohl eines Client-Server- als auch eines Peer-to-Peer-Netzwerkmodells für das prototypische Shooter-Spiel Planet π4. Einführend stellt eine Untersuchung aktuelle, speziell für Spiele entwickelte Peer-to-Peer-Systeme vor. Aus diesen Systemen wird eines für die Implementierung mit dem Spiel Planet π4 ausgewählt. Das Spiel Planet π4 wird in mehrerlei Hinsicht verbessert mit der Absicht, unterschiedliche Netzwerkimplementierungen zu testen. Erstens wird seine Architektur restrukturiert und aufgeräumt, um den einfachen Austausch der Netzwerkimplementierung zu ermöglichen. Zweitens wird sein Kern so modifiziert, dass er komplett ereignisorientiert arbeitet. So kann das Spiel in einem ereignisorientierten Netzwerksimulator ausgeführt werden. Drittens wird ein einfacher Künstliche-Intelligenz-Spieler für die Lasterzeugung in großen (und gegebenenfalls simulierten) Netzwerken. Außserdem wird das neu entwickelte Transportprotokoll CUSP für die Netzwerkimplementierung eingesetzt, so ist das Spiel Planet π4 die erste echte Anwendung welche CUSP nutzt. Schließlich wird was Spiel in den CUSP-Netzwerksimulator integriert, welcher es ohne weitere Änderung in einem simulierten Netzwerk ausfüren lässt.

Erklärung zur Master-Thesis

Hiermit versichere ich die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den October 7, 2009

_____

(Max Lehn)

Contents

1 Introduction

In the recent years, the computer gaming market has evolved to one of the most important (i.e biggest) markets of the entertainment industry. In 2008, computer games started to out-sell music CDs and video DVDs [25]. While much of the gaming marked counted here consists of single player games which may optionally allow to play in small groups either in a LAN or over the Internet, there is also a rapidly growing market of pure online games.

Massive multiplayer online games (MMOGs) allow large numbers of players (tens up to thousands) to interact in one common game world. Most importantly within the group of MMOGs, there are the role-play games (MMORPG) where players have their fictional characters (typically in an ancient fantasy world) solving quests together and thereby gaining experience (skill, etc.). The currently by far most popular MMOG is World of Warcraft[1] by Blizzard Entertainment which was released in 2004 and has reached 11.5 million subscribers at the end of 2008 [12, 38]. From a commercial perspective, having more than 10 million customers, each paying about 19$ per month (Europe: 13€), creates a good margin. Running servers for such a big amount of players causes costs that are not negligible. Server maintenance costs for World of Warcraft are estimated around $100 million per year [7]. Thus, large MMOGs require an adequate number of users willing to regularly pay for their accounts.

From a technical perspective, running powerful servers (typically server farms) is an acceptable solution. Games that are currently on the market show that this apparently works well enough. But as stated before, this requires a certain amount of financial resources. An additional limitation of current MMOGs is in the number of players that really play together in the same virtual universe. Altough large online games count millions of subscribers, they are split up into several worlds, i.e., distinct game worlds which do not allow for any intermediate interaction. In World of Warcraft, these separate game universes are called *realms*[2]. The maximum number of players in each of those worlds typically varies from a few thousand in World of Warcraft to several tens of thousands (e.g., in EVE Online[3] and Second Life[4]).

Depending on the particular game, the number of players in the game universes may be bounded by gameplay considerations. Small worlds limit the emergence of communities, while too many players may cause important places to be overcrowded, making the game unenjoyable. But more importantly, there are technical limitations to the number of players, both on server and on client side. One game universe must be handled by a server or a server farm that is highly synchronized as each player in the universe may potentially interact with each other. This fact and the hardware limitations naturally limit the size of a universe. There are server-based techniques that overcome this problem; the most common approach is zoning. The game world is split into disjoint spatial *regions* (zones), typically squares, which are each handled by a separate server. Assuming that interaction is only possible between nearby players, this reduces the necessary synchronization; each server only needs to communicate with adjacent servers.

This thesis concentrates on realtime capabilities of peer-to-peer MMOGs. The game used for implementation and analysis of network models is Planet π4 (introduced in section 1.1), an early prototype of a realtime MMOG. At the time of writing, there are no commercial MMOGs with as high realtime requirements as the shooter Planet π4 has. It has not yet been proven that the genre of shooters is suitable for MMOGs, but that aspect is not part of this work.

[1] http://www.worldofwarcraft.com/
[2] http://www.worldofwarcraft.com/info/faq/realms.html
[3] http://www.eveonline.com/
[4] http://secondlife.com/

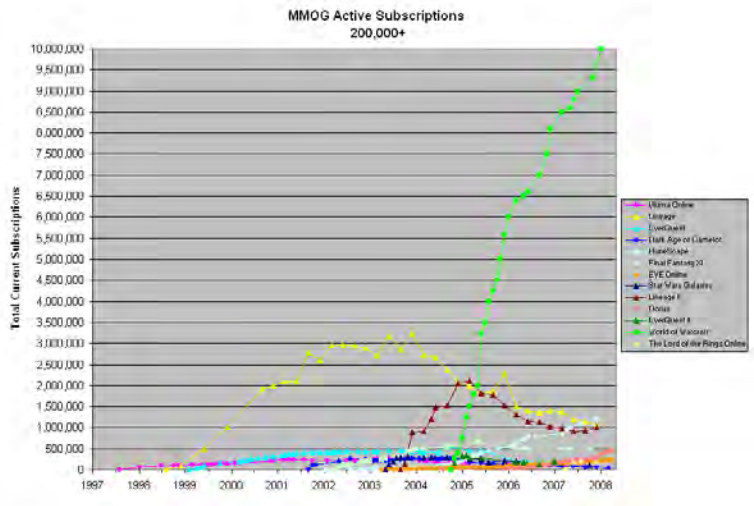

Figure 1.1: MMOG Active Subscriptions [38]

1.1 Planet π4

Planet π4 [37] is a simple multiplayer first person shooter game. The player navigates his spaceship in a 3D space, shooting 'energy missiles' at his opponents. Each player initially has 20 life points which are decreased by one every time his ship is hit by a missile. When the number of life points has reached zero, the player is 'dead', i.e., his ship is reset and he may immediately resume the game.

In this work, Planet π4 is used as the base game. Being a very simple game, it is relatively easy to port the network layer to different systems. But still, it has a recent 3D graphics engine which yields a high potential for making it attractive to test players. This is necessary for obtaining realistic information about the players' behaviors.

The game was developed by Tonio Triebel for research purposes at the Universität Mannheim. It has already been ported to a few different (peer-to-peer) networking systems. The first version used IP broadcast in a local network. Then, there is a version which uses Skype[5] chats for game communication [36]. Skype provides an API[6] allowing applications to access its communication functionality. The advantage of this approach is that Skype, being a peer-to-peer system, already manages all the 'nasty peer-to-peer work' including NAT traversal. One of the downsides is the limited group size for chats which is currently 25. Thus, a region in the game cannot handle more than 25 players. The third Planet π4 network implementation uses SpoVNet[7] [6], which provides an application layer multicast service.

The Planet π4 network model is based on (multicast) groups which refer to regions in the game world. Each player sees all other players in his group, i.e., game information like position updates are broadcast within each group. A player may be in multiple groups at the same time, giving the possibility of having overlapping regions to avoid disconnects when moving from one region to another.

Being a research prototype, the game is still under development for different aspects. The current implementation of the game does not yet support regions, thus the different groups are actually different game universes. There is no possibility to move from one region to another without leaving the game. Also, the snapshot version used for this work is still missing some important features like collision detection for spaceships (hitting each other) and the ground.

[5] http://skype.com/
[6] http://developer.skype.com/
[7] http://spovnet.de/

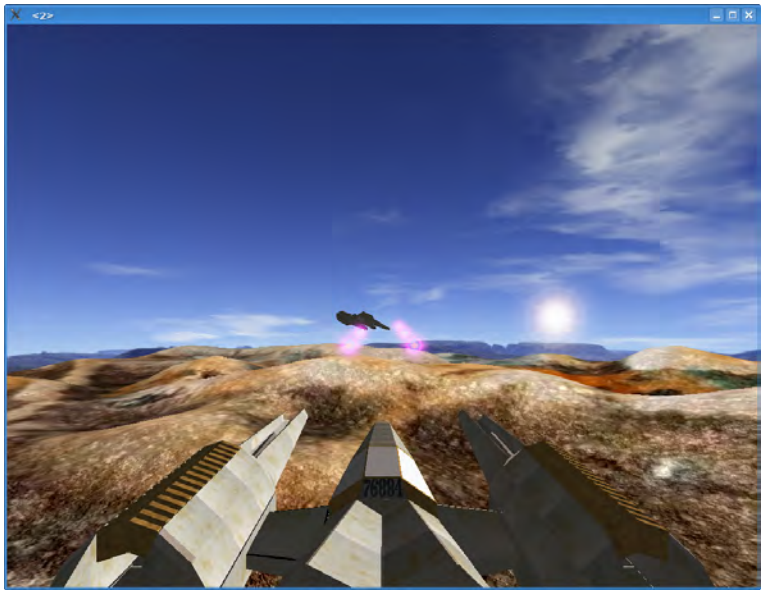

Figure 1.2: Screenshot of the original game Planetπ4

1.2 Glossary

This work contains several terms that may not be fully known to readers who are not particularly familiar with the topic of online games in the context of distributed systems. Additionally, some terms have several synonyms that are used alternately, partly even in this thesis. This brief glossary is intended to clarify at least some of these.

1.2.1 Gaming Terms

MMOG, MMORPG stand for **M**assively **M**ultiplayer **O**nline (**R**ole-**P**laying) **G**ame. As initially described, massively multiplayer online games are games in which a large number of players (typically a hundred up to several ten thousands) play together in one big game world. MMORPGs are currently the most common type of MMOGs; see role play games (RPG) below.

NVE means **N**etworked **V**irtual **E**nvironment, i.e., a virtual world in which participants cooperate over a network. Network multiplayer games (including MMOGs) are the most popular NVEs, and also SecondLife[8], which is not a typical game, is an NVE. But there are also 'serious' NVE application like virtual conference rooms.

RPG, RTS, FPS, CTF are different game types and gameplay modes respectively.

In **R**ole-**P**laying **G**ames (RPG), each player has a character, typically a humanoid or some fantasy figure, which has certain skills. The player interacting with other players (on online games) or NPCs in the role of their characters, thereby solving quests and improving their (virtual) skills. Computer role-playing games (to which the term RPG refers to in this thesis) are originally based on the older pen-and-paper role-playing games like *Dungeons & Dragons*. Some popular examples of computer RPGs are *Diablo*[9], *Neverwinter Nights*[10] and *Baldur's Gate*[11].

[8] http://secondlife.com/
[9] http://www.blizzard.com/us/diablo/
[10] http://nwn.bioware.com/
[11] http://www.bioware.com/games/baldurs_gate/

Real-**T**ime **S**trategy (RTS) games typically simulate military conflicts. The player controls one war party fighting with its army against one or more enemy parties. In many cases the player first has to build up his army by harvesting natural resources and building an colony. The army's units can either be controlled separately or in groups. The world is shown to the player in a bird perspective so that he can in principle overlook the whole world, except for the often used 'fog of war' which hides any area that is not nearby a friendly unit. Exemplary RTS are the *Warcraft* series[12], *Age of Empires*[13] and *StarCraft*[14].

In **F**irst-**P**erson **S**hooters (FPS) the player moves his avatar in a 3D game world and tries to shoot his enemies (which are other players in multiplayer games). The name of this game type originates from the player's perspective, seeing the world from the eyes of his avatar. The avatar is in most cases a humanoid fighting with range weapons in a war scenario. In a broader context, FPS also include any other kind of shooting game which the player perceives from a first person perspective, e.g., a 3D space shooter. Typical examples are Doom[15], Quake[16] and Halflife[17]/Counterstrike.

Capture **T**he **F**lag (CTF) is a special gameplay mode in multiplayer FPS or RTS. There are two teams, each possessing a base with a flag installed. Each team tries to pick up the flag from the opposing team's base and bring it to its own base. Every time this succeeds, the corresponding team gains a point. When a player carrying the flag is killed, he loses the flag and it can be picked up by another player.

NPC stands for **N**on **P**layer **C**haracter and denotes an in-game character in an RPG that is not controlled by a human player. Each NPC has some pre-programmed behavior (artificial intelligence, AI) that takes action when some event is triggered by a human player. NPCs are most common in single player RPG where all characters except for the player's are NPCs. But also in multiplayer RPGs, NPCs are commonly used for traders on the market place or just for any kind of non-human enemy.

Bot is a player in a multiplayer game that has the same rules and abilities as a human player but is controlled by a computer program providing a certain kind of artificial intelligence (AI). Bots are most common in FPS where they serve as team members or opponents, e.g., when there is a shortage of human players in the game.

Shard, Realm are synonyms for a server instance running the game world of an MMOG. No MMOG so far allows an unlimited number of players to play in one world; the size of a shard is limited to a few hundred up to several ten thousand players, depending on the game and the server capabilities. Millions of concurrent players are only feasible with multiple shards. Shards represent completely independent 'parallel universes', i.e., there is no potential for interaction between shards. When joining the game, a player has to choose or is assigned to a shard to play on.

1.2.2 Technical Terms

Node, Peer, User, Player are often used as synonyms depending on the context. A *node* in distributed systems is typically one machine or application instance that is connected to other instances over a network. A *peer* is a node participating a peer-to-peer network. The *user* (which in case of games is the *player*) is the person who uses the application/game. But in some cases these two terms are also used instead of node/peer, indicating that the latter are representing a user/player in the system.

[12] http://www.blizzard.com/war3/
[13] http://www.microsoft.com/games/empires/
[14] http://www.blizzard.com/starcraft/
[15] http://www.activision.com/index.html#gamepage|en_US|gameId:Doom3&brandId:DOOM
[16] http://www.idsoftware.com/games/quake/quake/
[17] http://www.half-life.com/

Avatar is the virtual representation of the player in a game. This term is used in FPS and RPG where the player identifies with a single virtual 'person'.

Region, Zone is a certain area of the game world in games whose worlds are split into disjoint parts. Regions may separate the game world in a way that is visible to the player, e.g., by the need for stepping through portals to get to another region. But region borders may also be invisible to the player with the transition being transparently handled by the game engine. This approach is used when a large continuous world is to be simulated that cannot be handled by a single server. In such case, regions are typically rectangular or hexagonal so that they can be efficiently arranged without any gaps.

Vision Range, Area of Interest (AOI), Interaction Range An important property of FPS and RPG is that the player can only perceive a limited area around his avatar which is called the *vision range*. The area within the vision range is the *area of interest* (AOI). In order to be able interact with other players or objects, it is necessary to be close to the corresponding player/object. The maximum distance that allows interaction is the *interaction range*, which is usually (depending on the game) somewhat smaller that the vision range.

Besides the effects on game mechanics, these properties are especially important for networked games. Each player only needs to receive information about changes in the game world within its vision range. The technique of minimizing the information (and thus amount of data) a player gets is called *interest management* [24]. In client-server games, this is typically performed on the server, thus minimizing the clients' bandwidth requirements.

Dead Reckoning is a technique that tries to eliminate the visibility of networking effects (delays and limited update frequencies) by predicting the remote players' actions in the time interval between receiving updates. A simple approach is to keep moving the avatars at the current speed in the current direction. This may be improved using hints that are sent with the update messages allowing a more precise prediction. Sophisticated approaches use bot-like artificial intelligence to achieve even more realistic behavior.

2 Related Work

This chapter introduces the scientific topics relevant for this work, summarizing related work. First, basic peer-to-peer concepts are introduced, followed by more advanced systems that are possibly providing services usable for peer-to-peer games. The largest part of the chapter describes several peer-to-peer systems specifically designed for (massively) multiplayer games.

2.1 Peer-to-Peer Infrastructure

This section introduces basic concepts of peer-to-peer systems and presents the most important systems.

2.1.1 Distributed Hash Tables

Since the late 1990s, there have been many approaches for organizing distributed systems in a way that there is no need for a global authority required to run the system. These approaches are generally known as peer-to-peer systems. One of the early systems is Gnutella [21], a file sharing network in which each participating node is connected to one or more other nodes in the system. Doing so, the nodes build an overlay network on the Internet. When a user searches for a file (based on keywords), the network is flooded with the query as each node forwards the message to its neighbors. Nodes having matching files contact the originator and the file transfer can be initiated.

When the networks grew larger, flooding the network with queries soon became inefficient. As a result, starting in 2001, several new approaches for locating objects in a fully distributed network were published, known as *distributed hash tables* (DHT). The early and most important systems are Chord [32], CAN [27], Pastry [28], Tapestry [39], and Kademlia [23]. Their common purpose is to provide a service for efficiently storing (or routing) objects and object lookup, based on keys. Keys are typically 128- or 160-bit identifiers, generated via hash functions like SHA-1. This interface motivates the name 'distributed hash table'. The complexity (in overlay hops) of store and lookup, or generally message routing, is typically $O(log(n))$ (Chord, Pastry, Tapestry, Kademlia) or $O(dn^{1/d})$ (CAN with d dimensions) with n being the total number of nodes in the system.

In a DHT's overlay network, each node has a unique ID (128 or 160 bit). The mechanism for determining which pairs of nodes need to be connected varies between the different approaches. In the simplest case of Chord, it is a ring with the nodes ordered by their IDs. Depending on the system, the connections may be more or less randomly chosen. All systems route messages based on their destination IDs. It is not necessary that the destination ID matches a node's ID but each message is routed the the node with an ID 'closest' to the destination ID (details defining 'closest' vary slightly between the different DHTs). Using this mechanism, objects can be deterministically stored and retrieved given their ID.

The introduction of DHTs brought a great hype to peer-to-peer systems as they allow for large networks (millions of peers) working autonomously using normal Internet connections. A widespread area of applications based on this technology has developed, some of them are file sharing, communication (e.g., Skype[1]) and live video streaming (e.g., Zattoo[2]). Also, there has been some research on peer-to-peer games based on DHTs (more to come later in this chapter).

But DHTs also have some weaknesses. Their structured topology needs to have a certain integrity allowing them to work properly. Although some systems have already shown to be functional in real

[1] http://www.skype.com
[2] http://zattoo.com

scenarios with millions of users, there is no formal proof on how reliable they are on intermittent or even catastrophic network failures. And second, though many problems can be formulated using the hash table interface, there are still many problems that cannot be solved (efficiently) that way. One example is full text search in text documents that are distributed in the network. For peer-to-peer games there are some other issues, especially concerning performance, that make DHTs suboptimal as the communication infrastructure of fast-paced games.

2.1.2 BubbleStorm

Addressing some of the issues of DHTs mentioned before, BubbleStorm [34] has been developed. BubbleStorm is a unstructured peer-to-peer system providing exhaustive search in a probabilistic fashion. Unstructured means that the peers are not connected following some system of routing tables, but completely random, thus building a random graph. Unlike in DHTs, nodes do not have IDs nor is data routed to certain deterministic places. BubbleStorm distributes both data and queries randomly in the network. With a certain probability, which depends on the replication factor and can be predetermined, a query and a matching date meet at some node (the *rendezvous node*) which then can respond to the query. This allows for complex queries like full-text search, as the rendezvous node has the full document to be searched on.

Algorithm

BubbleStorm's topology is a random multigraph, allowing self-loops and multi-edges. Nodes choose the number of connections (node degree) based on their bandwidth capabilities; each edge has about the same traffic because of the random selection of forwarding edges. To keep existing nodes' degrees unchanged when nodes join, the new nodes connect between pairs of previously connected nodes, so that both existing nodes do not need an additional connection. Accordingly, leaving nodes re-connect their neighbors. The first node of a network starts with a self-loop. Desired node degrees have to be multiples of two, but single broken connections are not repaired. Only when there are two of them, two new connections are created using the join procedure. Before becoming a peer, nodes join as clients. Clients are served through their connecting peer but do not provide any services to the network. After some time, clients become peers. This reduces churn and protocol overhead with short-lived nodes, which in this system never need to become peers.

To achieve a certain rendezvous probability, data and queries have to be spread in the network with a certain density. Thus, for calculating the number of necessary replicas, it is necessary to know the size of the network. Therefore, BubbleStorm runs a protocol for measuring global system state, including the number of nodes, the sum of node degrees and the sum of squared node degree of all nodes in the network. The fully decentralized gossip protocol [35] needs $O(log(n))$ rounds to calculate the necessary values and is run constantly to keep information up-to-date.

For BubbleStorm, it is necessary to control the precise number of replicas of data and queries. This is done efficiently by *bubblecasts*. Bubblecasts combine random walks with flooding. While flooding is fast with an exponentially growing number of reached nodes, the total number of reached nodes can only be controlled inaccurately via the messages' time-to-live value. In contrast to that, bubblecasts forward each message only to a fixed number (the bubblecast split factor; default: 2) of neighbors (random subset) in each hop. The desired number of replicas (the bubble size) is specified as the *weight* of the bubblecast messages. On each hop, the forwarding node decreases the weight by 1 (for the one copy he received) and distributes the remaining weight over the forwarded messages (as long as there is remaining weight).

To ensure that a datum-query-pair meets at at least one node with a given probability, the bubble sizes must be set accordingly. The BubbleStorm paper derives the formula $qd = c^2 n$, where q and d

are the query and datum bubble sizes, c is the reliability factor, and n is the size of the network. c is defined as $\sqrt{-ln(1-r)}$ with r being the success probability. A typical value of c would be 3, resulting in $r = 99.99\%$. The ratio of q and d does not affect reliability and is adjusted to minimize network traffic, depending on the data rate ratio of data and queries.

Evaluation

The BubbleStorm algorithm is evaluated using a custom simulator which provides messaging over simplified TCP connections. The scenario is a distributed wiki with as much as 1 million participants (peers) storing articles' metadata (2kb size) and searching for articles. The nodes' connection capacities are limited to 10kB/s for upstream and 100kB/s for downstream with a 40ms local link delay; their lifetime distribution is exponential with a mean of 60 minutes. The certainty factor (c) is set to 2, resulting in a rather low but better measurable reliability (98.2%) than one would use for real applications.

For each scenario, the network is exponentially grown to steady size (1 million nodes) using normal join operations. The warmup phase takes 3 minutes of simulated time. After 1 minute of steady state, different (massive) node failures are simulated. These include normal churn (reference case), massive leave (50% and 90%), churn with crashes (10% of leaves are crashes), and massive crash (5%, 10%, and 50% nodes die).

The normal churn scenario shows the expected query success rate of 98% with an average of 4.5 seconds for a search to complete (most successful searches within 2 seconds). Churn measurements with 10% crashes have very similar results, with no significant drops. The 50% leave scenario shows a slight bump (below 10%) in query latencies. The query reliability goes up to 100%, caused by the temporarily overestimated network size until the measurement protocol delivers a new value. With 90% leaves, the latency bump is much higher because of the high congestion caused by the leave operations. The 100% reliability phase is followed by a reduced reliability (96%) phase. This is because the measurement protocol underestimates the network size in the first round after the leave. When a large fraction of the nodes dies, there is a breakdown of query reliability for a few seconds. With 50% of the nodes crashing, the reliability reaches zero, but in all cases recovers quickly.

Additionally, a heterogeneous scenario is evaluated, where nodes have different bandwidths between ADSL-1000 and symmetric 10Mbit/s. The nodes select their degrees proportional to their bandwidths. BubbleStorm profits from a heterogeneous network requiring smaller bubbles and thus reducing total traffic. For details, refer to the original paper [34].

The BubbleStorm implementation provides a transport protocol based on UDP that can also be directly used by the application. The protocol is described in 2.5.2.

2.2 Application Layer Multicast

An important requirement of peer-to-peer gaming systems is an efficient multicast system. This is primarily needed for disseminating players' position updates. One possible approach is to use a general-purpose application layer multicast system that supports groups, mapping regions of the game world to groups. This section introduces one exemplary application layer multicast system.

2.2.1 Census

Census [8] was recently published and thus can be seen as a rather advanced and up-to-date approach. Census claims to be the first system providing a consistent membership abstraction at very large scale, being resilient to crashes and Byzantine failures. The consistent membership view allows the efficient distributed building of deterministic distribution trees based on network proximity.

Architecture

Census operates in *epochs*, which are fixed-length time intervals (30s is a typical length). Each epoch has a fixed and consistent membership view which only changes on epoch transitions. Also, each epoch has one member that acts as the leader, multicasting an *item* informing about membership changes at the end of the epoch. Within each epoch, application data can be multicast according to the group memberships. Multicast trees are built based on a network coordinate system like Vivaldi [9]. The authors present three versions of the algorithm, starting with a simplified single-region setup (scaling to more than 10,000 nodes), then introducing the multi-region system and finally adding the partial knowledge extension for particularly large or dynamic systems.

In the first version, there is a single leader processing all membership events. A joining node identifies itself to the leader and provides its network coordinates (for network proximity management) and an optional identity certificate. The leader informs the joining node about the current epoch number and a few members from whose the latter obtains the current membership information. When leaving, a node sends a departure request; failed nodes are detected by others. Nodes report major changes of their network location, enabling relocations for continuous network proximity.

For larger deployments, the system is divided into multiple regions based on network proximity. Each region has a region leader which is responsible for membership activities within the region. Towards the end of an epoch, each region leader sends a report for its region to the global leader who then creates the item for the next epoch. When a node's network coordinates change, it may move to another region by sending a *move* request to the region leader. The number of regions is not fixed. Instead, the system starts with one region, and whenever a region size has reached a certain threshold, it is split. If a region shrinks below a certain threshold, it merges with a neighboring region.

Even with multiple regions, each node still has the full membership view. This effects a high bandwidth requirement for all participating nodes particularly in very large and/or dynamic (i.e., memberships change frequently) systems. To solve this problem, there is a *partial knowledge* deployment option in which participants only have a full membership view for their own region but only summary information about the other regions. The summary includes the region leader, the region's size and centroid, and some region members.

Multicast

Multicast is is based on multiple distribution trees which are built deterministically and on-the-fly using the consistent membership view. Thus, each node computes where to forward messages to, using a deterministic algorithm. Doing so, the tree is rebuilt every epoch, independent from the previous one. Multiple trees allow for redundancy using m-of-n erasure-coded fragments, i.e., there are n trees (typically 4 to 16) and a message can be reconstructed if at least m fragments have been received. If more than $n - m$ fragments are missing, a node requests missing fragments from random nodes of his membership view.

The trees within each region are *interior-node-disjoint*, i.e., each node is an interior node of at most one tree. This ensures that failed nodes cannot break more than one distribution tree. Trees are recursively built based on node locality. In each step, the current sub-region is split through the centroid across the longest edge, choosing one node on each side as a child. Before building the trees, all nodes are colored (round robin), and one tree is built per color, using only nodes with that color as interior nodes. With multiple regions, inter-region trees are built the same way, just containing only one representative of each color in each region.

Bandwidth requirement is minimized by letting only one parent send the membership update for a particular child, and only m parents send a fragment. Latency is optimized using the membership knowledge. A fragment is only sent if it is on one of the m fastest paths to the child.

Discussion

Census' most important feature is the consistent membership view of all members. But this feature causes a rather high bandwidth and memory consumption on all members. Although peer-to-peer MMOGs do not necessarily have several 10,000 players, they are though highly dynamic. Players continuously move in the game world and thus group memberships tend to change frequently. Also, Census exploits network proximity which allows for low latency distribution. Systems specifically designed for games (see next section) rather concentrate on game world proximity because the highest traffic is usually between nearby nodes in the game world. Finally, membership operations in Census take a whole epoch (typically 30s) to complete. This is by far too slow at least for fast-paced games.

2.3 Peer-to-Peer Gaming

This section provides an overview of the recent research publications in the topic Peer-to-Peer gaming. Here, the different approaches are described separately, while the next section tries to combine them defining a generic design space for Peer-to-Peer MMOGs.

Peer-to-peer games in general are not particularly new. Since the early days of networking, there have been networked multiplayer games. But these games always were very limited in the number of players; some recent games support up to 32 or even 64 players. Basically, there are two alternate 'simple' approaches that can be taken in these multiplayer games:

Single master. One machine is chosen to be the game master. This is typically done explicitly by the players; the one who creates the game acts as the master, everyone else connects as a slave. This approach relates to client-server, where the server (master) quickly becomes the bottleneck, either because of a lack of processing power or network bandwidth.

Full replication. Every machine executes the whole game logic and only the actions of each player are exchanged. Although this distributes the load very well, it may be hard to efficiently synchronize the game mechanics on all machines and avoid 'alternate realities', i.e., inconsistent views of the game world. And the game world (including game mechanics calculations) can only be as complex as supported by the weakest machine in the network.

Both techniques obviously do not scale with limited bandwidth and/or processing capabilities. The approaches presented in the following try to overcome these limitations by exploiting special properties of games like locality of interest (effected by limited vision ranges).

2.3.1 SimMud

In 2004, Knutsson et al. published a paper [20] presenting an architecture for Peer-to-Peer massively multiplayer games based on a DHT. For evaluation purposes they implemented the prototype game *SimMud*.

The authors point out three main challenges that need to be considered when developing Peer-to-Peer games: 1. Performance. There is the need for frequent and low latency updates with a low network capacity. 2. Availability. Leaving and failing nodes must be compensated. 3. Security. Prevent cheating and account thefts. As for most of the papers related to this work, the topic of security is identified as a separate issue and not discussed any further.

The main concept introduced by the paper is the distribution of the transient game state among all participating players while the persistent state like account information and character experience is kept on a central server. This allows distributing the workload but still being able to keep control of the

administrative data. For short gaming sessions where persistent storage is not necessary, servers may be completely omitted.

Game State

The paper's target games are role-playing games (RPG) like EverQuest and Ultima Online. Game state in those games consists of three types of data. First, there is the terrain, i.e., the immutable part of the game world. It does not need to be updated and is usuallay installed with the game client. Second, there are mutable objects like items that players may interact with or NPCs (non player characters). These objects have some state and are associated with game rules. And finally, there are the player's avatars. Actions of players are categorized as position change, player-object interaction and player-player interaction.

The world is split into regions whose size is based on the sensing range of the players. Doing so, interest management is applied by distributing information only within a region. Thus, each region forms an interest group. Player's positions are multicast at a fixed interval to all other players within the region. Dead reckoning [1] may be applied on application layer when messages are lost or delayed. Each object (e.g., item) is assigned to a coordinator, a node that is responsible for resolving conflicts when the object is manipulated. Updates to objects are multicast to all players in the containing region.

Distribution on a Peer-to-Peer Overlay

The system is based on the Pastry DHT. Each region is assigned an ID and mapped into the DHT. The node responsible for a region's ID is responsible for that region. To archive a better load balancing when certain regions are under heavy load, the authors propose to create a different region IDs for each object type. Cheating is prevented by the (pseudo-)random mapping of regions to coordinators, as this makes it unlikely that the person, who is interested in illegally manipulating an object, also has control of it.

Fault tolerance is achieved by replicating the game state objects over several nodes, assuming that node failures are independent and rather infrequent and that the pastry routing algorithm itself will tolerate failures, always routing messages to the proper destination. The basic goal is to keep always one replica up besides the primary copy. The primary copy of an object (whose owning node is the coordinator for that object) is on the node closest to the object's ID. The replica is stored at the node with second closest ID, so that in case of a failure of the primary copy, the replica can immediately take over. When a new node joins which has an ID that is closer to an object than the node that has the primary copy, the object is replicated to the new node. While replicating, the previously primary node forwards all messages related to the object to the new node, which afterwards applies all updates and takes over.

Failures are detected using regular game events instead of explicitly checking for peers for being alive. The authors state that there is a window of vulnerability between the time of a failure and its detection and recovery. But they claim the problem of network partitions is worse. Network partition means that there are two (or more) coordinators for the same object, immediately leading to inconsistencies. The authors citate Brewer's 'CAP conjecture' [14], stating that a distributed system can only have two of the tree properties consistency, availability and tolerance of network partitions. The authors' proposed solution is to 'bless' all coordinators by a central server who knows of all coordinators, thus sacrificing availability.

Implementation

The implementation for evaluation of the presented system is *SimMud*, a Java application using FreePastry, simulating a simple world with food objects and players who can move, eat and fight against each

other. To avoid cheating in player-player interaction (fighting), the identical algorithm is executed on all participating parties with the same input, and the results are compared afterwards.

For object updates, messages contain both the old and the new value, allowing the coordinator to check for conflicts. If the old value does not match, i.e., the value has been modified in between, the coordinator rejects the update. Where necessary, it is also possible to use version numbers.

Experimental Results

For evaluation, up to 4000 nodes are run in a simulated FreePastry environment. The players are controlled using a simple model that is supposed to be similar but more aggressive than the players' behavior in RPGs like EverQuest. The map grid has a size of 200x300 tiles (i.e., valid player positions), position and object updates are about 200 bytes each. Position updates are sent every 150ms.

In a 300-second simulated game with an average of 10 players per region and a link delay between 3 and 100ms, each node receives 50 to 120 messages per second. A breakdown of message types shows that more than 99% of all messages are position updates; object updates and player-player interaction messages are relatively rare. Message delays are mostly below 200ms, with a maximum of 400ms. Comparing runs with 1000 and with 4000 nodes, there is only a slight difference in delay by the slightly increased average number of hops.

While the total number of players does not have a substantial effect on the performance directly, the message rate grows linearly with the player density, i.e., the number of players per region. Thus, it may be necessary to limit the regions' capacity by the game mechanics. The proposal for reducing the position update message load is to aggregate the messages at the root of the multicast tree, but this will not reduce the load of the root itself.

With nodes joining and leaving (i.e., failing) at a rate of 0.06 per minute (average session length of 16.7 minutes), the average message rate increases from 24.12 to 24.52, caused by the necessary object relocations. There is the possibility that region data is completely lost when a node fails in the window of vulnerability (during replication after a node has failed). With a window of vulnerability of 10 seconds and a node fail rate of 0.06 per minute, such a 'catastrophic failure' would happen once every 16 minutes. With a longer session time of 2.3 hours, this value would be reduced to once per 20 hours.

Discussion

SimMud is one of the early peer-to-peer game implementations, concentrating on rather slow games like EverQuest, which was very popular at that time. Slow in this context means that it is applicable for adventures and/or role-playing games that do not have strong timing constraints. Many of the system's performance and robustness properties depend on the underlaying DHT's features. This has the advantage that it can make use of the DHT's features that are already known and evaluated. On the other hand, DHTs are not optimized for gaming requirements, especially concerning realtime capabilities.

The implementation presented here only creates one additional replica for each object, resulting in long windows of vulnerability. Keeping more replicas would heavily decrease the probability of catastrophic failures, i.e., data loss. Also, the paper lacks an explanation on how the replicas are kept up-to-date.

The approach presented in the paper is yet not purely peer-to-peer. Having a central server for user management may be desirable also for non-technical reasons. But using the central server for 'blessing' coordinators is definitely reducing scalability. Finally, the paper deals with some cheating issues, but its solutions are rather simple and certainly not satisfactory for a real game.

2.3.2 Hydra

Hydra [5] is a peer-to-peer game framework developed by Chan et al. and published in 2007. Its primary goal is to enable game development without the need for coping with peer-to-peer-specific techniques.

Architecture

The game world is divided into disjoint regions (*slices* in Hydra terminology), each managed by a region server. Player transitions between the regions has to be managed by the game application. In general, Hydra only provides messaging between nodes but no connections.

The game client interface provides methods for sending reliable and unreliable messages over UDP. The message order of reliable messages is retained, but that is not the case for unreliable messages. Addressing is done via IDs, not IP addresses. The server interface provides a queue for incoming messages, in which the messages are sorted using a timestamp, Hydra's *tick* value.

There are certain requirements for the game server engine: It is the server's task to process incoming messages up to the current timestamp (i.e., for the whole frame) and afterwards to send response messages. These responses are not allowed to be sent before all incoming messages for the current frame have been processed. Also, the simulation has to be deterministic, thus random numbers have to be pseudo random, generated using a fixed (or deterministic) seed. Finally, there is a checkpointing interface that the server has to implement in order to be able to consistently save and restore game state at certain points.

A game's client module consists of a normal (non-peer-to-peer-aware) game client and a client proxy connecting to the Hydra network. A global tracker acts as an entry point for new clients. The Hydra servers each serve one or multiple slices, addressed by unique *sliceIDs*. For each region there is a primary copy and a number of backup slices. The primary slice reliably forwards all received messages to the backup slices, which thus can run the game engine with identical information. Failure detection among primary and backup slices is done with a timeout check on the reliable channel between them. If the primary slice fails, a leader election selects the backup slices that will take over. The new primary slice continues with the tick count at which the old primary slice would have been at that moment. Unreliable messages may get lost, while reliable messages in the backup queue are still processed. When necessary, the primary slice creates new backup slices by replicating their game state to another machine.

All server replication and failure handling is accomplished transparently to the actual game server part which only implements the server interface. Client failures have to be handled by the game application.

Evaluation

For evaluation, a simple game, *Tankie* (capture-the-flag gameplay), was run with bots on 15 machines in a LAN. In a 14-minute session, clients join incrementally, a backup slice is created, the primary slice is killed and after take-over, an additional backup slice is created. As expected, the number of received messages of clients grows about linearly with the number of nodes, while the primary slice's sent messages grow in square. The average response time is about 300ms, but there is a peak of several seconds at the point where the primary slice is killed.

Discussion

Hydra provides a clean and simple interface making it relatively easy to develop decentralized network games. But there are still several specialties and constraints that the developer has to deal with to make the game work properly, for instance the checkpointing support.

Strictly distinguishing between clients and servers, Hydra is rather a dynamic server clustering system than a full peer-to-peer system. The load is distributed amongst multiple servers and the clients do not contribute to the system. Although each client could also run a server instance, there are probably not enough slices to generate work for everyone since slices cannot be allocated dynamically. Also, Hydra does not provide load balancing; the authors mention this as future work. Finally, the transit between slices is to be managed by the game application, not allowing for continuous game worlds where players do not notice the transition to another slice.

2.3.3 Colyseus

Colyseus was published in 2006 as a distributed architecture for online multiplayer games by A. Bharambe et al. [4]. The system builds on top of (different kinds of) DHTs for data lookup. In order to overcome slow DHT lookups, it introduces a caching mechanism which allows every node to create read-only replicas of an object. Modification of an object is synchronized by the node that stores the primary copy. Keeping local replicas where necessary only guarantees weak consistency, but ideally each node may anticipate which objects will be needed in near future and perform pre-fetching.

The main purpose of Colyseus is to act as a manager for mutable game objects. These can be avatars, NPCs, doors or items. Associated with the objects there are also their 'think functions' (i.e., the functions actively processing the object's state) that are distributed amongst the participating nodes.

Replication

As mentioned initially, there is always a primary copy of each object's state. Each replica is synchronized with the primary copy in an application-dependant way. This synchronization is stated to be the same as in client-server architectures. The objects' think functions are each executed on node that store the primary copy. The functions may need remote objects which are also locally replicated; those objects also have to be predicted in order to be pre-fetched by Colyseus.

Replicas are updated on each frame using delta messages. Modifications to the primary copy are transmitted via delta messages, too. When a node stores a replica of an object, it communicates directly with to node storing the primary copy instead of using the location service. This allows for low-delay updates.

Proactive replication is supported via a function that allows to *attach* to another object in order to be automatically replicated to all nodes interested to the latter object. The authors' example where this functionality is useful is launching a missile; the missile attaches itself to the attacker so that everyone nearby immediately can see the missile without a time-consuming lookup.

When a replica is modified, the changes are tentative. The modification is forwarded to the node holding the primary copy which serializes all incoming modifications and finally decides what changes are applied to the object. This optimistic consistency model aims at fast-paced games where it is more important to have low-delay updates than having strong limits on staleness.

Location

Although the object lookup service is based on a DHT, it provides range queries for areas of interest, called *subscriptions*. Each object makes its location public via *publications*.

Colyseus implements the range query feature both on a traditional DHT and a range-queriable DHT, comparing their performance. For both DHTs, Colyseus uses an implementation of Mercury [2]. Using the traditional DHT, the map is split up into regions. Range queries are accomplished by querying each region within the range separately. Using the range-queriable DHT, the x-dimension is mapped to the

DHT's key space. A comparison of the two DHTs shows that the range-queriable DHT has a slightly higher latency but a lower bandwidth consumption than the traditional DHT. Additionally, the former has a better load balancing.

The discovery latency is reduced using the spatial and temporal properties of the game. Players (usually) move more or less constantly within the game world. Using this property, Colyseus automatically expands the area of interest towards the direction of movement, performing speculative pre-fetching of objects. The amount of expansion depends on the player's velocity, always pre-fetching for a certain amount of time. This amount of time is a parameter which needs to be adjusted to be high enough to have an acceptable view consistency but not too high to avoid a lot of unnecessary pre-fetching (e.g., caused by the player changing direction). For further optimization, Colyseus implements a local subscription cache to avoid too frequent subscription updates and an optional aggregation filter merging overlapping subscriptions. Furthermore, not only the subscriptions but also the publications are stored in the DHT for a certain time. This allows for immediate responses to new subscriptions without a delay till the next publication, also making longer publication intervals possible.

Implementation and Evaluation

For evaluation with a real game, Quake II was modified to use Colyseus. Alternatively, unmodified Quake II clients may connect to the distributed servers so that the system may either be used as a full peer-to-peer game or using a distributed server community. The Quake II objects are sent using field-wise diffs, resulting in an average diff size of 145 bytes. The original Quake II server-client messages have an average size of only 22 bytes.

In small scales, the generated traffic with colyseus is higher than the traffic for simple broadcasting. But with an increasing number of nodes, Colyseus quickly outperforms broadcasting, and the traffic load of each node is an order of magnitude lower than the server's load in client-server mode. The analysis of view consistency shows that half the replicas take less than 100ms to update and only 1% takes more than 400ms. Of the 23 remote replicas a node requires in average for a 40-player game, it has the complete set in about 40% of the time. After waiting up to 100ms for a replica, 60% of the replicas are available in avarage, and waiting up to 400ms increases the proportion of available replicas to 80%.

Discussion

Colyseus focuses on object management for fast first person shooter games, introducing techniques compensating the problem of relatively slow DHT operations. Performing optimistic updates, it also aims for low delays, providing only weak consistency. As the system relies on spatial and temporal locality, e.g., for its prefetching mechanisms, it is only suitable for games where players have a limited range of perception and interaction. Real-time strategy games (RTS), for instance, usually require the player to see large parts of the game map, in which case Colyseus' mechanisms will not work.

For first person shoothers though, Colyseus shows a good performance. Being able to run in full peer-to-peer mode or as a cluster of distributed servers, the system is flexible for different purposes. The evaluation using a modification of an existing game can be taken seriously, but lacks a user study.

2.3.4 Donnybrook

Donnybrook [3] was published by Bharambe et al. in 2008, and thus may be seen as Colyseus' successor. But in fact, it has a different focus. While Colyseus deals with object management, Donnybrook concentrates on player update management. It continues the optimization of the players' interests, trying

to prioritize based on the importance of the various objects for the player. Additionally, it introduces a multicast system specialized for games.

According to the paper, the key challenges in peer-to-peer MMOGs are the following: 1. Nodes have a limited connection bandwith, especially for upstream, as most end users have asymmetric Internet connections (ADSL or cable). The communication demand generally grows quadratically with the number of players (in area of interest). 2. The heterogeneity of resources, i.e., the nodes in the system have heavily varying processing and connection capacities. 3. The heterogeneity of interest, i.e., in the game, there are some players that are of interest to many others while not having a high capacity. This may be the case e.g., for players wearing the flag in capture-the-flag game modes.

Interest Management

Donnybrook's interest management is based on *interest sets*. Each player has its own interest set containing other players that he is most interested in. This set is estimated by the game client, leveraging the limitation of human recognition by setting the size of interest sets to the constant number of five players. This number deduces from studies about human recognition and a user study initiated by the authors of this paper. The players who are in the interest set are updated in a high frequency, while the other (nearby) players are represented by *low-fidelity replicas*.

A player's interest set is build (and frequently updated) by assigning attention values to each other player. There are three metrics affecting the attention value for a particular player:

Proximity. Nearby players are more important and players outside a certain range are not visible at all. Thus, the value is the normalized distance within the vision range, raised to the power of 1.5. The argument for the power of 1.5 is that in a 3D space, the visible area of an object decreases in square with the distance, but a typical first person shooter game world is typically somewhere in between a 2D and a 3D space.

Aim. The players who are aimed at are with high probability of interest. This metric bases on the *aim deviation* which is the angle between the user's screen center and the particular object (within 45° in each direction). Considering distance, this value is raised to the power of 1.5 times the distance to the object, as farther objects need to be aimed more precisely.

Interaction Recency. Within 3 seconds after the last interaction, this metric decays exponentially, considering the importance of players that have been interacted with (e.g., shot at) recently.

The weighted sum of these three metrics is the attention value, sorted by which the top five players are elected into the interest set. The weight factors need to be tuned depending on the particular game. When a player enters the interest set, he gets a *subscribe* message in order to initiate high frequency updates. After leaving the interest set, he gets an *unsubscribe* message.

The players who are in the vision range but not in the interest set are updated about once per second. To make them look naturally instead of skipping around with the low update frequency, they are replaced with *doppelgängers*. A doppelgänger is a bot trying to simulate the behavior of the real player in the period between receiving updates. This *guided AI* of the doppelgängers is supported by guidance information sent with the updates. This information, containing e.g., the direction of movement and targeting, helps predicting the player's behavior for the next second.

Multicast

Donnybrook has two different message dissemination schemes for update messages and for guidance messages (update messages are addressed to subscribed nodes, i.e., nodes whose interest set contains

the sender). Interest sets limit the traffic caused by incoming updates, but there may be nodes which are of interest for many other nodes and whose outgoing bandwith does not have enough capacity to send update messages to all subscribed nodes. Therefore, nodes with spare capacity can help disseminating these updates.

If a node's available bandwith is greater than a certain reserved amount, it joins a *forwarding pool* serving those with scarce bandwith. Nodes whose outgoing update messages exceed their available bandwith build a probabilistic forwarding tree by choosing a random subset from the forwarding pool, whose total capacity suffices the requirement. This is done separately in each frame where necessary. As the selection from the forwarding pool is made randomly, the nodes in the pool advertise only half of their capacity to be able to cope with being selected twice in one frame.

Guidance messages of nodes with a low outgoing bandwidth are also forwarded, by a set of forwarder nodes. The assignment of these forwarder nodes is done in the beginning of the game. Guidance forwarders are nodes who are not in the update forwarding pool and have spare inbound capacity. Depending on that spare capcity, a node can forward guidance messages for a certain amount of other nodes. The guidance messages are always sent to all other players. Nodes forwarding guidance information for multiple nodes may reduce traffic by aggregating guidance messages. Failed forwarder nodes are detected via timeouts. Replacement forwarders are discovered using a bit in each guidance packet, indicating that the sending node still has unused forwarding capacity.

Evaluation

The paper has two evaluation sections. The first one is about the evaluation of user experience, the second one deals with performance and scalability.

The user experience part is accomplished with a user study using a modified Quake III game. Pairs of human players play a game together with 30 bots. There are three three different setups: Low bandwith (rate limit of 108kbps) with interest sets (`LoBW-IS`), low bandwidth without interest sets (`LoBW`), and high bandwidth (no rate limit) whitout interest sets (`HiBW`). Each pair of players is told that there are two different servers which they have to compare. One group compares `LoBW-IS` with `LoBW`, the other has `LoBW-IS` and `HiBW`, both unaware of what setups they got. Players may chose how much time they spend playing and afterwards have to rate the fun-ness of each game on a scale of 1 to 10. As expected, `LoBW-IS` got statistically significant higher game times and rates compared to `LoBW`. Comparing `LoBW-IS` with `HiBW`, the times and rates of the two are almost equal, with `HiBW` being only insignificantly better.

For evaluating the effectiveness of the algorithms with larger numbers of players which the current FPS implementations are not made for, deathmatch and capture-the-flag (CTF) games are simulated with varying numbers of players using the workload generator delveloped for Colyseus [4], modified using Quake III's bot code. CTF games are intended to represent games with skewed player attention. Three different bandwidth models are applied. The first is derived from the US broadband host distribution (`BB`), the other two are based on BitTorrent host measurements, with 4 and 6 MB/s cap respectively (`P2P-4` and `P2P-6`). The primary performance metric is the percentage of update messages that are received on time (within 150ms); the threshold for acceptable performance is set to 96%.

Three different systems were set up for the competition: A standard multicast system like in Colyseus (`Standard`), a Donnybrook implementation with and one without guidance forwarding (`Donnybrook` and `NoGF` respectively). The results show that `NoGF` supports about 13-16 times more players than the `Standard` system. And `Donnybrook` is still 2-3 times better than `NoGF`. `Donnybrook` works for up to 400 players in the `BB` network model and up to 800/1000 in the `P2P-4/P2P-6` model, indicating that a few nodes with a high bandwidth can help to significantly increase the game size. The authors conclude that in general, greater variation enables greater scale. Large subscriber set sizes (many interested in one player) have only a slight impact on the update delivery quality. Even with half the number of total players in a subscriber set, the percentage of packages delivered in time is still over 90%. Churn

was simulated with probabilities between 10 and 50% per 30 minutes, showing almost no performance degradation.

Discussion

Being one of the most recent academic peer-to-peer gaming systems, Donnybrook introduces some rather advanced topics like doppelgängers update forwarding. Evaluation has shown that strong nodes may effectively help weaker nodes to participate in the network, which is an important feature in typically heterogeneous peer-to-peer networks. The principle of Donnybrook's interest sets includes game semantics to estimate the importance of certain objects to a particular player. This is a very powerful feature effectively reducing network traffic without compromising the user's view of the game. On the other hand, this feature requires specific adjustments for each game, maybe even for each gameplay mode.

Replacing less important players with doppelgängers is an advancement of the dead reckoning technique. The paper does not go into detail about the guided AI but refers to [26]. Certain situations may still be problematic using doppelgängers. Players not in someone's interest set could suddenly perform activities that affect the other player. This information may not be delivered in time, especially when affecting several other players at once, which could be the case for sound effects or explosions and range-affecting weapons in general. Doppelgängers cannot reliably perform exactly the same activities, for instance to realistically produce the correct footstep noises.

2.3.5 VON

VON [16, 17] is a peer-to-peer system for networked virtual environments based on the concept of the Voronoi diagram. VAST (`http://vast.sourceforge.net/`) is a SourceForge project that provides a library for building peer-to-peer NVEs based on VON. Until today, there has been a lot of research around VON; this section will only give a brief overview.

Concept

The main idea of VON is to build a Voronoi diagram [15] from the players which are scattered on a 2D plane, the NVE/game world. Using the Voronoi diagram, each node can find all *enclosing* and *boundary neighbors*. Enclosing neighbors are nodes having a common edge with the node in the Voronoi diagram. Boundary neighbors are those whose areas are cut by the node's area of interest border. The Voronoi diagram is not built globally but every node builds its own for its proximity. Position updates are sent to all recorded neighbors in the Voronoi diagram. Boundary neighbors inform the node when there is a new node overlapping the area of interest boundary.

For joining the network, the new node contacts a gateway server (basically some node in the network) who forwards the join request to the node whose region contains the desired position of the new node. This *acceptor node* sends the list of neighbor nodes to the new node which then builds up its Voronoi diagram. Before leaving the network, a node informs all of its enclosing neighbors which may then decide to notify neighbors about changes where necessary.

To avoid bandwidth bottlenecks in crowded areas where nodes have many neighbors, the authors propose a dynamic area of interest adjustment. Nodes shrink their area of interest radius when a certain connection limit is exceeded and restore it when the number of connections falls below that limit. Nodes who are still in the area of interest of a neighbor node are not disconnected.

Evaluation

VON is evaluated in a simulator with up to 2000 nodes in a 1200x1200 area. Nodes make 10 movements per second, representing rather fast games.

The transmission size (bytes/s) grows linearly with the node density for a fixed and sublinearly for a dynamic area of interest. The topology consistency (which is not defined in detail) remains above 99.9% between 200 and 2000 nodes. Also the average drift distance (wrong position information) is below 0.1 is most cases. Message loss up to 50% does not cause the topology consistency to fall below 90%; beyond that, it decreases rapidly.

Discussion

Voronoi diagrams allow for efficient and completely decentralized overlay topology management. There is no global knowledge necessary and nodes only communicate with nodes in their vision range. Direct communication causes low delays and thus a good awareness of the surrounding players.

But the Voronoi diagram requires maintenance. Especially in very crowded places this could cause a high transmission and computation overhead and even topology inconsistencies because the Voronoi diagram has to be changed frequently. Direct communication may overburden the connection capacities of some nodes. A dynamic area of interest limits the required bandwidth, but this will not help for heterogeneous networks as a node cannot disconnect from nodes whose area of interest cover its position. Message forwarding approaches (see Donnybrook and pSense) could help overcoming this problem.

2.3.6 pSense

pSense [30] is an unstructured and dynamic peer-to-peer network specialized for MMOGs. Based on locality within the game world, it provides interest management and efficient multicast in an highly dynamic game world. Unlike many other peer-to-peer gaming approaches, pSense does not split the world into (static) regions. Instead, every node (i.e., player) keeps track of its neighbors in the game world and only exchanges information with those. Thus, each node is immediately informed about activities nearby and (almost) completely unaware of everything that happens outside its vision range.

Algorithm

Besides the usual task of disseminating position updates, the system has a second important task, which is to keep the network connected, as there is no underlaying system like a DHT or super peers with global knowledge that do this. Each node maintains two lists, a *near node list* and a *sensor node list*. The near node list contains all nodes in vision range. The system assumes that vision ranges are symmetric, thus all nodes in the near node list have the local node in their list. The sensor node list contains nodes that are just outside the vision range, but possibly not too far away. They are like antennas pointing to different directions. Their function is to keep in touch with nodes outside the vision range, especially those that are approaching the vision range.

Messages (position updates) are sent to all nodes in near node and sensor node list. If the outbound bandwith capacity is exceeded, a random subset of messages is dropped. To make sure that all messages in the vision range receive the message, receivers forward messages to all known nodes which are within the vision range of the originator but not in the message's receiver list, as long as a certain hop count has not been exceeded.

The area around the vision range is partitioned into eight sections for the different directions. There is one sensor node for each of those sections. The sensor node list is maintained by periodically sending

sensor node request messages to all nodes in the sensor node list, returning new sensor node suggestions. For each section, the nearest node outside the vision range is chosen as the new sensor node.

When joining the network, the new node connects to some arbitrary node that is already in the network. That node provides information about nodes closer to the new node via a sensor node request. The new node then repeatedly sends sensor node requests, and its position updates are forwarded by the sensor nodes, so that the node finally finds all of the neighbors in its vision range.

Evaluation

System quality is measured using the following protocol quality model: $PositionAge(p,q)$ is defined as the time (in rounds) between player q sending his position update to p and p receiving the update. Within the interaction range (defined by the game application), the protocol quality function $PQ(p,q)$ between the players p and q equals the position age function. Outside the interaction range but within vision range, it is defined as $PositionAge(p,q)^{(1-\frac{dist(p,q)-IR}{VR-IR})}$, where $dist(p,q)$ is the distance between the nodes. VR and IR are vision and interaction range. Outside the vision range it is set to 0. The protocol quality $PQ(p)$ for node p is the sum of $PQ(p,q_i)$, and finally the total protocol PQ quality is defined as the average over all $PQ(p_i)$.

For evaluation, the system is run in a simulated network with up to 600 players. The default game world size is set to 1000x1000 with a relatively large vision and interaction range with radii of 200 and 50 respectively. The node's outbound bandwith is limited to 5KB per round. The experiments show that the protocol quality does not depend on the total number of players but just on the density (to be more precise: the average numbers of players in the vision range). With 100 players, PQ is almost 1, which is the optimal value. Client/server solutions have a PQ around 1.4 as all messages have to pass the server and thus require at least two hops. Increasing numbers of players in the vision range require (linearly) increasing bandwith; a limitation of the latter reduces protocol quality. But even with 50 players in the vision range, pSense is still slightly better than client/server.

Discussion

pSense is a system based on locality (in the game world) that works completely decentralized. There is no global authority required and all peers have the same responsibilities. The evaluation results show an excellent performace as long as the game world does not get too crowded. As many games have a small number of important places of interest, this could though be a problem. One could think about techniques for dynamically decreasing the vision range (like VON) and/or trying to focus on important neighbors (like Donnybrook).

pSense only considers players and does not provide a solution for handling other game objects. Although in principle objects can be handled in a similar fashion as players, there are still synchronization issues. The authors argue that synchronization and conflict resolution can be handled by a central server as this is necessary rather infrequently. However, this is may not be a satisfying solution.

2.4 Design Space for Peer-to-Peer Games

This section concludes the related work section for peer-to-peer gaming by assigning the presented solutions to the different problem categories *low-latency information dissemination*, *network maintenance*, *object management*, and *security*. Clearly, not all presented approaches provide solutions for every topic. Thus, a complete system needs to adopt parts of various of these approaches.

2.4.1 Low-Latency Information Dissemination

One of the most specific requirements for fast-paced peer-to-peer multiplayer games is a low-latency information propagation to interested receivers (interest management [24]). The most important interest factor is locality (closeness in the game world), as the games discussed here are games in which players have a very limited vision range (and thus area of interest) compared to the size of the game world. Examples of those games are first person shooters or role-playing games. This does not apply to e.g., real-time strategy games where the player is able to see large parts of the game world. The typical information that is to be disseminated quickly are position updates. While most other information exchange is only necessary within small groups (mostly 2 players) interacting with each other, there may be other information that everyone around is interested in. But assuming a simplistic game, position updates are sufficient, and other types of information can be handled the same way.

The probably most obvious (and in fact most common) approach is spatial multicast, i.e., information is multicast to all participants within a certain range in the game world. There have been approaches using IP multicast for NVEs [22, 11], but they have shown to be practically infeasible because of basically two problems. First, the spatial structure needs to be mapped to multicast groups which are inflexible and only available in very limited numbers. Second, and more importantly, IP multicast is not widely supported in the Internet, limiting the area of application to small networks. Instead, all recent approaches do application layer multicast which is flexible and less network-dependent. But they have the downside of being less (upstream) bandwidth efficient.

The easiest way of doing spatial multicast is through a server. The server collects all (position update) messages from the players, assigns each update to those players that are interested according to the interest management rules, and sends them to the the corresponding players in an aggregated form. This has the advantage that the players have minimal bandwidth requirements as they only have to send their updates to one server and receive perfectly customized and aggregated data. The downside, of course, are the server requirements. The server does not necessarily need to be one central dedicated server; instead in a peer-to-peer game, for each region a server may be elected among the peers. But in any case, the server has to cope with a high load, needing to communicate with all players (in its region) and performing interest management.

To avoid bottlenecks at machines managing the whole region, several approaches completely decentralize the position update dissemination. Position updates usually account for a large part (if not largest) of the traffic in multiplayer games. Thus, other types of information may still be managed by central instances (see below). Decentralized approaches usually do not rely on fixed regions. Although this option is also available using generic application layer multicast techniques [2, 10], e.g., with one multicast group per region, dynamic approaches have several advantages. First, using regions with a continuous game world, there is always the necessity of (smooth) region crossing and visibility across regions. One solution for these problems would be having overlapping regions or multiple interleaved region grids, so that the player is always member of (at least) two multicast groups. Second, statically sized regions may become overloaded in crowded places. Smaller regions allow for a higher player density but cause higher overhead due to more frequent region switches.

In the unstructured approaches (i.e., those without fixed regions), each node keeps a neighbor list of nearby nodes. Typically, every node is in direct contact with all nodes in its vision range (and/or nodes in whose vision range it is; this is the same in simple cases with equally sized and undirected vision ranges). The challenge for these approaches is to quickly and efficiently detect new neighbors approaching the vision range. Late detections will cause other players (or generally objects) to suddenly appear in the vision of a player who will certainly be annoyed about that happening.

The techniques for neighbor detection vary between the different approaches. VON builds voronoi diagrams to find the boundary neighbors who are responsible for introducing approaching nodes. pSense uses sensing nodes which are neighbors outside the vision range pointing like antennas in different directions. In Donnybrook instead, low fidelity updates are cast to all nodes in the game (for large game

worlds though, this needs to be split into regions), and each node subscribes to those neighbors it is interested in and then receives high-fidelity updates.

Although decentralized position update dissemination distributes the load amongst all nodes, there may still be cases where some nodes do not have enough (upstream) bandwidth to perform their task. This could be just due to network heterogeneity, i.e., there are some weak nodes. Another possible reason is asymmetric interest in systems like Donnybrook where each node can choose who it is interested in. Therefore, some systems have forwarding mechanisms helping weak/overloaded nodes disseminating their updates. This can be done by randomly excluding receivers from the update messages and making the remaining receivers responsible for forwarding to missing receivers like pSense does. Donnybrook has a more sophisticated mechanism which is necessary because nodes do not know the interest sets of other nodes and thus cannot simply forward to other nodes who are interested in a certain update. Instead, there are pools of nodes with spare capacity that can be utilized by the senders as explicit forwarders.

2.4.2 Global Network Maintenance

Besides the need for efficient update dissemination, it is also of major importance for a peer-to-peer overlay system to prevent network partitioning. For games, this means in particular that whenever two players are nearby (i.e., in each other's vision range), they must be aware of each other. Network partitions in peer-to-peer games result in unintended 'parallel universes' where players stand beneath each other but cannot see each other.

A related issue comes into play when nodes want to join the network. A new node always needs an entry point to the network which is ususally simply one peer that is already part of the network. The task of finding such a peer is assumed by almost all peer-to-peer systems to be accomplished off-the-band. But still, the initially contacted peer (gateway node) needs to introduce the new node to several others in the network to make the new node become a peer. In peer-to-peer gaming systems, it is particularly necessary to make the new player known to the other players nearby in the game world.

Systems using a DHT have almost all of the above tasks performed by the DHT. This approach has the great advantage of allowing to make use of mature (and thus potentially more stable) existing DHT implementations and simplifying the remaining implementation. But, as mentioned before, general DHTs lack some properties that are important for games. One of these is the lack of realtime capabilities. A typical DHT key lookup takes several hops resulting in delays in the order of magnitude of seconds.

Many newer systems abstain from using a DHT and have their own optimized network maintenance techniques. These are usually based on game world proximity and closely related to the update dissemination mechanism. As needed for sending direct updates, a node already knows all of its neighbors in the vision range. Assuming a high density and uniform distribution of players in the game world, this is already sufficiently preventing network partitions and allowing new players to be routed to their neighborhood. The former is accomplished implicitly by consistently informing neighbor nodes about approaching new neighbors. Thus, moving nodes always keep connected to their current neighbors. The latter can be similarly done via repeated neighbor suggestions which allow the new node to iteratively approach its destination neighborhood.

As the previous assumption about player distribution is never true in real games, it is necessary to have additional mechanisms especially for bridging empty areas in the game world. pSense uses 'antennas' in the form of sensing nodes. Each node keeps connection to a set of sensing nodes outside the vision range pointing in different directions. Empty regions then cause the sensing nodes to be far away but the network remains connected. In VON, the properties of Voronoi diagrams ensure that there is always contact to the boundary neighbors, even if they are far outside the vision radius, to prevent partitions.

2.4.3 Object Synchronization and Game Logic Execution

As any real (and not completely simplistic) computer game has objects with associated state that can be manipulated, there is obviously the need for supporting them in massively multiplayer online games. The challenge in distributed multiplayer games is to synchronize the objects so that each player has a (sufficiently) consistent view and all his actions on these objects have the expected consequences. This is particularly difficult to achieve when two players manipulate an object concurrently. Game objects are basically everything in the game world that is not directly related to a single player and that is not part of the immutable terrain.

Immutable parts of the game are usually installed (and if necessary updated) together with the game and do not require any synchronization; all participants agree based on identical data. The player's avatar (typically including its properties and direct belongings) is treated separately and not the same way as other game objects because it constantly belongs to one player (and thus one node in the network) and is always controlled from there. Each avatar has a dedicated node (i.e., the corresponding player's node) that is naturally responsible for serializing all operations.

Some types of objects, most commonly any type of NPC (non-player character), have think functions associated that (regularly) execute some piece of code which automatically makes these take some action, for instance moving around in the game world. But the think functions may also be much simpler, possibly just causing some slow resource regeneration. These functions must be executed somewhere; the decision on which node(s) a particular think function is to be executed is closely related to the decision whose nodes are responsible for managing object modifications.

Basically, it has to be decided which consistency mechanisms are to be applied to the objects. Pessimistic models (e.g., locking objects for modification) is not feasible for distributed realtime games mostly of two reasons. First, in realtime games, it is not possible to wait possibly several seconds whenever modifying an object. And second, as the system consists of unreliable nodes connected over the Internet, locks may not be correctly released in some cases. Necessary timeouts would cause even more waiting for object manipulation or just reading the current state.

Thus, there is the need for optimistic consistency. In most cases it is not necessary to have strict consistency, but the game logic must then be prepared for handling some inconsistency properly. Colyseus has an optimistic consistency model in which all updates are tentative until approved by the node responsible for the primary copy. The responsible node serializes all modifications and rejects conflicting changes. Of course, with Internet-typical latencies, the player will eventually notice when an object that he manipulated is reverted after a few seconds. The game logic will not always be able to compensate and hide these effects from the player.

The common approach for object consistency is to serialize changes on one node that is chosen to be the owner of the particular object. In SimMud there is a coordinator node for each object, and Colyseus has nodes holding the primary copy which is the reference object, i.e., everyone eventually has to synchronize to that object. This approach makes the serialization relatively easy as it is done locally on one node. But it also entails a reliability issue, since a failure of the one node responsible for a particular object will result in a loss of its state.

Data loss can be prevented by having certain nodes keeping backup copies that are always synchronized with the primary copy. Ideally, one of the nodes holding a backup copy can immediately take over when the failure of the node holding the primary copy is detected.

One approach of selecting the nodes responsible for a particular object is based on regions. Each region has one coordinator node who is responsible for all objects in that region. Another possibility is to have one coordinator for all objects of a particular type. Better load balancing is achieved by assigning an ID to each object and giving each node the responsibility for a certain range.

All approaches that cover game object management use a DHT for manging responsibilities. Although DHTs have the previously described weaknesses, especially concerning latencies, they appear to be the most feasible solution so far. Timing concerns can be answered by pre-fetching object copies before they

are needed (see Colyseus). DHTs easily allow assigning coordinator nodes to the objects in any of the above mentioned ways. It is just necessary to assign IDs to the regions, types or objects respectively, and it is the DHT's task to assign those to nodes.

One important feature that is missing in most DHTs is range-queriability which is required in the typical case where a node is interested in everything (i.e., all objects) within a certain area. There are approaches (like Colyseus) extending a DHT with range-query features, consequently avoiding the need for starting several lookups at once for querying a whole region. But this kind of range lookup is still much more complex compared to a simple DHT lookup. BubbleStorm may be a solution to that problem as it allows to efficiently execute any kind of (exhaustive) query and thus makes it easy to implement range queries.

2.4.4 Security

An obviously important issue in commercial quality games, especially in MMOGs, is cheat-resistency. The operators of popular MMOGs put a lot of effort in cheat prevention as users can even make real money by selling virtual goods on trade platforms like eBay. Although this may not be the case in other games, cheating players make the game unattractive for others. A related topic is authentication which is particularly important for games that players have to pay for; but that feature is desirable even in free games for preventing undesired behavior like identity theft.

In current publications, this topic is mostly seen orthogonal to the previous topics and thus covered separately. There are some papers about cheat prevention particularly in peer-to-peer games, e.g., [18, 19]. Since it is a separate topic, it will not be further covered in this work.

2.5 Implementation-relevant Technologies

This section introduces the Irrlicht engine and the protocol CUSP, both being important external components for this work's implementation.

2.5.1 Irrlicht Engine

Irrlicht[3] is a free and open source realtime 3D engine. It can be used with C++ and .NET on various platforms including Windows, Linux and Mac OS X. In the current version 1.5, 3D rendering is accomplished either via Direct3D (8.1/9.0), OpenGL (1.2-3.0) or one of two software renderers. The 3D backend is completely abstracted by Irrlicht, so that besides the initialization the application programmer does not need to care about the particular backend.

Apart from its cross-platform capabilities, Irrlicht's main goal is to make the development of 3D applications (of which most are games) as easy and convenient as possible. It supports reading all common file formats for textures (i.e., bitmaps) and 3D meshes and allows for direct reading from archives. There is a customizable scene management for indoor and outdoor and support for character animation. Several effects including particles, light maps, environment mapping and stencil buffer shadows allow for appealing 3D renderings. The built-in collision detection makes it easy to efficiently respond to object collisions in the game world.

The above features make Irrlicht very attractive for creating a 3D game like Planet π4. This is especially the case because Planet π4 is supposed to run on different platforms and the focus is not on the technical details of the 3D graphics. Using OpenGL and/or Direct3D directly would require a much higher learning and programming effort.

[3] `http://irrlicht.sourceforge.net/`

API Introduction

This section briefly introduces the Irrlicht C++ API. As this is only an overview, it is recommended to refer to the complete API[4] for detailed descriptions. The Irrlicht API classes are assigned to several namespaces. All parts of Irrlicht are in the top level namespace `irr` which has the sub-namespaces `irr::core`, `irr::gui`, `irr::io`, `irr::scene` and `irr::video`. In the following, there will be no references to the namespaces of the particular classes; again, refer to the complete documentation for any details.

The core class of the Irrlicht API is `IrrlichtDevice`. Each application using Irrlicht usually creates one instance of that class. Instances are created using the function `createDevice`. `createDevice` has several optional parameters of which the most important allow specifying the 3D device type (i.e., whether to use software rendering, Direct3D, OpenGL or none), window size and color depth. An `IrrlichtDevice` object holds a video driver (`IVideoDriver`) and a scene manager (`ISceneManager`) object.

The `ISceneManager` object represents the virtual scene that is to be rendered. A scene contains any number of *scene nodes* (`ISceneNode`). Subclasses of `ISceneNode` implement the various scene node specializations, e.g., meshes, animated meshes, terrain, text, light sources and cameras. A new scene node is added using one of many corresponding `ISceneManager` methods of which most are named `add...SceneNode`. For instance, `ISceneManager::addMeshSceneNode` adds a mesh scene node from a mesh (`IMesh`) that can easily be loaded from a file using `ISceneManager::getMesh`. Different kinds of animations (e.g., movements, rotations or delayed deletes) can be realized by assigning `ISceneNodeAnimators` to scene nodes.

To be able to render the scene, it is necessary to have a *camera* scene node. Such is added using `addCameraSceneNode` or one of the two specialized versions `addCameraSceneNodeFPS` and `addCameraSceneNodeMaya` which already support basic user control.

`IVideoDriver` is used to control anything related to the screen. For rendering the scene, you first have to call `IVideoDriver::beginScene`, then `ISceneManager::drawAll` and finally `IVideoDriver::endScene` which results in the presentation of the rendered image on the screen. Besides drawing the whole scene, it is also possible to perform direct 2D or 3D drawing to the video driver object between the `beginScene` and `endScene` calls. Loading textures is also done via the video driver.

For processing 3D geometry, Irrlicht provides an extensive library of optimized geometric functions. One important exemplary structure is the (3D) vector. The `vector3d<T>` class (`T` is the element data type, ususally `float` or `double`) provides various methods including simple addition, multiplication, length and distance, but also some angle computation.

User input is passed to the application through the `IEventReceiver` interface. An application implements that interface and registers itself using `IrrlichtDevice::setEventReceiver`. There are different event categories: GUI events (e.g., a button is clicked), joystick events, key input events, log events, mouse input events, and user events. So for example processing mouse events allows the application to react on mouse input.

Finally, Irrlicht provides a simple GUI environment for creating dialogs etc. The predefined controls include buttons, check boxes, lists, combo boxes and scroll bars. The GUI API also includes 2D image and text rendering functionality.

2.5.2 CUSP

CUSP (**C**hannel-based **U**nidirectional **S**tream **P**rotocol) [33] is a transport protocol initially developed for BubbleStorm but available for any application. BubbleStorm originally used TCP as its transport protocol but TCP has soon shown to be too far from optimal for BubbleStorm. Firstly, BubbleStorm requires a message passing protocol. But unlike plain UDP, messages have to be delivered reliably, and UDP does

[4] `http://irrlicht.sourceforge.net/docu/`

not support in-order delivery. TCP does not provide prioritization, and the head of line blocking problem [29] makes it impossible to acceptably implement priorized messages. Once data is queued for transmit in TCP, data with higher priority cannot pass the older data in the buffer. Thus, big messages with a low priority may block high priority data.

There are two other transport protocols that tend to be more suitable for BubbleStorm: SCTP [31] and SST [13]. Both support multiple streams, allowing for parallel transmission of data. But SCTP only allows for a fixed number of streams that has to be selected when creating the connection. SST has several shortcomings (e.g., a broken windowing) that make it unusable for BubbleStorm.

CUSP has unidirectional streams which is efficient for realizing the packet-oriented behavior. A stream can be created, a message sent and shut down, resulting in only one packet. Thus, streams do not produce any packet overhead. Connections are encrypted by default. Applications do not need to take care of encryption; optional user authentication is a simple task. Finally, CUSP supports mobile IP and NAT traversal (with help of the application) of which the latter has become a very important feature of peer-to-peer applications in general.

One of the reasons for using CUSP in this work even without BubbleStorm is simply the need for testing the protocol. At the time of writing, the CUSP implementation is still very new and needs for evaluation by applications using it. But that is not the most important reason. CUSP is very suitable for implementing peer-to-peer games, mostly because its prioritization and NAT traversal capabilities. The specific benefit for a peer-to-peer game is the stream concept which allows for both a reliable transport like TCP and a low latency packet-oriented behavior like UDP. By having feedback about buffer states, the application gains the potential of precisely controlling its traffic, and it may remove selected data (e.g., packets with outdated information) from the buffers by resetting streams.

Functional Overview

Between each pair of hosts that communicate using CUSP, there is one *channel*. A channel is the 'physical' link between the hosts. Encryption and congestion control is located at the channel layer. Channel management is not the application programmer's duty. Once a handle to a remote host has been obtained, the application may create any number of *streams*. Similar to TCP, an application *listens* for incoming streams at certain *service IDs*. Service IDs correspond to port numbers in TCP and UDP. But as CUSP is based on UDP, each CUSP instance binds to exactly one UDP port through which all services are realized. As mentioned before, streams are unidirectional. Applications can only create outgoing streams (to a remote service) and receive incoming streams (after listening for a certain service). Each outgoing stream has a priority based on which data is sent; the queued data of the stream with the highest priority is sent until the stream's queue is empty or data has been queued for a stream with a higher priority. Each stream has its own flow control.

API Introduction

The following provides an overview of the CUSP API. Part of this work is the implementation of C++ bindings for the transport protocol which itself is written in Standard ML. As this work only uses the C++ API, the section does not cover the original Standard ML API. But besides language-related conceptual differences, both APIs are very similar. Also, this section is simplified and reduced to the core functionality and thus does not cover the full API including all (helper/handler) functions necessary for an implementation. Refer to the CUSP documentation[5] for more details

EndPoint An `EndPoint` is the local connection point for the transport protocol. Any application using CUSP usually creates one instance of `EndPoint` which is provided with a UDP port to bind to.

[5] `http://www.dvs.tu-darmstadt.de/research/cusp/`

The `EndPoint` class provides two main functions: contacting a host given by its IP address and advertising (i.e., listening for) a service.

Methods:

create(port, key, encrypt, publicKeySuites, symmetricSuites) Creates an `EndPoint` bound to the given UDP port. All parameters besides `port` are optional and specify encryption settings. The private key set is passed as `key`, encrypt decides wether encryption is required. Encryption is always turned on for a cconnection if at least one of the two endpoints requires encryption, thus having encryption keys is mandatory. If no key is specified, a new random key is created. `publicKeySuites` and `symmetricSuites` allow for selecting or deselecting particular encrption suites.

contact(address, service, handler) Asynchronously contacts a remote host (peer) given by its address and creates an outgoing stream to the given service ID. When successfully contacted, a host handle (see `Host`) and an `OutStream` are passed to the corresponding handler function. The `Host` can then be used to create further streams.

advertise(service, handler) Advertises a certain service given by its ID (*service*), i.e., listens for incoming streams for the service. The handler method is invoked for each incoming stream.

unadvertise(service) Closes a previously advertised service. Subsequently, no more incoming streams will be accepted for that service.

setRate() Sets the maximum transmission rate (bytes/s) for the endpoint.

key() Returns the local key set used for encryption, e.g., for persistently saving it to disk.

hosts() Returns the set of hosts known to the endpoint.

channels() Returns the set currently active channels.

Host Represents a reference to a remote host. Each host is identified by its public key. Once a host handle has been obtained, its IP address may change without the need for a re-contact on the application side.

Methods:

connect(service) Creates an outgoing stream to the given service. The `OutStream` object is immediately returned and ready for write. If the stream is not accepted (or reset) at the remote side, the write will be responded with a reset (see `OutStream`).

listen(handler) Listens for incoming streams exclusively for the specific remote host. Also, unlike `EndPoint::advertise`, the service ID is not chosen by the application but returned by the function.

unlisten(service) Stops listening for the given service. Subsequently no more incoming streams will be accepted for that service.

key() Returns the remote host's public key.

address() Returns the current IP address, if there is one. The address is only available when the there is an active channel to the host.

InStream An `InStream` is a stream for reading data. Objects of this class are obtained via the `EndPoint::advertise()` and `Host::listen()` handlers.

Methods:

read(handler, maxCount) Asynchronously reads data from the stream. The handler has three methods: `onReceive` is invoked when data is read, `onShutdown` indicates a shutdown from the remote end, and `onReset` indicates a stream reset. The optional parameter *maxCount* specifies the maximum number of bytes to be read.

reset() Resets the stream, ensuring that no more data will be read, i.e., no more handlers will be invoked. Subsequent writes on the remote side will result in a reset notification.

OutStream Allows writing data to a remote host. Instances of this class are created using `Host::connect`. Each `OutStream` has a priority represented as a `float` value defaulting to 0.

Methods:

getPriority() Returns the stream's current priority.

setPriority(priority) Sets the stream's priority.

write(data, size, handler) Writes data to the stream. When the stream becomes ready for writing further data, the handler's `onReady` is invoked. It is illegal to call `write` a second time before `onReady` is invoked. If the remote side has reset the stream, the handler's `onReset` is invoked.

shutdown(handler) Shuts down the stream, i.e., indicates to the remote end that no more data will be written. The handler method is invoked with a `bool` value informing about success or failure of the operation.

reset() Resets the stream, ensuring that no more handlers will be invoked. Also, no further operations on the stream are allowed after a reset.

Address `Address` objects represent IP addresses.

Methods:

fromString(string) Creates an `Address` object from a string representing an IP address or host name with an optional port specifier.

toString() Converts the address to a string.

3 Concept

This work's implementation is supposed to be a basis for the comparison of different overlay network systems concerning their capabilities for MMOGs. The following sections describe the setup and the main ideas behind the implemented networking approaches.

3.1 Basic Setup

As introduced earlier, the prototype game Planet π4 is used as the reference game. An alternate approach would be to use a complete existing game, ideally a popular one. The authors of Colyseus and Donnybrook, for instance, used modified versions of Quake II/III. That has the advantage that there is already a reference networking implementation, namely the original client-server version. Also, it is easier to find (experienced) players for a popular game than for a completely new one. Finally, the existing game is surely enjoyable to test players, while the research prototype game is not primarily intended to be enjoyable, thus not so much optimized in this respect.

But there are good reasons for using a custom game. First, having the whole source code of the game under control allows for any kind of necessary modification. For most games, there is no source code available; and even if, there are mostly restrictive licenses at least on the media data (3D models etc.) that limit scientific reuse. Second, today's games have rather complex game mechanics with potentially a lot of state that needs to be synchronized. A custom game however can be kept as simple as necessary to allow for easy implementation and analysis for the most important aspects. The game architecture can be designed for an easy adaption to different overlay network models, which may not be the case in existing games that are designed just for client-server operation. With these arguments in mind, the decision was made to build on Planet π4.

The Planetπ4 implementation, especially the networking part, is keenly restructured to make it adaptable to various network models. The original implementation snapshot's network implementation is based on multicast groups, possibly serving as a region-based update distribution system. But support for multiple regions was not finished at the time of starting of this work. The messaging model is originally based on plaintext messages that are simple to transfer over various channels, e.g., a Skype chat.

To briefly recapitulate the Planet π4 game mechanics: Each player controls his spaceship (i.e., avatar) in a free 3D space, trying to destroy other players' ships by shooting them with 'energy missiles' (the technical term in the implementation is *bullets*). The following are the important mesages that are exchanged within groups in the game:

`SHIP` is regularly multicast by each player to all group members updating his ship position and rotation.

`FIRE` indicates the firing of a missile (bullet).

`IHITYOU` informs another player that he is hit by a missile (bullet). The destination player has to decrease his hit point value by one whenever he receives this message.

`IDIED` is multicast by a player whose hit points reach zero, indicating that his ship is destroyed and that he is restarting.

It is obvious that this simple messaging scheme is completely insecure against even simple cheating attempts on the network level, e.g., by just sending arbitrary `IHITYOU` messages to another player. But as cheat prevention is not a focus of this work, that issue will be ignored, assuming all participants are cooperative. For more details on message types and implementation, refer to chapter 4.

3.2 Client-Server over TCP

For getting started and getting familiar with the game's implementation details, a simple client-server solution using TCP is implemented. Each client connects via TCP to a dedicated server that manages the set of clients and group membership. Also, message multicast within groups is completely performed by the server. A client, once connected to the server, may join any number of groups, each by sending a `GRP_JOIN` request. After a group has been joined, the client receives all messages that are sent to that group using `GRP_SEND`. Clients leave groups via `GRP_LEAVE` requests.

Each client is assigned an ID by the server. The IDs allow the clients to identify different players, e.g., for `GRP_LIST` showing all members of a group; the server identifies its clients based on their remote address/port pair. A new group can be created via `GRP_CREATE`. Clients can also unicast messages to a single other client identified by the ID (`SEND_USER`).

Discussion

The TCP-client-server implementation is primarily intended as a reference for comparing to the other implementations. It is kept as simple as possible and clearly does not have any advanced features. The server does just as much as necessary to distribute the messages amongst clients.

The whole traffic between the server and a client is routed through a single TCP connection. This particularly means that both group management and regular position updates share the same channel. Although this appears to be the simplest solution, it has in fact several unintended consequences. First, using TCP for sending messages requires a mechanism for separating the messages within the stream. For this implementation, this is done using null characters as separators between the text messages.

Additional effort is necessary to distinguish between simple one-way messages (e.g., position updates) and message *dialogs*. Examples for the latter are request-response cycles such as group operations. The problem is that the first message a client receives after sending a request may be a position update that was queued or inflight while sending the request. Thus, the client must filter incoming messages subsequent to a request for an expected answer. This approach is not very convenient for the application developer, but is the only (simple) solution if out-of-band signaling (i.e., through a separate connection) is not desired.

Another problem intrinsic to TCP is *head-of-line blocking*. Data that has been enqueued for transmission cannot be removed from the buffer and subsequent send operations can only add data the the end of the transmission queue. In case of congestion, important signalling messages cannot pass queued position updates. And old position updates may still wait for transmission while there is already a new update which makes the old one obsolete.

A common solution for overcoming these problems is using UDP instead of TCP. UDP sends and receives datagrams, fitting better to the messaging paradigm than TCP's streams. Also, UDP does in principle not have the head-of-line blocking problem as datagrams are not guaranteed to be delivered in order. UDP datagrams may be dropped in case of congestion, which is possibly more desirable than the queuing of out-of-date updates.

But as UDP does not provide any of TCP's features like reliable transport, ordered delivery and flow and congestion control, these need to be implemented by the (game) application where necessary. While special measures for achieving reliability of regular (position) update messages are usually unnecessary (a new update will be sent anyway), there are messages that have to be delivered reliably. Examples are server requests like group operations but also possibly messages for direct interactions between players which, if lost, may cause game inconsistencies. The `IHITYOU` message tells a player that he is hit; if not delivered, the player's hit points will not be affected. Even more, this type of message requires *exactly once* semantics since receiving such a message twice means that hit points are affected twice.

A solution could be using a mix of UDP and TCP for position updates and reliable messaging respectively. But it also introduces additional complexity. The described challenges lead to the decision of solely using TCP for the simplicity of the implementation.

3.3 Client-Server over CUSP

As a first test introducing the new CUSP implementation and C++ bindings, the TCP client-server implementation is ported to CUSP. Since CUSP provides an API that is notably different to the well-known Berkeley sockets API, the trial of several new design patterns was necessary to find convenient and reliable ways for using the API. Details on the implementation, again, can be found in the 4 chapter.

Each client is connected to the server via a permanent stream for each direction. The permanent streams carry their regular update messages between server and clients. The server acts like in the TCP version, just distributing all update messages from the clients within the groups. Thus, this solution is not expected to scale any better than the TCP version.

But the CUSP solution eliminates TCP's head-of-line blocking problem. Client-server request-response cycles (group operations) create separate streams with a high priority. This ensures that the client-server operations are not disturbed by the update message traffic. The update messages could also be sent in separate streams, which would allow avoiding undesired congestion. But for keeping the solution simple, that feature is not chosen to be implemented.

Discussion

The client-server version using CUSP does not come with many new concepts, it just ports the TCP functionality to CUSP. Though, TCP's head-of-line blocking problem is solved for group operation messages.

3.4 Peer-to-Peer over CUSP

The peer-to-peer version of the game implements a slightly simplified version of pSense (see section 2.3.6 and [30]). pSense is chosen because it is one of the recent systems that are optimized for low latency requirements and a highly dynamic game environment. Planet π4 does not (yet) have any object management, so the peer-to-peer system does not require that capability. Direct alternatives to pSense are VON and Donnybrook. But both VON and Donnybrook are more complex and do not appear to have any advantages over pSense in a simple scenario like the current Planet π4 implementation.

pSense does not work with regions and/or multicast groups, so Planet π4's networking layer has to be reworked, also requiring restructuring parts of the game core. The previously built-in group networking part is now replaceable. Any network layer providing group multicast can use the old group networking layer, while unstructured peer-to-peer implementations like pSense substitute the group network layer and directly interact with the game core (details can be found in chapter 4).

In pSense, each player knows all his neighbors in its vision range and directly communicates with them, i.e., each node keeps a permanent connection for (position) updates to each neighbor as long as the latter remains in the vision range. Outside of the vision range, each node keeps connections to a set of sensor nodes. Sensor nodes have the function of early notifying the node about other players approaching the vision range. More importantly, the mechanism of sensor nodes prevents network partitions particularly in low density regions were players potentially have no neighbors in the vision range.

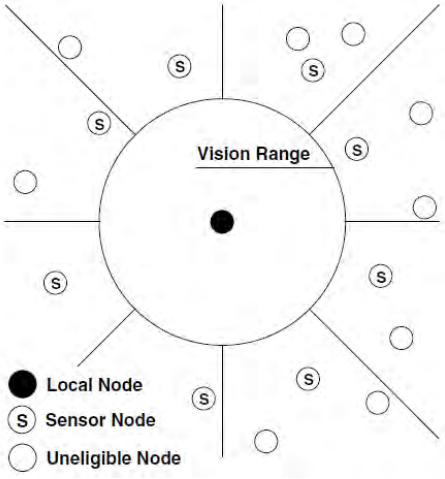

Figure 3.1: pSense sensor nodes [30]

Sensor Nodes

The concept of sensor nodes is only described for two dimensions in the pSense paper; the paper defines eight equally sized sectors surrounding the vision range (Figure 3.1). A node keeps one sensor node (if available) for each of these sectors. For a 3D game, this scheme obviously has to be adapted to the 3D space. The first idea was to split the sectors along longitudes and latitudes (like on a globe). But that would result in sectors of varying sizes. Especially those around the poles would be very thin, causing an unnecessarily high sensor node substitution frequency. The next idea was to arrange the sectors corresponding to the faces of a platonic solid; either the dodecahedron or the icosahedron. This approach ensures that all sectors are of equal size, but it requires relatively complex geometric calculations.

In order to avoid difficulties with the 3D geometry, it was chosen to start with treating the Planet π4 world as a 2D space, implementing the original pSense concept. All distances are calculated in the 2D space projected to the terrain ground plane. Although Planet π4 does in principle allow free movement in the 3D space, its terrain and ship control suggests a certain alignment to the ground. Also, the 2D approach is much more clearly arranged, making testing and debugging a lot easier. Though, the platonic solid approach appears to be a suitable optimization for real 3D worlds and should be considered to be added in the future.

Connection Management

A node keeps an outgoing stream to each known neighbor in its vision range and regularly (each 100ms) sends position updates. Furthermore, each node keeps track of inbound streams and the positions of the corresponding nodes. Nodes are identified via their public key provided by CUSP. Each position update message (from node A) contains a list of abbreviated IDs (the low 32 bit of the host key) of all receivers. This list allows the receivers to notify node A about any so far unknown neighbor (say node B) in the vision range via an *introduce* message. Once introduced, node A creates a stream and sends position update messages to B which in turn gets to know about A and starts sending its updates.

Position update messages contain the sender's vision radius. Although in the current version the vision radius is fixed and equal for all players, this always allows neighbors to identify whether they are still in another node's vision radius. When node A leaves B's vision range, it does not immediately close its connection but just stops sending further updates. For each connection, a node stores the time of the last received message. These times are periodically checked and after receiving no message for a certain time (currently 5 seconds), the connection is closed. Two things are accomplished using this approach. First,

closing connections immediately is undesired because a neighbor may leave the vision range just for a short period of time and come back in which case closing and re-establishing the connection would be wasteful. And second, crashed nodes are detected the same way, as CUSP itself does not have connection timeouts.

As specified by pSense, a node selects a sensor node for each sector to be the nearest node outside the vision range within the section. When node A selects node B as its sensor node for a particular sector, it sends a *sensor node request* to B, including the sector which B is responsible for. B stores this information and keeps sending updates to A (although it is outside A's vision range). A also has to keep sending updates, or otherwise B's connection timeout will take effect. As both A and B are updated about each other's position, B may suggest other nodes as sensor nodes that are more appropriate (e.g., nearer) to A. When A decides to replace B with a new sensor node, it informs both B and the new node via sensor node request mesages. B then stops sending its updates to A (unless it is in A's vision range) which will finally cause a connection timeout. Even if for some reason B is not properly informed about its replacement, A will stop sending its updates and B's connection to A will also time out. This provides a certain robustness against faulty nodes.

Messaging

Like in the client-server version, each connection between two peers consists of one permanent stream for each direction which transports the regular update messages. For technical reasons, i.e., to prevent CUSP from tearing down the channel, it is advisable to keep at least one stream permanently open.

As already mentioned in the client-server section, there are certain advantages in sending each update message in a separate stream instead of using the permanent stream. This concept is explained in a little more detail in the following. When sending data trough a CUSP stream or a TCP connection, the application does not have any control over the amount of data that is buffered for sending or in flight to the receiver. While this is dispensable for non-realtime applications where it is just of interest to transfer a block of data (as fast as possible), update messages have very diffrent requirements for realtime applications. In many cases it is not important that every update message arrives at its destination, but it is important that updates get delivered as soon as possible. Both is easily fulfilled when the channel has much more bandwidth available than required for sending the messages. But as soon as there is congestion, reliable in-order protocols like TCP and CUSP have to buffer data before sending. And as the application continues sending messages with a constant rate, buffers may fill up. Since the transport protocol buffers are limited, the application will eventually notice that it has to interrupt sending data, e.g., via blocking write calls. But then, there are already potentially a lot of update messages buffered of which many are out of date.

That is why realtime applications, especially games, tend to use UDP instead of TCP. UDP messages are simply dropped in case of congestion. Thus, not all messages arrive, but those that arrive, are (usually) in time. But UDP does not make any guarantees about reliability and in-order delivery, requiring the application to take care of all these requirements where necessary. CUSP allows for a compromise between the two opposite solutions by using separate (prioritized) streams for each message; the stream is created, the message data is sent and the stream is shut down immediately. Except for a few header bytes, this approach does not have any overhead compared to UDP, once CUSP has established a channel. The application may than assign increasing priorities to each message sent; the latest messages will consequently be sent first if the channel is congested. If a message has not been acknowledged (the application is notified via the stream shutdown callback), it is likely to be still buffered and may be removed via a stream reset call. If the message (including the stream shutdown indication) is already in flight, it will be delivered even after calling reset. The application only has to include timestamps to the update messages allowing to detect (and probably ignore) late messages.

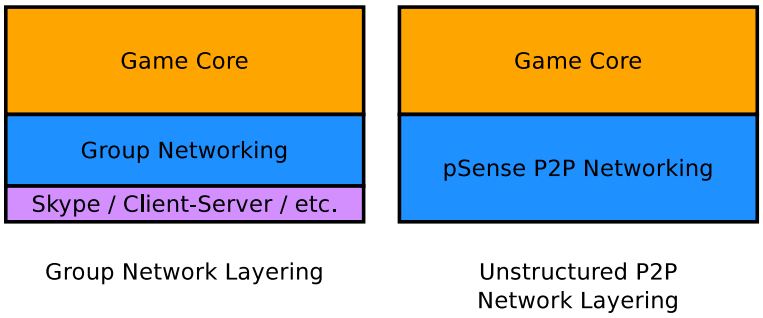

Figure 3.2: Network Layering

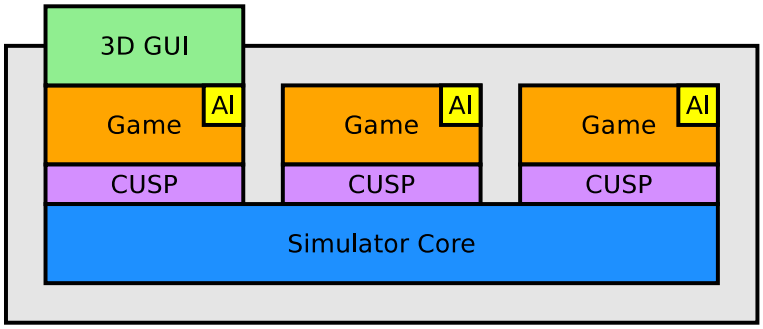

Figure 3.3: Game simulation setup

Other features

The peer-to-peer version uses a binary message format for reducing message sizes. Also, messages do not only contain current position and rotation vectors but also motion direction vectors and velocities. This allows for a simple dead reckoning; other player's ships are animated according to the motion vectors so that their movements are less jerky.

3.5 AI Player

For creating a workload from many players, especially in simulation runs, a simple artificial intelligence (AI) player (i.e., bot) is developed. Designed for deathmatch mode, the AI player aims at the nearest opponent in its vision and fires at him. Once aiming at a certain player, it tends to keep aiming at him even if another player becomes closer; this avoids frequent re-aiming and is intended to increase hit ratio. Additionally, the AI has a simple motion estimation to increase targeting precision for moving objects with the relatively slow missiles of Planet π4. Certain parameters affecting velocity, fire frequency and AI processing interval can be modified to tune the generated workload.

3.6 Simulation

Together with CUSP/BubbleStorm, a discrete-event UDP network simulator is being developed for evaluation purposes. Since CUSP works seamlessly on top of this simulator (instead of using real UDP sockets), the idea of running the game in the simulator is self-evident.

While the standalone executable of the game uses CUSP from a library, for simulation purposes the game is compiled into a library and linked to the simulator. Instead of `main()`, the game library has a `startNode()` function that similarly to the `main()` function in the standalone version sets up a single

game instance. `startNode()` takes a few parameters from the simulator, including the address of the peer to connect to. Addresses of the simulated network are not IP addresses but just plain numbers, but that difference is completely transparent to the game implementation. The simulator starts any number of game instances as defined in the particular scenario by calling `startNode` once for each node.

Adapting the Game to the Simulator

Especially for game programs, there are some challenges while adapting for simulator use, but having full control over the relatively simple code makes all necessary modifications possible. At this point, the utilization of a simple game prototype instead of a full existing game pays off. Typical games have a main loop that continuously renders frames (i.e., the user interface content) as fast as the hardware can do. This is in total contrast to most other applications which spend most of the time waiting for user input. When running an application in a simulator, the simulator has control over the main loop, and as the simulated applications do not run in real time, there is no 'as fast as the hardware can do'. The game has to become fully event-based in order to be run in the simulator. Planet π4's architecture is redesigned accordingly, allowing the replacement of the main loop and timing facilities. All game components that regularly have to execute register at that replaceable core component, which either runs the main loop by itself (standalone game) or just passes the registrations to the simulator's eventing interface. A special case is only the rendering of frames which in the standalone case still should run as fast as possible, but with a limited rate (e.g., 10 times per second) in simulation mode. Therefore, the event interval has to be chosen differently for the two cases. But all other parts of the application register and run the same way for both standalone and simulation mode.

Another notable difference in simulation mode is in the time. While in normal game implementations all components assume the time base is real time, in a simulated environment it must be made sure that all components use the same time source which is the simulator's virtual time. This is particularly a problem if the game framework provides an own timing source which is the case for Irrlicht. But fortunately, Irrlicht's timer can be re-set to simulation time in each iteration; that easily solves the problem.

A similar issue is random number generation. If it is desired to have an exactly reproducible simulation which is particularly useful for debugging, it is necessary that the application uses random sources provided by the simulator instead of e.g., standard C library functions. If the simulation core is deterministic, the simulator's random generator may be seeded with a constant value, resulting in completely equal simulation runs. The game framework usually does not affect game logic behavior, so it is a realistic assumption that all game logic components only use the simulator's random generation. But since deterministic simulation is currently not a priority and the simulator does not have a random generation interface yet, the current implementation does not provide this feature.

It is rarely of any use for a human player to play the game running in a simulator in non-realtime. Instead, each game instance is equipped with an AI player which controls the game. Still, technically is is possible to give the control for one player to a human even in simulation mode. Game instances may be instantiated with or without a 3D-rendered GUI; when running multiple instances, especially in a simulator, it is not of interest and a waste of resources to render the views of all instances. But just for verifying correct game behavior, it may be desirable to get the view of one player in the completely AI-controlled game. Therefore, there is by default one rendered window to the game in simulation mode.

4 Implementation

This chapter covers all implementation-relevant details, including an architecture overview, detailed component descriptions and usage information.

4.1 Game Architecture

While Planet π4's user interface and game features are mostly unchanged, the architecture has been extensively reworked to enable the simple replacement of various components. Besides the general goal of a modularization of the source code, two aspects are particularly important for this work. First, the network layer interface has to be as general as possible, allowing for different network architectures, including client-server and peer-to-peer concepts. And second, for running the game in a simulator, the timing and eventing facilities have to be exchangeable as a separate module.

Most components interact with each other through interfaces. This interface driven design is the key for exchangeability of functionality. The coupling between modules is reduced using an eventing infrastructure. Any module that is interested in a certain event registers at the Event Manager for the corresponding event type. There are event types for user input, avatar actions (such as fire), and system events.

The main components, as shown with their interdependencies in figure 4.1, are the following:

Eventing The eventing component provides the basic infrastructure for events that are passed between components. The event manager delivers each event fired by some component to all components registered for the particular event type. A component that registers for a particular event type does not need to know about the sender of the event; this ensures low coupling between the components.

Irrlicht Device Provides the main connectivity to the Irrlicht library. Scene setup and rendering is done here. This module and the Irrlicht library is even used when the game is run without a GUI; Irrlicht does not only provide rendering but also the whole 3D world modelling including collision detection. These features are essential to the game mechanics and thus always required.

Timer The Timer provides both the core time source and a facility where components register functions to be invoked regularly (e.g., every 100ms). In the standalone version, timing is provided by the Irrlicht engine; in non-realtime simulations the time comes from the simulator.

Game Core Is responsible for the main game mechanics. It processes user input, keeps a list of avatars of all known players, and processes interactions (e.g., missile fire/hit) between players. It also manages all currently active missiles and explosions.

Network Engine Is responsible for synchronizing the game between all players over the network. It actively polls the Game Core for the avatar position and is informed about asynchronous events (e.g., shots) via the eventing infrastructure. Depending on the network architecture the Network Engine may be completely replaced or certain layers (currently only the group network layer) can be reused.

AI Contains the interface and implementations for bots, i.e., players controlled by artificial intelligence.

The components including important classes and interfaces are described in detail in the following sections.

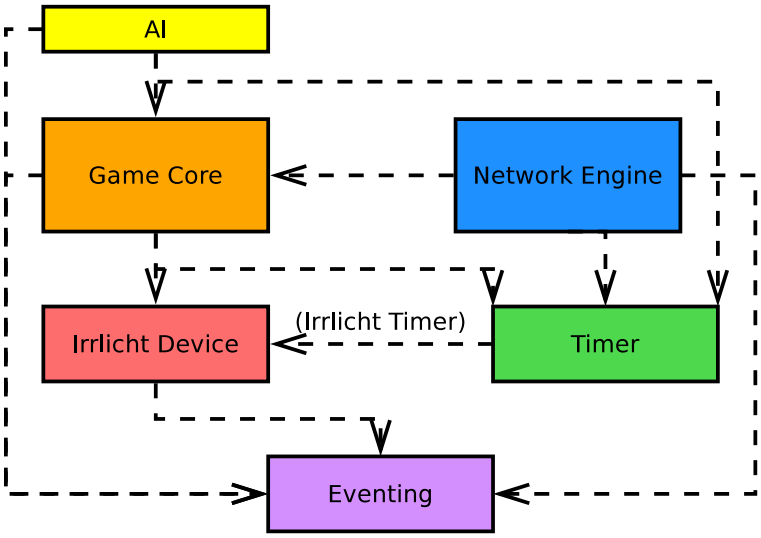

Figure 4.1: Game Component Dependencies

4.1.1 Eventing

The eventing component provides the very basic functionality of event type definition and event distribution. All event instances are encapsulated as objects of classes inheriting from `Event`. The `Event` base class just defines the `EventType` enumeration which contains all of the game's event types. Those are the following:

`SYSTEM` System events are those generated by the operating system. Currently the only system event subtype defined by the class `SystemEvent` is EXIT, which is fired when the application window is closed.

`INPUT` User input events are mapped to this event type. The corresponding class `InputEvent` defines input event subtypes for each possible control command, e.g., ROT_LEFT (rotate left), MOVE_FWD (move forward), FIRE, etc. Key strokes are mapped to input events based on the game's keymap.

`NODE` Node events are those generated by the scene node animation during the rendering phase. Currently the corresponding class `NodeEvent` supports the two subtypes LIFETIME_END and COLLIDE. LIFETIME_END events are fired when an object's time to live has expired. The optional time-to-live setting is used for instance for missiles that have only a certain range they can reach before disappearing. COLLIDE events indicate a collision between objects, e.g., between a missile and a spaceship.

`AVATAR` Avatar events are generated by the game component to pass asynchronous events to the network engine. Those are currently FIRE_BULLET, HIT and DIED. FIRE_BULLET means that the local player has fired a missile. The event contains the position and direction of the corresponding missile so that this information can be delivered to the other players in the network. HIT means that a missile of the local player has hit some other player's ship, and DIED indicates the death of the local player.

An `Event`'s type can be retrieved via its `getType()` method.

Event Manager

Event distribution is managed by the `EventManager` class:

`void registerReceiver(Event::EventType type, EventReceiver* receiver)` Registers the `receiver` for any event of type `type`.

`void unregisterReceiver(Event::EventType type, EventReceiver* receiver)` Unregisters the `receiver` for events of type `type`.

`void fireEvent(Event* event)` Fires the given event. This method (synchronously) invokes the event handler methods of all receivers registered to the event's type.

The `EventReceiver` interface has to be implemented by any class that registers at the event manager. Its only method `bool receiveEvent(Event* event)` is invoked by the event manager for each event of a type which the particular component has registered for.

4.1.2 Irrlicht Device

The `IrrlichtDev` class provides the connectivity to Irrlicht's `irr::IrrlichtDevice` and manages the basic scene setup. `IrrlichtDev` is another core component class that is not accessed through an interface. Since most components of the game depend on the Irrlicht infrastructure, it is not feasible to replace Irrlicht, and thus the `IrrlichtDev` implementation does not need to be replaced either.

The important interface methods of `IrrlichtDev` are documented in the following:

`IrrlichtDev(EventManager* evtMgr, bool createUI)` Creates the Irrlicht device and builds the basic game scene. `evtMgr` refers to the game's Event Manager, and the `createUI` flag specifies whether a window with 3D rendering (i.e., the GUI) should be created.

If `createUI` is `true`, OpenGL or Direct3D rendering is used for Linux and Windows respectively. If `createUI` is `false`, the `irr::IrrlichtDevice` is still instantiated since it provides functionality that is also necessary in non-GUI mode. But it uses the *null* video driver resulting in no video output. The `createUI` parameter also affects texture loading. Where possible, texture loading is omitted to save memory in non-GUI mode.

`irr::IrrlichtDevice* getDevice()` Returns the `irr::IrrlichtDevice` instance.

`bool hasUI()` Returns whether GUI is enabled, i.e., the value of `createUI` as passed to the constructor.

`bool isRunning()` Returns whether the Irrlicht device is still running. This value may become `false` when the rendering window has been closed by the user or the operating system. It can also be set to `false` by the application using `closeDevice()`. The value should be checked in each iteration of the main loop, especially before calling `renderScene()`. If `false`, the application should terminate.

`bool hasFocus()` Returns whether the Irrlicht rendering window is the currently active (focused) window.

`void renderScene()` Renders the scene. This method has to be called periodically from the main loop. The the rendering step, Irrlicht also invokes the animation of all nodes in the scene; thus collision detection is also performed in this step.

`void closeDevice()` Sets the return value of `isRunning()` to `false`.

`void attachCamera(Entity& entity)` Attaches the camera (i.e., the view of the rendered frame) to the given Entity

`void setStatusMessage(const string& category, const string& message)` Sets status messages that are shown as a head-up display (HUD) on the rendered frames (see figure 4.2). For each `category` there can be one message. Calling this method a second time with the same category

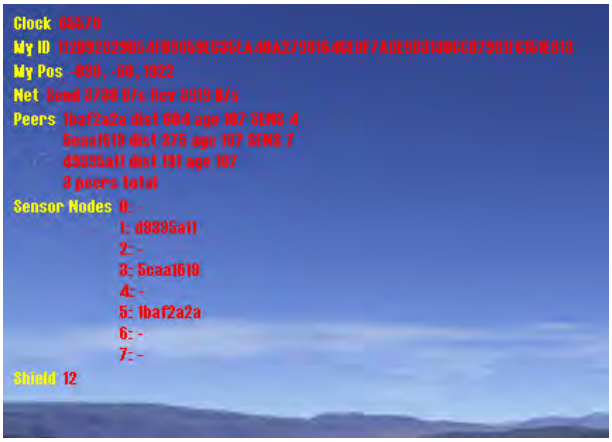

Figure 4.2: In-game status message display (HUD)

will overwrite the previous message for that category. `category` is displayed as the caption of the corresponding message. Passing an empty string to `message` will remove the message for the given category.

Entity

Each object in the game's 3D world is represented by an `Entity` object. An `Entity` encapsulates an Irrlicht scene node (`ISceneNode`) and provides basic manipulation of that scene node. The most important extra feature added by `Entity` is animation including collision detection.

Entity animation provides basically two modes: absolute and relative. For both translatory and rotary movement, absolute animation translates/rotates the scene node with a constant velocity to the destination position. When that position has been reached, the movement stops. For relative animation, there is a direction vector and a velocity describing the movement. The animation does not stop until it is explicitly aborted, for instance using `setPosition()` and `setRotation()` respectively.

`Entity` has the following public methods:

`const vector3df& getPosition() const` Returns the corresponding scene node's current position.

`const vector3df& getDestPosition() const` Returns the destination position (if movement is absolute) or direction (if movement is relative). See also `movementIsAbsolute()`.

`const vector3df& getRotation() const` Returns the corresponding scene node's current rotation.

`const vector3df& getDestRotation() const` Returns the destination rotation (if movement is absolute) or direction (if movement is relative). See also `movementIsAbsolute()`.

`const s32 getPosSpeed() const` Returns the translational speed (units/sec) of the current animation.

`const s32 getRotSpeed() const` Returns the rotary speed (units/sec) of the current animation.

`const bool movementIsAbsolute() const` Returns whether the current animation (and destination position) is absolute (`true`) or relative (`false`).

`const vector3df& getScale() const` Returns the scene node's scale.

`void setPosition(const vector3df& pos)` Sets the scene node's position and stops its movement.

`void setRotation(const vector3df& rot)` Sets the scene node's rotation and stops its rotation.

`void moveTo(const vector3df& pos, s32 speed = -1)` Moves the scene node to the (absolute) position `pos` with the given speed (units/sec). The default speed of `-1` means immediate positioning. Unlike `setPosition()`, this method does not affect the position before the next animation step.

`void moveBy(const vector3df& direction, s32 speed = -1)` Moves the scene node in the given `direction` with the given speed (units/sec if the length of `direction` is 1). The movement will continue until explicitly stopped. Also, the direction vector is cumulative. Subsequent calls to this method will add the direction vector; use `setPosition()` to stop the movement.

`void rotateTo(const vector3df& rot, s32 speed = -1)` Rotates the scene node to the (absolute) rotation `rot` with the given speed (units/sec). The default speed of `-1` means immediate rotation. Unlike `setRotation()`, movements by this method do not affect the rotation before the next animation step.

`void rotateBy(const vector3df& angles, s32 speed = -1)` Rotates the scene node with the given `angles` at the given speed. Rotation will continue until explicitly stopped. Also, the angles vector is cumulative. Subsequent calls to this method will add the angles vector; use `setRotation()` to stop the rotation.

`void setScale(const vector3df& scale)` Sets the scene node's scale.

`void setTtl(s32 ttl, EventReceiver* receiver)` Sets the entity's time to live (`ttl`; in milliseconds). When the time to live has expired, a `NodeEvent` of subtype LIFETIME_END is sent to `receiver`. It is the receiver's responsibility to delete the entity; otherwise the entity will not disappear.

`void collideWith(const Entity& entity, EventReceiver* receiver)` Enables collision detection among this entity and the specified `entity`. Subsequent collisions will generate `NodeEvents` with subtype COLLIDE which are passed to the `receiver`.

`irr::s32 getId() const` Returns the entity's id. The id is unique within the local application.

4.1.3 Timer

The timer component is in principle a simple helper component. The main reason for having it in a separate component is the fundamental need for replaceability of the timer functionality. While the standalone (and realtime) version of the game uses the timing facilities from the Irrlicht engine which constantly runs in real time, in the discrete-event simulation, the game is not supposed to run in real time, and thus all components have to use the (virtual) time value provided by the simulation core. Accessing time information only through the common `GameTimer` interface allows for an easy exchangeability without the need for modifying any other game component.

The timer component in fact does not only provide the common timing facility but also contains the application's main loop. This is because in discrete-event simulation, the main loop cannot be part of the simulated application but has to be in control of the simulator. Thus the main loop provided by the timer component may either be a busy loop, rendering frames as fast as possible for standalone mode, or it is passed to the simulator by calling the simulator's main event loop.

The `GameTimer` interface is the following:

`void registerTask(GameTask* task, int ms)` Registers a task that is to be run periodically every `ms` milliseconds. The exact rate is not guaranteed, but provided on a best effort basis. In realtime mode, tasks taking a long time to compute (e.g., the rendering of a frame) may (and usually do)

block other tasks, resulting in a far from constant rate. In discrete-event simulation mode, the rate is usually met exactly based on virtual time.

The `GameTask` interface has only the one method `int executeTask()` which is called periodically to execute the task. The return value of `executeTask()` indicates whether the task was successful. A value of zero (the constant GAME_TASK_OK) means success, any other value (generally GAME_TASK_FAIL) will abort the main loop (see `run()`).

`int run()` Runs the main loop. This method does not return before the main loop is aborted, which can happen for three reasons:

- the `executeTask()` method of a `GameTask` returns a value different from GAME_TASK_OK in which case that return value is also the return value of `int run()`,
- the `abort()` method is called in which case the return value is `run()` is GAME_TASK_OK, or
- the loop is aborted for internal reasons (e.g., the Irrlicht device has been closed) in which case the return value is GAME_TASK_ABORT.

`void abort()` Aborts the main loop. The consequent return value of `run()` is GAME_TASK_OK.

`int getTime()` Returns the current time in milliseconds. The time base is usually the start of the program, but this may depend on the timer implementation.

Timer Implementations

There are currently two implementations of the `GameTimer` interface, `IrrlichtTimer` and `BSTimer`.

`IrrlichtTimer` is the standalone game timer. Its timing source is the Irrlicht device which provides realtime in millisecond resolution starting at zero when the application starts. But the time value is only increased when a frame is rendered, which requires some special treatment. A `GameTask` rendering frames must be scheduled with interval 0 so that the timer will continue running. This also results in frames being rendered with a frequency that is only limited by the hardware. Any other tasks are executed between the rendering of two frames, as in conventional single-threaded games.

`BSTimer` is the timer implementation for non-GUI and/or simulation mode. It uses the eventing main loop from the BubbleStorm/CUSP library. Depending on the configuration of the linked library, the event loop may either process events in real time using real sockets for communication, or it can run in a purely simulated environment, processing events as fast as possible using a virtual time. In both cases, the `BSTimer` class does the same thing. Registered tasks are directly registered with the simulator's scheduler. The `run()` method just starts the main loop provided by the library which in turn can be provided by the transport layer for realtime mode or by the simulator. Finally, the `getTime()` method returns the time provided by the library which again may be either real time or virtual time.

The use of `BSTimer` has the advantage of running the game purely event-based and thus avoiding any busy wait. But it depends on the BubbleStorm/CUSP library and thus cannot be used with other transport protocols. `IrrlichtTimer` is necessary for achieving the conventional game behavior consuming the whole computational power for rendering as many frames per second as possible.

4.1.4 Game Core

The game core contains the main game mechanics. `GameInstance` is the interface of the main class of the game core. Its most important tasks are the handling the local player's and all known neighboring players' avatars (ships) and their interactions.

The game core receives information from various sources. User inputs are transformed to avatar actions. The other players' avatar positions are updated by the network engine; events from the latter notify the game core about asynchronous game events like the firing of missiles. Collisions of objects (e.g., missiles hitting a ship) are passed via events by the collision detection of the rendering component.

Therefore, `GameInstance` has the following methods:

`void init(IrrlichtDev* device, GameTimer* timer, EventManager* evtMgr)` Initializes the game. Registers for the events that are of interest, registers with the timer, and creates the local player's avatar.

`Avatar* getAvatar()` Returns local player's avatar.

`PlayerList::iterator getKnownPlayersBegin()` Returns the begin of the remote player's known avatar list.

`PlayerList::iterator getKnownPlayersEnd()` Returns the end of the remote player's known avatar list.

`Avatar* getKnownPlayer(const string& name)` Retrieves the avatar object for the player with the specified name. Returns `null` if no player with the specified name is known.

`Avatar* addKnownPlayer(const string& name, const vector3df& pos, const vector3df& rot)` Adds an avatar for a newly known player. Returns the created avatar object.

`void removeKnownPlayer(const string& name)` Removes an avatar from the known player list.

`Bullet* createBullet(const vector3df& pos, const vector3df& dir, s32 speed)` Creates a bullet (missile) at the given position moving with the given speed in the given direction. This method is to be called by the network engine for creating remote player's missiles (corresponding to the network message FIRE).

`void hit(const string& by)` This method is called by the network engine when a remote player has hit the local player's ship (corresponding to the network message IHITYOU). The local player's hit points will be decreased accordingly and if it has reached zero, the ship explodes. `by` specifies the remote player's name.

`Explosion* explodeShip(const string& who)` The network engine calls this method when a remote player's ship explodes (corresponding to the network message IDIED). An explosion will be generated at that player's position. `who` specifies the remote player's name.

`void setStatusMessage(const string& category, const string& message)` Sets a status message that is passed to `IrrlichtDev::setStatusMessage()`. This method is there for convenience reasons; if the `IrrlichtDev` has not yet been initialized/set, the status messages are buffered and passed to `IrrlichtDev` later.

`GameInstance::PlayerList` is defined as `map<std::string, Avatar*>`. It maps the player names to the corresponding avatar objects.

The class implementing the `GameInstance` interface is `ImplGameInstance`. Although there could be an alternative implementation of `GameInstance`, the implementation is rather straightforward and the interface does not really allow for completely different game mechanics. However, defining an interface still loosens the coupling, making the points of interaction more explicit.

Avatar

An abstract representation of an avatar, either belonging to the local or a remote player, is provided by the interface `Avatar`. The interface defines the basic methods for avatar positioning, which are mostly equivalent to the equally named `Entity` methods:

`const vector3df& getPosition() const` Returns the avatar's current position.

`const vector3df& getRotation() const` Returns the avatar's current rotation.

`const vector3df& getDestPosition() const` Returns the avatar's destination position (if absolute movement) or direction (if relative movement).

`const vector3df& getDestRotation() const` Returns the avatar's destination rotation (if absolute movement) or direction (if relative movement).

`const s32 getPosSpeed() const` Returns the avatar's current translational speed (units/sec).

`const s32 getRotSpeed() const` Returns the avatar's current rotary speed.

`const bool movementIsAbsolute() const` Returns whether the current destination is absolute (`true`) or relative (`false`).

`void setPosition(const vector3df& pos)` Sets the avatar's position and stops its movement.

`void setRotation(const vector3df& rot)` Sets the avatar's rotation and stops its rotation.

`void moveTo(const vector3df& pos, s32 speed = -1)` Moves the avatar to the (absolute) position `pos` with the given speed (units/sec). The default speed of `-1` means immediate positioning. Unlike `setPosition()`, this method does not affect the position before the next animation step.

`void moveBy(const vector3df& direction, s32 speed = -1)` Moves the avatar in the given `direction` with the given speed (units/sec if the length of `direction` is 1). Movement will continue until explicitly stopped. Also, the direction vector is cumulative. Subsequent calls to this method will add the direction vector; use `setPosition()` to stop the movement.

`void rotateTo(const vector3df& rot, s32 speed = -1)` Rotates the avatar to the (absolute) rotation `rot` with the given speed (units/sec). The default speed of `-1` means immediate rotation. Unlike `setRotation()`, movements this method does not affect the rotation before the next animation step.

`void rotateBy(const vector3df& angles, s32 speed = -1)` Rotates the avatar with the given `angles` at the given speed. Rotation will continue until explicitly stopped. Also, the angles vector is cumulative. Subsequent calls to this method will add the angles vector; use `setRotation()` to stop the rotation.

`irr::s32 getId()` Returns the object id which is unique within the application. Note that this value does not identify the player across nodes.

Spaceship

The `Spaceship` class integrates the `Avatar` interface with `Entity`, connecting `Avatar` methods to `Entity`'s functionality (see figure 4.3). Furthermore, it is responsible for the spaceship 3D model.

From the point of view of the Irrlicht component, `Spaceship` is an `Entity`, thus encapsulating a scene node and being animated. From the point of view of the game core, it is primarily seen through the

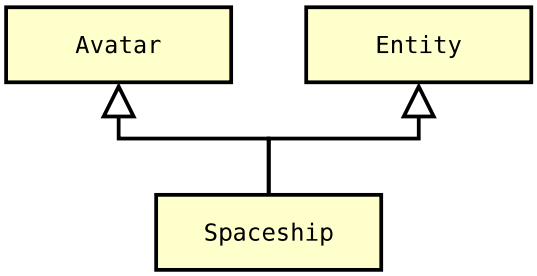

Figure 4.3: `Spaceship` class inheritance

`Avatar` interface for being moved through the game world. `Spaceship` instances are used for both the local player's and remote players' avatars just with the difference of the source of control.

4.1.5 Network Engine

The focus in this work's implementation is on the network engine. The network engine is the component that makes the game a multiplayer game. Compared to its complex task, the main interface for a network engine implementation, `NetworkEngine`, is rather simple:

`void init(GameTimer* timer, EventManager* evtMgr, GameInstance* instance)` Sets up the network engine. `timer` is typically used to register for regular position update processing. `evtMgr` delivers the game core's asynchronous user events (e.g., fire) to the network engine. `instance` is the game instance (i.e., game core object) which is both polled for the player's state and pushed with remote player updates.

`void configureUser(irr::IrrlichtDevice* device)` Optionally performs an interactive user configuration (e.g., via a dialog). `device` is supposed to be used for creating the GUI. In non-GUI mode, it may be `NULL`; in that case interactive configuration may be done via the console.

`bool connectSync()` Synchronously connects to the network. This method blocks until the connection is established (i.e., the game can begin) or connection failure. Returns `true` on success, `false` otherwise. Because of its blocking behavior, this method cannot be used in discrete-event simulation mode. For that purpose there is `connectAsync()`.

`void connectAsync(ConnectCallback cb, void* userData)` Virtually does the same as `connectSync()`, just in an asynchronous manner and thus usable in simulation mode. The callback function `cb` is defined as `void (*ConnectCallback)(bool result, void* userData)`.

`result` informs about the success and corresponds to the return value of the synchronous version. `userData` allows passing a user-defined context pointer.

The typical interaction of the network engine with the other game components is shown in figure 4.4. Concrete implementations of `NetworkEngine` are described in section 4.2.

4.1.6 Artificial Intelligence

The artificial intelligence (AI) component provides the infrastructure for bots, i.e., players controlled by artificial intelligence. Basically there is just one simple interface that AI implementations have to stick to, namely `GameAI`. `GameAI` has just one method:

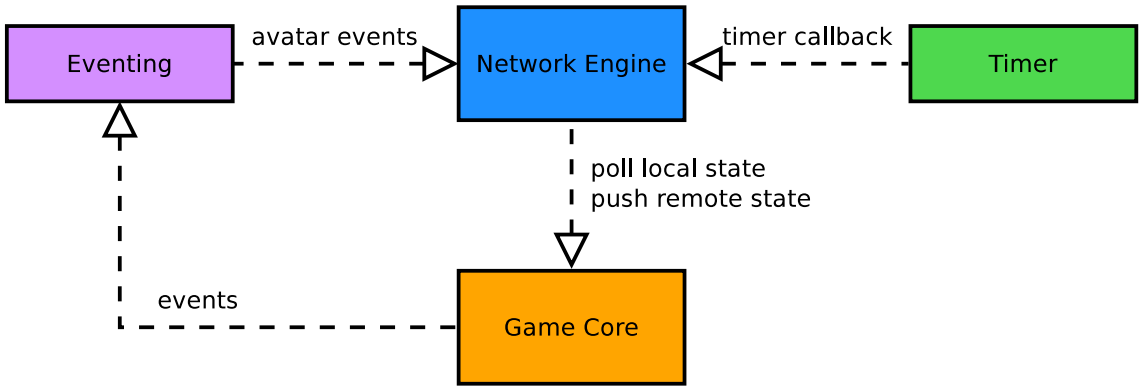

Figure 4.4: The network engine's information flow with other components

`void init(GameInstance* game, GameTimer* timer, EventManager* evtMgr)` Initializes the AI module. `game` is supposed to be used by the AI implementation to retrieve the current game state, i.e., the local avatar and the avatars of nearby remote players. Thus, `game` is the one and only game information source for the AI implementation to take decisions. As any AI implementation has to regularly run its decision process, `timer` is used to register for being called e.g., every 100 milliseconds. `evtMgr` may be used for both receiving events that are of interest and firing events. Events of interest may be for instance, collision or missile fire events. Events generated by the AI are most commonly emulated input events for ship control.

This interface gives AI implementations a lot of freedom both in perception of other players and in control of the own ship. Thus an AI implementation may easily 'cheat' either by using more information than a human player could perceive or by jumping anywhere in the world without speed limitations. Although these properties are obviously undesired for serious bots, it may be useful to have this potential for testing purposes.

Simple AI Implementation

A simple AI implementation developed for this work is `SimpleAI`. It has basically just the simple directive to follow the nearest neighbor and fire missiles when there is a chance to hit. There is no support for any strategy or the concept of friends, thus the AI implementation is so far only usable for deathmatch gameplay mode.

`SimpleAI` runs its AI processing routine once per 50 milliseconds. The routine has basically three processing steps:

1. *Select target player to aim at.* The algorithm currently used is very simple. Of all known players the one is chosen that has the smallest euclidean distance. The point of reference (i.e., the one from which the distance is measured) is not the own ship's center but a point 1000 units in front if it. This optimization results in a preference for players in front of the own ship which are easier to aim at, avoiding frequent and unnecessary turnarounds.

   There could be various simple optimizations in target player selection that may further improve the bot's effectiveness. For instance the selected player could be memorized between the processing rounds to avoid too frequent target changes. Another optimization could be concerning the distance metric. Ships that are in the center of the crosshairs should be preferred even if they are farther away because they require less movement (and thus time) for aiming.

2. *Aim at the selected target player.* The algorithm calculates the angles (horizontal and vertical) between the own ship's current direction and the the difference vector between the own and the

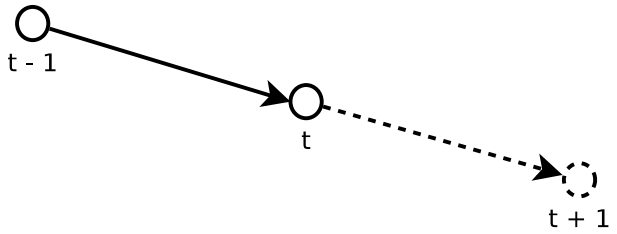

Figure 4.5: AI simple motion prediction

target ship. Based on these values, the ship is controlled to turn horizontally and vertically respectively.

As all ships are typically moving continuously, it is important to have some kind of motion prediction in order to gain an acceptable hit ratio. The most simple approach compensates the prediced motion of the target ship in the time interval between the current and the following processing round. The motion vector between these two rounds is assumed to be the same as the vector between the previous and the current round (figure 4.5).

`SimpleAI` has a more advanced motion prediction which becomes necessary because of the relatively low velocity of the missiles. Thus depending on the distance to the target ship, it is necessary to aim at a point more or less far away in front of the ship. Otherwise the ship will likely have moved away before the missiles can reach it. The following formula describes the calculation for the aim point (P_{aim}; P_t is the current known target position, $P_{t-\Delta t}$ is the target position from the previous round, ms_{calc} is the processing round interval in milliseconds, d is the distance to the target, and $v_{missile}$ is the missile velocity in units per second; see also figure 4.6):

$$P_{aim} = P_t + \left(P_t - P_{t-\Delta t}\right) * \frac{1000}{ms_{calc}} * \frac{d}{v_{missile}}$$

The ship is set to a constant forward movement; there is no need for stopping. Due to practical and testing reasons, the AI ships move only at half the speed of those controlled by human players, but this may be changed when necessary.

This method is still not completely accurate, but tests have shown that the bots using this method perform well enough to generate some (more or less) realistic game workload.

3. *Fire missiles when there is a chance to hit.* Fire is triggered when both the horizontal and the vertical angle to the target are less than a certain threshold. As the fire rate is not limited by the game engine, the AI limits it to at most once per 4 rounds, resulting in at most one shot per 200 milliseconds.

4.2 Network Models

As described earlier, all network model implementations share the common interface `NetworkEngine`. Currently there are two implementations, `GroupNetworkEngine` and `P2NetworkEngine`. The former is a general group multicast network engine, i.e., a wrapper for any underlaying network supporting group management and multicast. Both the TCP and CUSP client-server models use the `GroupNetworkEngine`. `P2NetworkEngine` is the self-contained implementation of the (simplified) pSense protocol.

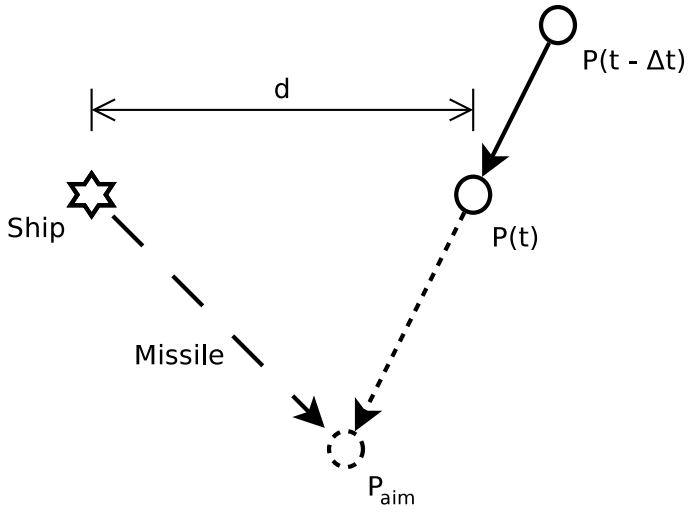

Figure 4.6: AI motion prediction with missile flight compensation

Figure 4.7: Screenshot of a simulated peer-to-peer game with 16 bots

4.2.1 Group Based Networking

The group networking wrapper consists of two classes. `GroupNetworkEngine`, which has already been introduced, is the `NetworkEngine` implementation. The second class, `GroupNetwork`, is the abstract base class for the concrete network implementations.

Classes deriving from `GroupNetwork` have to implement the following methods:

`GroupPtr createGroup()` Creates a new broadcast group and returns a pointer to its ID.

`void joinGroup(GroupPtr group)` Joins a group given by its ID. After a successful join the local player may send messages to the group and receive messages from the group.

`void leaveGroup(GroupPtr group)` Leaves a group given by its ID.

`vector<UserPtr> listGroupMembers(GroupPtr group)` Lists all members of the group given by its ID.

`GroupPtr createGroupID(string id)` Creates a group ID from the given string. The method may fail if the string provided cannot be converted to a group ID.

`void receiveData(vector<UserPtr>& users, vector<string>& messages)` Receives (i.e., polls) data from the network. The two vectors passed as parameters have to be filled synchronously with tuples of the sender's user ID and the message string.

`void transmitToUser(UserPtr user, string message)` Sends a message to a specific user.

`void transmitToGroup(GroupPtr group, string message)` Sends a message to the specified group.

`void processData()` This method is already implemented by the base class `GroupNetwork`. It is invoked periodically by the network engine and calls `receiveData()`. It may be overwritten to perform specific action before or after the original `processData()`.

On failure the above methods' implementations should throw exceptions, usually instances of the class `string`.

Each group network implementation has to provide two additional classes implementing the interfaces `NetworkGroupID` and `NetworkUserID`. The two are used to identify groups and users respectively, allowing for any kind of internal ID representation. Both have to implement the `string toString()` method allowing the application to present a human-readable form. `NetworkUserID` also defines an `equals()` method allowing the application to compare two instances for equality. `GroupPtr` and `UserPtr` in the `GroupNetwork` interface above are pointers to the two interface classes.

The creation and processing of messages is performed by `GroupNetworkEngine`, thus all underlaying `GroupNetwork` implementations use the same ASCII message format. Table 4.1 lists all messages that `GroupNetworkEngine` multicasts within groups.

4.2.2 Client-Server over TCP

The TCP client-server implementation is based on the group multicast networking model. Its implementation using the `GroupNetworkEngine` implementation is quite straightforward, as the text-based messaging scheme, as presented in 3.2, is designed to be mapped directly to the `GroupNetwork` interface. User and group IDs are plain strings with no spaces. Just the in-band signaling of client-server messages requires some extra functionality filtering out group messages while waiting for a server response.

The standard Berkeley sockets API does not provide asynchronous operations using callbacks; particularly for reads, sockets have to be polled. Thus the socket for the server connection is regularly checked for data to read using the `select()` function. Although the polling does not seem very efficient, it has the advantage that incoming messages are only received at defined states of the processing loop.

Table 4.1: Message types used by `GroupNetworkEngine`

Message	Description
SYSTEM JOIN	Join the system. This message is sent repeatedly to make sure all participants are aware of the node.
SYSTEM ACKJOIN	Acknowledge a join. Sent as an answer for JOIN when received for the first time.
SYSTEM LEAVE	Indicates that the node will leave the system.
SHIP *PosX PosY PosZ RotX RotY RotZ*	Position update. *Pos*{*XYZ*} and *Rot*{*XYZ*} are placeholders for the new ship position and rotation represented as decimal floating point numbers.
FIRE *PosX PosY PosZ DirX DirY DirZ Spd*	Missile fire. *Pos*{*XYZ*} and *Dir*{*XYZ*} stand for the Missile's initial position and direction, *Spd* is its velocity.
IHITYOU *PlayerId*	Indicates that a missile has hit a remote player. *PlayerId* is the hit player's id (name).
IDIED	Sent when a player's ship explodes.

Server

The TCP server class is `TCPServer`. Its interface is simplistic:

`TCPServer(int bindPort = 8585)` Creates a server configured to listen at the specified TCP port.

`void run()` Starts the server. This method blocks until the server is aborted.

`void abort()` Aborts the server.

The server implementation mostly consists of networking code. Initially it creates one socket listening for incoming connections. In its main loop it monitors the listening socket and all open connections using the `select()` function. This approach limits the total number of connections to `FD_SETSIZE - 1`. But the default value of `FD_SETSIZE` is 1024 for Linux[1] which is assumed to be sufficient. The server keeps a list of groups and for each group a list of clients. Group broadcasts are simply accomplished by iterating over the list of clients and sending the broadcast message.

4.2.3 Client-Server over CUSP

The CUSP client-server implementation (class `CUSPClientServerNet`) does in principal the same as the TCP implementation. Even though the CUSP API is very different to the Berkeley sockets API, the implementation is very similar to `TCPClientServerNet`. This is because `CUSPClientServerNet` uses a wrapper class, `CUSPClient`, whose interface is similar to the socket wrapper class used in the TCP version. `CUSPClient` in turn uses the two CUSP convenience classes `BiDiConn` and `SingleCycleConn` (see section 4.6).

`CUSPClient` has the following interface:

`void connect(int localPort, const string& remoteAddr)` Connects to the server at the given remote address (in the form "<*host*>" or "<*host*>:<*port*>"), locally binding to UDP port `localPort`.

`void send(const string& msg)` Sends a message to the server.

[1] Current Linux versions define `FD_SETSIZE` in */usr/include/linux/posix_types.h*

`string recv()` Receives a message from the server. Blocks as long as no complete message is ready to be returned.

`bool isReadyRecv()` Returns whether a message can be read without blocking.

`string requestResponse(const string& request)` Synchronously performs a request-response cycle, i.e., sends the request and waits for the response from the server. Each call creates a new pair of streams with a high priority, thus doing out-of band signaling and avoiding problems like head-of-line blocking.

The synchronous API provided by `CUSPClient` is much easier to use than CUSP's completely asynchronous and callback-based API. But it also has substantial limitations. Most importantly it is not usable with the simulator as the latter does not allow for blocking operations. But as the simulation of the client-server version is not in the focus of this work, that seems acceptable.

Server

The CUSP server's group handling is identical to the TCP server's. Also, the interface of the CUSP server class `CUSPServer` equals the `TCPServer`'s (see 4.2.2). The differences are again in the networking part. Unlike the client, the CUSP server cannot use a blocking API since it has to handle multiple clients simultaneously. Thus the server uses the CUSP API directly; for each connection the server holds an `InStream` and an `OutStream`. Just the request handling is managed by the `SingleCycleServer`, the server side correspondent to `SingleCycleConn`. Refer to section 4.6 for more information about CUSP usage patterns and helper classes.

4.2.4 Peer-to-Peer over CUSP

The peer-to-peer networking implementation is, being in the focus of this work, considerably more sophisticated than the client-server implementations. As denoted before, the algorithm is a slightly simplified and modified version of pSense (section 2.3.6). The simplification mainly consists of the omission of update message forwarding. Since that is an optimization for low bandwidth users and the behavior with limited bandwidth is not analyzed here, it seemed feasible to leave this feature out. Though it may be added later if necessary. Another difference to the original pSense protocol is that the implementation is not round-based. Updates are sent in fixed time intervals but to achieve delays as low as possible, incoming messages are processed immediately.

The peer-to-peer network engine is implemented in the class `P2NetworkEngine`. Its constructor is `P2NetworkEngine(int bindPort, const string& connectAddr,`
specifying the local UDP bind port and the peer address to connect to for bootstrapping. `P2NetworkEngine` keeps a map containing all logical connections to peers, each represented by an instance of the internal class `PeerConn`. The keys of the map are peer names, i.e., their host's public keys. The map allows assigning new incoming streams to existing `PeerConn`s.

`PeerConn` keeps two outgoing streams and any number of incoming streams (typically there also two incoming streams because the opposite end has two outgoing streams). One of the two outgoing streams is used for regular position update messages. The other has a higher priority and carries all other types of mesages (e.g., sensor node requests). Additionally, `PeerConn` keeps a reference to the corresponding player's avatar, updating its position whenever a position update has been received. Each message that is received sets the last-seen-timestamp to the current time. When no message is received for a certain time (currently 5 seconds), the peer is assumed to be dead and the connection (including all incoming and outgoing streams) is closed.

Table 4.2: Message types used by the peer-to-peer network engine

Type	Type ID	Description
POSITION	0x01	Position update.
INTRODUCE	0x04	Introduces a new player that has entered the target player's vision range.
SENSOR_REQUEST	0x08	Sensor node request. Tells the destination node to become sensor node for the sender.
SENSOR_SUGGEST	0x09	Suggests a node as new sensor node for a particular sector.
FIRE_BULLET	0x81	Indicates a fired missile.
IHITYOU	0x82	The destination player is hit by a missile from the sending player.
IDIED	0x83	The sending player's ship explodes.

Messages

The peer-to-peer protocol has a binary message format reducing the required bandwidth and processing power compared to the ASCII messages of the client-server version. But position update messages contain more fields, allowing for a basic dead reckoning. Besides position and rotation, update messages also contain the velocities of position and rotation. According to these values, the remote player's ships are animated in the time period between two update messages, resulting in a much smoother and more precise ship movement. Each message has a 4 byte header including two 16 bit fields specifying the size (in bytes) and type type of the message. All message types are listed in table 4.2.

POSITION messages are sent regularly (every 100 milliseconds) to all known peers for which at least one of the following conditions is met (assume A is the sender and B is the potential receiver):

1. B is in A's vision range.
2. A is in B's vision range. (Position update messages contain a field specifying the sender's (current) vision range.)
3. B is in A's sensor node list.
4. A is sensor node for B.

Position update messages have a total size of 64 bytes, containing the following information:

- 3-dimensional position (12 bytes)
- 3-dimensional rotation (12 bytes)
- 3-dimensional direction of translatory motion (12 bytes)
- 3-dimensional direction of rotatory motion (12 bytes)
- Velocity of translatory motion (4 bytes)
- Velocity of rotatory motion (4 bytes)
- Vision radius (4 bytes)

Both message types INTRODUCE and SENSOR_SUGGEST have a variable length body each containing two null-terminated ASCII strings. The first string is the hexadecimal representation of the introduced/-suggested peer, the second is its address. In fact, both message types are currently handled the same way. The receiver first contacts the referred peer (if not already known) to receive the position. After that, it decides whether the peer is a sensor node candidate or already in the vision range. If none of the two conditions are met, the connection becomes idle and is shut down after the timeout of five seconds.

The SENSOR_REQUEST message carries only one field indicating the number of the sector which the receiver is requested to become sensor node for. FIRE_BULLET, IHITYOU and IDIED are similar to their client-server correspondents. FIRE_BULLET carries the position, direction and velocity of the bullet. IHITYOU does not have any body fields since messages of this type are sent directly between the two involved peers. IDIED does not have any body fields either but is, like FIRE_BULLET, sent to all known peers.

Since CUSP is a reliable transport protocol, it is unnecessary to send acknowledge messages for the overlay operations. The connection management is kept pretty simple. Connections are kept open as long as messages are received from the corresponding peer. All connections are regularly (once per second) checked for the timestamp of the last received message. If no message has been received within the timeout interval (5 seconds) on a particular connection, that connection is immediately closed. This mechanism works because the conditions for sending position update messages are symmetric. As long as node A is interested in the position of player B, it keeps sending position updates to that player. When the arrival of position update messages from B ceases, A can assume that B either has crashed or is not interested in A's updates anymore.

4.3 Main Program

The previous sections described the various components and depicted the interaction between the components. This section briefly explains how the main program composes the components.

4.3.1 Standalone Game

The standalone game application's entry point is defined in *main.cpp*. The `main()` function first processes possible command line parameters which are currently the only way for setting runtime parameters. Available parameters are listed in section 4.5.3. Many more variations are currently only possible via source code modification; a good starting point is *config.h* (see below).

The game is set up by instantiating the components and connecting them via their initializer methods. The following C++ code extract is a simplified (but working) version:

```
// create core components
EventManager* eventMgr = new EventManager();
IrrlichtDev* device = new IrrlichtDev(eventMgr, true);
ImplGameInstance* game = new ImplGameInstance();
GameTimer* timer = new IrrlichtTimer(device);

// set up the network
NetworkEngine* networkEngine = new P2NetworkEngine(localPort, serverAddr);
networkEngine->init(timer, eventMgr, game);
networkEngine->configureUser(device->getDevice());
if (!networkEngine->connectSync()) {
    printf("Connect failed!\n");
    return EXIT_FAILURE;
```

```
}

// init game
game->init(device, timer, eventMgr);

// set up AI (optional)
GameAI* = new SimpleAI();
ai->init(game, timer, eventMgr);
```

Finally, the main event processing loop provided by the timer component is started:

```
timer->run();
```

The `run()` method blocks until the game is aborted. When it returns, all game objects are cleaned up and the program exits.

An alternate network engine can be used simply by instantiating another implementation. For instance, the client-server network for the group network engine is set up using the following two lines:

```
GroupNetwork* network = new CUSPClientServerNet(localPort, serverAddr);
NetworkEngine* networkEngine = new GroupNetworkEngine(network);
```

4.3.2 GUI-less Setup

If the game is to be run without a 3D rendered GUI, two minor changes are necessary. Most importantly, `IrrlichtDev` has to be instructed to omit the setup of 3D rendering by setting the second parameter of its constructor to `false`:

```
IrrlichtDev* device = new IrrlichtDev(eventMgr, false);
```

Second, it is advisable to use the `GameTimer` implementation for the BubbleStorm/CUSP library to get rid of the busy-wait main loop. Of course, this only works if CUSP is used as the transport protocol.

```
GameTimer* timer = new BSTimer();
```

4.3.3 Further Configuration

Some further configuration settings for various components are stored in *config.h*. Currently these are primarily the locations of media files (3D models and textures) for the rendering engine.

4.4 Simulation

To run the game in the BubbleStorm/CUSP simulator, it is not built as an executable but as a static library to be linked to the simulator. The library is built without *main.cpp* but instead with *simnode.cpp* containing the entry point for the game. Instead of a `main()` function, there is `void startNode(const char* peerAddr, const char* gatewayAddr)`.

The `startNode()` function initializes a single node game instance in a way very similar to the main function of the standalone game version. The two function parameters specify the peer addresses for connecting to the virtual peer-to-peer network. `peerAddr` specifies the peer to connect to, and `gatewayAddr` optionally specifies a gateway node that helps with the connection to the peer if that peer is behind a NAT in the simulated network. In the simulator, addresses are not IP addresses but just node IDs. But as long as the application does not expect a particular addres format, the CUSP simulator keeps the differences to real networking completely transparent to the application.

Thus, the network initialization code can be exactly the same as in real networking mode, with the only difference that blocking operations are not allowed. The only case in which this currently matters is the process of connecting to the network which is executed using `NetworkEngine::connectSync()` for convenience reasons in the standalone game example above. For use with the simulator it is necessary to use the non-blocking version `NetworkEngine::connectAsync()` (which can also be used in standalone mode). Any operation that needs to be executed after a successful connect has to be moved to `connectAsync()`'s callback function. The corresponding C++ code (*simnode.cpp*) is included in appendix A.1.

4.4.1 Simulator Configuration

The `startNode()` function is not invoked directly by the simulator environment. Instead, there is some Standard ML code that configures the simulator and calls `startNode()` imported from the C++ library. The code that connects the game with the simulator (*simulator.mlb*) is listed in A.2. Additionally there is the file *setup.sml* which configures the different simulator components, including:

Simulated networking. The simulated UDP networking component replaces the real networking using sockets. A CUSP transport instance is created on top of the simulated UDP network.

Scenario and node creation. The scenario, i.e., the number and distribution of the nodes including their connection properties, is loaded from a SQLite[2] database. A node factory automatically creates nodes corresponding to the scenario configuration.

Logging facilities. The simulator's logging facilities allow for a detailed logging that may be post-processed for evaluation. For instance, the logging output could be written to an SQL database where it is possible to query for certain events.

A detailed documentation on the simulator is beyond the scope of this work. Refer to the CUSP sources[3] for more information.

4.5 Usage

This section briefly explains how the software project related to this work, *planetp4v2*, can be built and executed.

4.5.1 Requirements

The project was developed and tested using Linux. Although the required tools and libraries are also available for Windows and Mac OS X, it is recommended to use Linux as this guide does not cover the other operating systems.

Irrlicht

`http://irrlicht.sourceforge.net/`

The project depends on Irrlicht version 1.5. The Irrlicht SDK is available for Windows and Linux and as a separate Mac OS X version.

[2] `http://www.sqlite.org/`
[3] `http://www.dvs.tu-darmstadt.de/research/cusp/`

Irrlicht for Linux requires a few additional libraries, which should already be installed in a typical desktop Linux installation. These include the X server, OpenGL, libpng, and libjpeg. Refer to the Irrlicht documentation for more information.

CUSP

`http://www.dvs.tu-darmstadt.de/research/cusp/`

The CUSP website provides pre-compiled CUSP libraries for both Linux and Windows on x86 platforms (32 and 64 bit). Alternatively, CUSP may be built from the source code from the DVS SVN.

The CUSP library depends on the GMP library[4] which may typically be installed via the package manager. For building the library you also need the MLton[5] Standard ML compiler and a GNU make[6] environment including a C++ compiler.

Boost

`http://www.boost.org/`

Parts of the *planetp4v2* project depend on the Boost library, particularly on the smart pointer infrastructure[7].

CMake

`http://www.cmake.org/`

The *planetp4v2* project is a CMake based project. CMake does not build the the software directly but allows creating UNIX makefiles or project files for Microsoft Visual Studio or KDevelop. See section 4.5.2 and the CMake documentation for more details.

4.5.2 Building

This section briefly describes how to build the components for this project. We assume a Linux environment fulfilling the requirements from 4.5.1. There are three parts of which each depends on the previous one. The first is about the CUSP library, followed by the main game project. The last part deals with the game embedded in the CUSP simulator.

CUSP

Building CUSP from source is optional. The C++ bindings of the CUSP library are located in the subfolder *cusp/cbindings/* of the BubbleStorm project. They are built simply calling *make*:

```
cd <project>/cusp/cbindings
make
```

[4] `http://gmplib.org/`
[5] `http://mlton.org/`
[6] `http://www.gnu.org/software/make/`
[7] `http://www.boost.org/doc/libs/1_40_0/libs/smart_ptr/smart_ptr.htm`

Output is the archive file *libcusp.a*. The corresponding C header file is generated as *libcusp.h*. As the C API is inconvenient to use, C++ wrapper classes provide an API that is similar to the original Standard ML API. Thus, applications should always use the C++ header file *cusp.h*.

planetp4v2

planetp4v2 is a CMake project. Using the *cmake* command you can generate either Unix makefiles or set up a project for a common IDE like Microsoft Visual Studio, CodeBlocks, Eclipse CDT, or KDevelop. For just building the project, it is recommended to create a makefile and execute it:

```
cd planetp4v2
cmake .
make
```

If successful, this will generate the executable *planetp4* and the archive for the simulator *libplanetp4sim.a*.

Depending on the location of the BubbleStorm sources, the CMake project description file, *CMakeLists.txt*, has to be adapted. The line

```
SET(TRANSPORT_DIR <...>)
```

specifies the (absolute or relative) path to the *cbindings* directory, e.g.

```
SET(TRANSPORT_DIR ../../BubbleStorm/Middleware/cusp/cbindings)
```

Simulator

The Simulator build uses the *libplanetp4sim.a* and links it against the simulator core and setup code. For building the simulator there is a makefile, as usual.

```
cd planetp4v2/simulator
make
```

Similarly to the CMake project, the makefile contains the variable *TRANSPORT_DIR* specifying the location of the *cbindings*, e.g.

```
TRANSPORT_DIR=../../../BubbleStorm/Middleware/cusp/cbindings
```

4.5.3 Command Line Parameters

The *planetp4* executable can run in client, server, and peer-to-peer mode as selected by command line switches. All available command line options are listed in the following table:

Option	Description	Default
`-m, --mode`	Selects the networking mode. Available modes are `p2p`, `cuspserver`, `cuspclient`, `tcpserver`, and `tcpclient`.	`p2p`
`-b, --bind-port`	Local UDP bind port number.	`8585`
`-s, --server`	Server or gateway peer address.	<none>
`-p, --server-port`	Server or gateway peer port number.	`8585`
`-c, --console`	Disables 3D rendering and runs the game in console mode.	<off>
`-a, --ai`	Enables AI player control (bot mode).	<off>
`-t, --terminate`	Terminates the game after the given number of milliseconds.	<none>

Examples

Running the program without any parameters

```
planetp4
```

activates peer-to-peer mode but does not connect to a network. To connect to an existing peer-to-peer network, run

```
planetp4 -s <server>
```

To start a second instance on the local machine, it is necessary to specify an alternate bind port:

```
planetp4 -b 8586 -s localhost
```

Alternatively, you can connect a bot to the network with

```
planetp4 -b 8586 -s localhost -c -a
```

For client-server mode, it is necessary to start a dedicated server instance first:

```
planetp4 -m cuspserver
```

A client is connected using the following command line:

```
planetp4 -m cuspclient -s <server>
```

On localhost, use

```
planetp4 -m cuspclient -b 8586 -s localhost
```

4.6 CUSP Bindings

Part of this work is to create C++ bindings to the Standard ML CUSP API. This section discusses the key points of that task.

4.6.1 Concept

MLton's foreign function interface[8] (FFI) allows calling C functions from Standard ML and exporting Standard ML functions to be called from C code. This is the basis for the C++ bindings. The file *libcusp.sml* contains Standard ML glue code that exports C functions. From that file the MLton compiler generates a C header file (*libcusp.h*) which declares the exported functions. The header file is included by the C++ wrapper code in *cusp.c*, which is compiled to an object file, which in turn MLton finally adds to the archive containing the whole transport protocol implementation, `libcusp.a`. To use the library, applications just have to include the C++ header file *cusp.h* and link *libcusp.a* to the the program. Figures 4.8 and 4.9 depict the concept and the build process.

[8] `http://mlton.org/ForeignFunctionInterface`

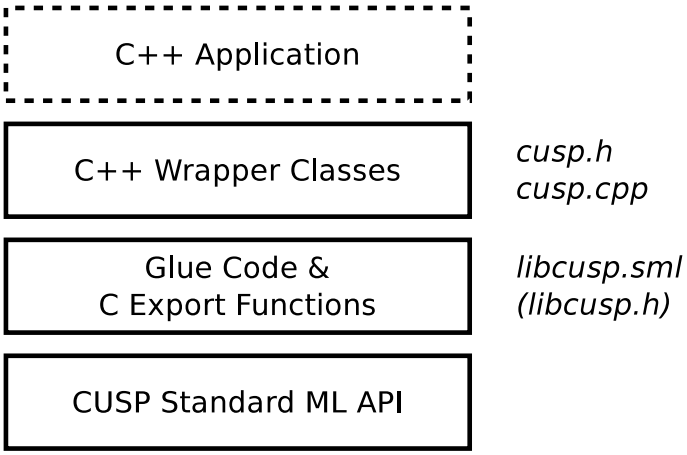

Figure 4.8: CUSP bindings concept and important source files

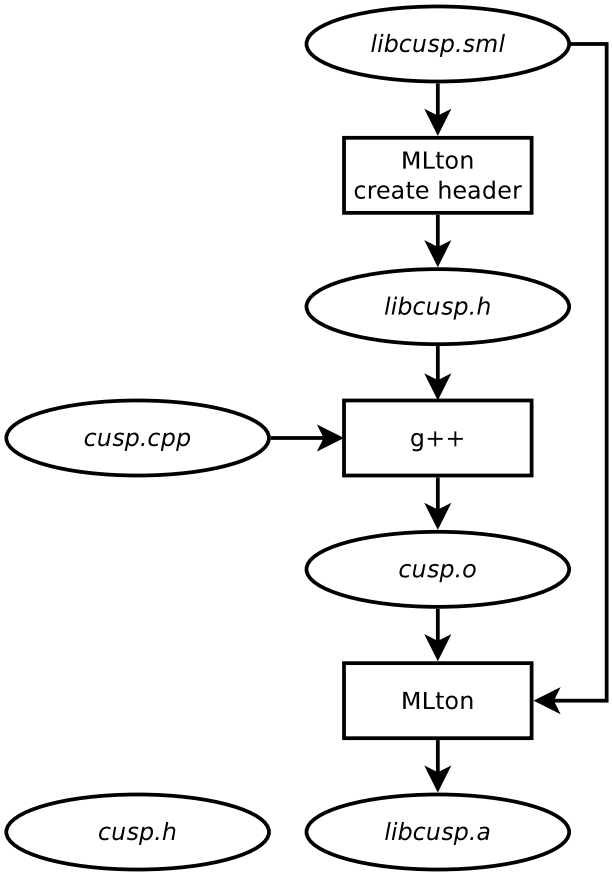

Figure 4.9: CUSP bindings build chain

4.6.2 Implementation Concerns

Unlike C++, Standard ML is not object oriented and thus does not have the concept of classes. But the CUSP API with its signatures and implementing structures for endpoints, hosts, streams, etc. can be mapped directly to an object oriented API providing a corresponding class for each signature. The main challenge when creating C++ bindings for Standard ML is how to cleanly map between the garbage collected Standard ML environment and C++. The solution developed for this purpose is generally applicable for mapping Standard ML structures to C++ objects.

The C interface exported by the Standard ML glue code makes the structures accessible via *handles*. Whenever a structure is to be passed to the C part, it is internally stored in an `IdBucket` which returns the index of the newly inserted element. That index is passed to C as a handle. Accordingly, any function that takes a structure as a parameter is exported to C expecting a handle. When called, the structure is retrieved from the corresponding `IdBucket` and the wrapped function is invoked. By keeping the structures in the `IdBucket`, they remain referenced within Standard ML and thus are prevented from being garbage collected while still being needed by the C part. Consequently, it is the C part's responsibility to free all structures when they are not used anymore. All three operations of `IdBucket`, storing, retrieving, and removing structures, have $O(1)$ complexity. Structure retrieval is based on an array lookup by index, while storage and removal of structures profits from keeping a linked list of free cells.

The object oriented C++ interface in turn wraps the exported C functions. Although the C functions can be used directly by applications, that is not recommended. Especially because of the need for explicitly freeing any handle ever obtained, using the C interface is both inconvenient and error-prone. The C++ API disburdens the programmer from most of those tasks. Each C++ API class basically just keeps a handle to the corresponding structure in Standard ML. Creating, copying and deleting classes can be done in the usual C++ way and the whole handle management is safely wrapped. The resulting C++ API classes match the Standard ML signatures (for an overview see 2.5.2) with only a few exceptions.

But still there is one important difference significantly affecting the usage of the C++ API, which is the lack of closures and/or lambda calculus in C++. As the CUSP API is completely asynchronous, there is often the need for passing callbacks. Standard ML provides closures, allowing to pass functions that keep their surrounding scope, greatly simplifying the programming and improving code readability. In C/C++ it is in most cases necessary to explicitly define and create an object that keeps the context the the calling function and that is passed with the callback function. This limitation of C/C++ cannot effectively be overcome. The solution chosen here is to provide interfaces (i.e., purely virtual classes) as callback types, which is the typical way of implementing callbacks in Java[9]. For instance, `EndPoint` defines a handler interface for the `contactPeer()` method:

```
class EndPoint
{
    (...)

    class ContactHandler {
        public:
            virtual void onContact(Host& host) = 0;
            virtual void onContactFail() = 0;
    };
    void contactPeer(const Address& addr, ContactHandler* handler);

    (...)
}
```

[9] http://www.javaworld.com/javaworld/javatips/jw-javatip10.html

Applications implement this interface and pass a reference to an instance of the implementing class when calling `contactPeer()`. The implemented class may carry any context information that is necessary for the application. Depending on the outcome of the contact operation, either `onContact()` or `onContactFail()` are invoked.

4.6.3 Convenience Helper Classes

CUSP only provides unidirectional streams. So applications requiring a TCP-like bidirectional connection have to set up one stream for each direction. The typical approach is the following. The client (i.e., the node that initialates the connection) creates an outgoing stream to the server (which is listening at a commonly known service). Additionally, the client starts listening and sends the corresponding service ID to the server via the stream. The server receives the service ID and creates a stream to the client. Since this is a common pattern, the helper class `BiDiConn` implements the client side functionality, providing an easy-to-use bidirectional connection interface.

Another common pattern is the single request-response cycle as typically appearing with an RPC call. For that purpose the two classes `SingleCycleConn` (client side) and `SingleCycleServer` (server side) were implemented. `SingleCycleConn` basically just provides the request functionality that takes the request message as a string and returns the server's response as a string. There are a few variants of that functionality for both synchronous (blocking) and asynchronous invocation. `SingleCycleServer` expects an application callback that handles incoming requests and returns the response message (both passed as strings).

Two additional helper classes are `BufferedOutStream` and `AbortableContactHandler`. The former is similar to a normal `OutStream` but allows to write data in any state. It buffers all written data until the encapsulated `OutStream` is ready to write. The downside of using `BufferedOutStream` is that the application does not have any information about how much of the data is already written. `AbortableContactHandler` performs an `EndPoint::contact()` call that is abortable, i.e., after calling its `abort()` method, the handler's `onContact()` will not be invoked anymore.

4.6.4 Discussion

The mapping of the Standard ML's API signatures/structures to C++ classes using `IdBucket`, handles, and C++ wrapper classes is surprisingly effective and problem-free after the basic patterns have become clear. Still, a lot of glue code is necessary for each new structure-to-class mapping, but always applying the same pattern, most of the work can be done by copying existing code.

In contrast to that, the translation of the callback mechanisms has not shown to be satisfactory. The Java-like way of passing callbacks suffers from the lack of anonymous classes in C++. Thus keeping track of the context still needs much extra code. A better alternative solution may be to use a C++ signal framework like libsigc++[10] or Boost signals[11]. These frameworks at least offer more flexibility to the application developer, not requiring to implement a new class for each callback use. But still, these do not offer the convenience of closures.

The development of the CUSP networking code for Planet π4 has shown that the development for such a heavily callback-driven API quickly becomes tricky as it is likely to produce bugs that are hard to discover. A main reason is that callbacks that are passed to a function are usually called later in the event loop (e.g., after data as been received), but the callback may also be called immediately from within the function, if certain conditions apply. One example is the `write()` method of `OutStream`. While in normal operation its callback is invoked later when all (or most of) the data has been transmitted, the

[10] `http://libsigc.sourceforge.net/`
[11] `http://www.boost.org/doc/libs/1_40_0/doc/html/signals.html`

reset callback may be called synchronously. An example case illustrating the problem can be found in section 5.3.

The C/C++ bindings of the CUSP API rely on MLton's foreign function interface which is not supported by other Standard ML compilers. Hence it cannot be built with any other compiler, generally limiting the portability. But since the current implementation of CUSP depends on a few features that are exclusively provided by MLton, it is non-portable anyway, thus there would be no effective benefit portable bindings.

5 Evaluation

The evaluation part starts with game protocol measurements comparing the client-server to the peer-to-peer network implementation. Section two evaluates the performance of the game implementation concerning computational and memory requirements, with the focus on simulation scalability. A qualitative evaluation on the usability of CUSP for C++ applications finalizes this chapter.

5.1 Networking Measurements

The game engine can be run on both real and simulated networks without modification. The current implementation enables easy switching between a real and a simulated network for the networking modules using CUSP. Though, for a detailed analysis on the quality of the different network implementations, an important tool is still missing: the (distributed) collection of protocol quality data. Protocol quality data includes information about the game world view consistency between different players and change propagation delays. More about this topic can be found in the future work section (6.1). The following measurements are thus limited to the locally measurable traffic.

This section presents the network traffic measurements for client-server and peer-to-peer mode. Since two client-server versions (TCP and CUSP) have the same messaging model, only the CUSP version is measured as the representative client-server implementation for measurements. The scenario for the peer-to-peer measurements is run in the CUSP simulator on a single machine. The client-server version does not run in the simulator (because it partially uses blocking network operations), so the scenario uses discrete instances connected over loopback on a powerful machine (16 processors, 64GB RAM). In both cases the available network bandwidth is practically unlimited and the latency is negligible, thus it is assumed that they do not influence the measurements.

In each run, there is a constant number of bots, all joining initially (in the first second of the game) and staying until the end of the simulation. The number of players is varied between runs. Because of the simple bot behavior, all players remain within each other's vision range. Especially with a growing number of players remaining in a limited space, it can be observed that all bots are firing almost continuously as there is almost always a target nearby. Concerning the generated load, the scenario can be seen as a worst-case scenario. The traffic numbers are averaged over 30 seconds gameplay and three clients/peers. In a second measurement round players join continuously every two seconds and the traffic of one single node is measured over time.

5.1.1 Client-Server

Due to design limitations, the server of the client-server version does not reliably support more than around 35 concurrent players on the machine used for the measurements (4x4 core AMD Opteron 8356, 2.3GHz, 64GB RAM; but the single-threaded server only uses one core). This is partly because the client-server implementation misses some common optimizations like compact (binary) messages, message aggregation, or multithreading. So interpretation of the measurements should not be focused on absolute values, but on trends. And the peer-to-peer version also lacks some optimization, that's why this comparison seems acceptable.

Table 5.1 and figures 5.1 and 5.2 show the generated traffic depending on the number of players of client and server respectively. The clients' outgoing bandwidth is almost constant because the clients send their updates only to the server. Slight variations come from their irregular activities, particularly

Table 5.1: Client-server traffic depending on the total number of players

# Players	Client Total (B/s)	Up (B/s)	Down (B/s)	Server Total (B/s)	Up (B/s)	Down (B/s)
2	7,511	4,514	2,997	14,933	5,955	8,978
3	5,513	3,510	2,002	16,799	6,072	10,727
4	6,518	4,649	1,869	26,030	7,440	18,591
6	12,080	9,161	2,919	71,592	15,758	55,834
8	16,805	13,328	3,477	131,114	24,210	106,903
12	22,405	19,191	3,213	269,093	38,355	230,738
16	26,834	23,594	3,240	427,596	53,215	374,381
24	47,501	43,044	4,457	1,139,158	112,402	1,026,756
32	63,338	58,788	4,551	2,012,340	142,783	1,869,557

shooting. The incoming bandwidth grows linearly with the number of players because each client received the updates of all others. The server's traffic measurements show why the number of players is limited to less than 40. With 32 players, the average outgoing traffic already reaches about 20MB/s, and, more importantly, its growth is quadratic. The time-traffic plots 5.3 and 5.4 show the same picture. Despite any potential optimization, the server's work always increases quadratically with the number of players (of which everyone can see all others) that it has to serve. Message aggregation will help to significantly reduce the outgoing traffic, but it will not change the complexity. A possible countermeasure is to reduce the level of detail of update messages depending on the load, e.g., by decreasing the frequency of outgoing update messages.

5.1.2 Peer-to-Peer

The peer-to-peer implementation using the slightly adapted pSense algorithm, which is run in the simulator, supports much larger numbers of players than the simple client-server version. For limiting simulation time and memory requirements, the measurements are set to a maximum of 64 and 128 players respectively for the two measurements.

The peer traffic (table 5.2, figures 5.5 and 5.6) shows a quadratic growth with the number of players in the vision range, equally for both upstream and downstream. This behavior evidently results from the fact that in the simple scenario each player sends its updates to everyone else and also receives updates from everyone. Because of the different message formats, the peer-to-peer traffic cannot be compared directly with the client-server version. The binary peer-to-peer position update messages have a fixed size of 64 bytes while client-server messages are variable in size, with an average position update message size of around 50 bytes. With 32 players, the resulting traffic in each direction is around 100KB/s which can be seen as an upper bound for upstream bandwidth of today's internet connections. A client in the client-server version requires far less bandwidth. But as for the client-server version, there is still a great potential of optimization for traffic reduction. The pSense algorithm describes a forwarding technique to disburden weak nodes which has not been implemented, and Donnybrook introduces different levels of update accuracy.

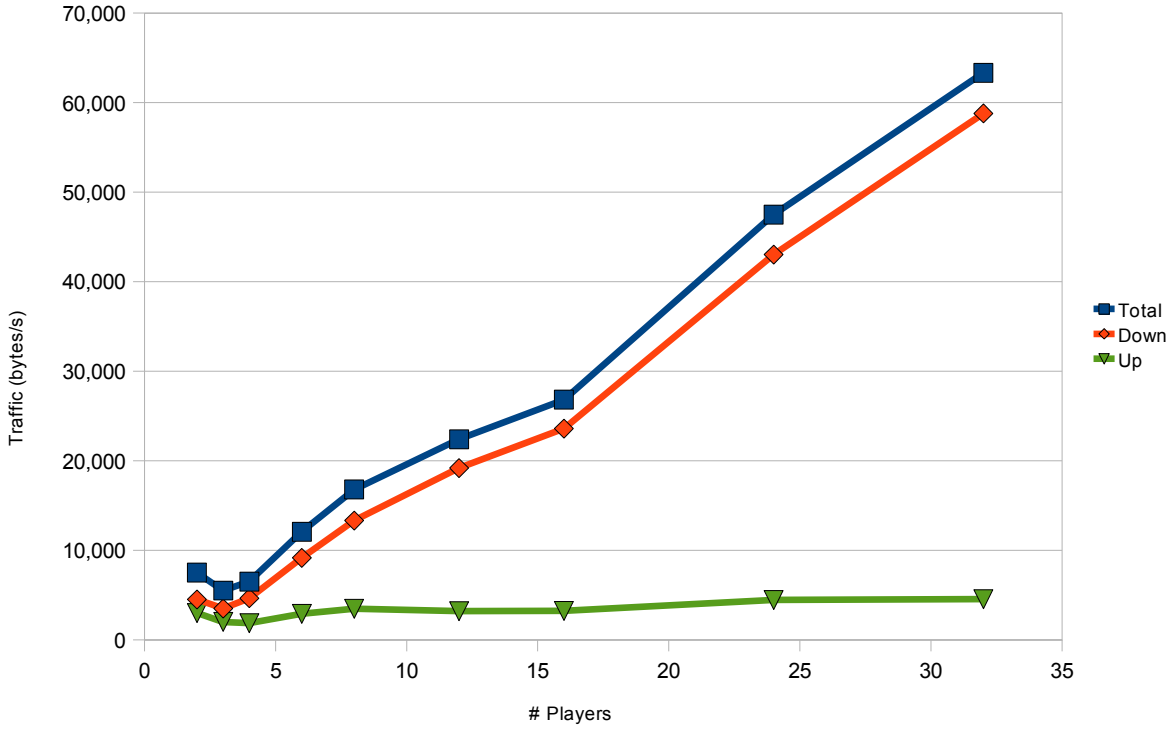

Figure 5.1: Client traffic depending on the total number of players

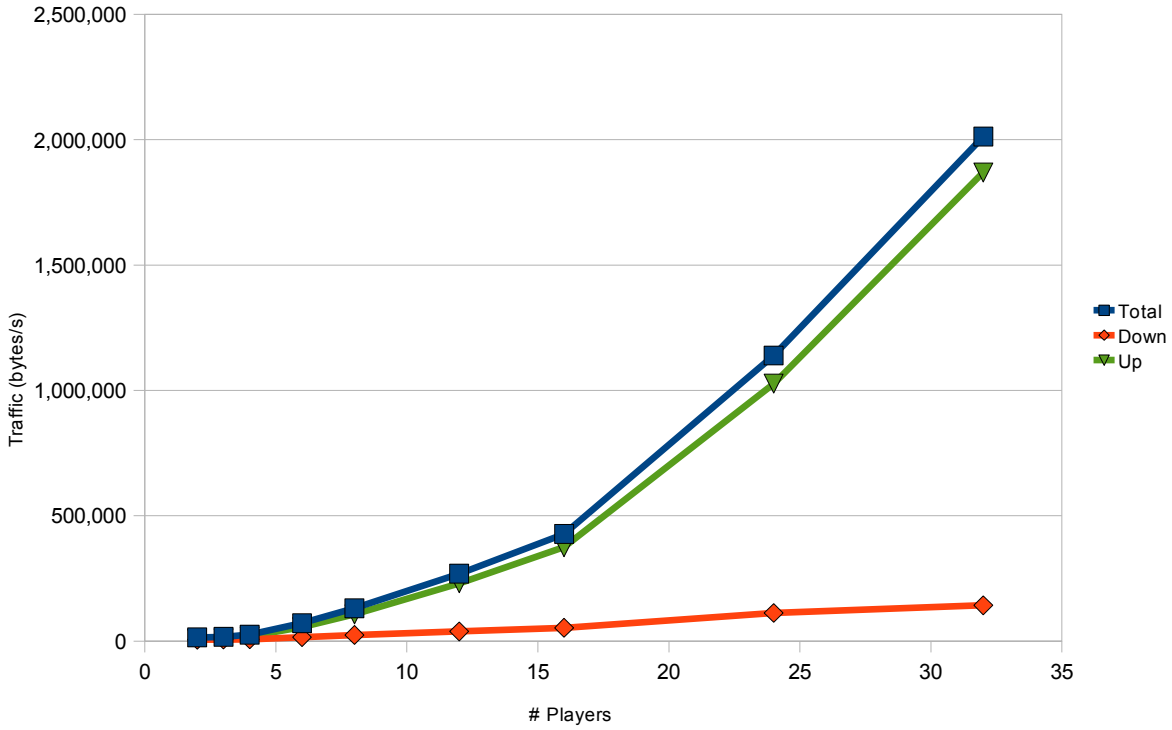

Figure 5.2: Server traffic depending on the total number of players

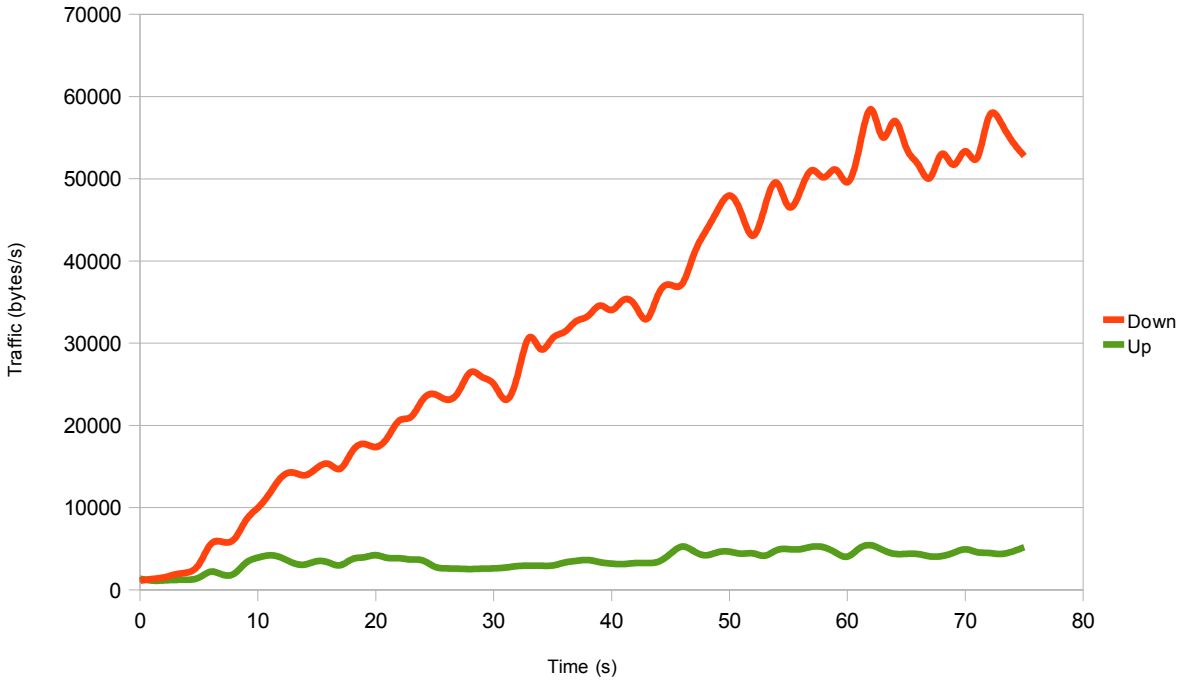

Figure 5.3: Traffic of one client with new clients joining every 2 seconds, up to a total of 32 clients at 64 seconds

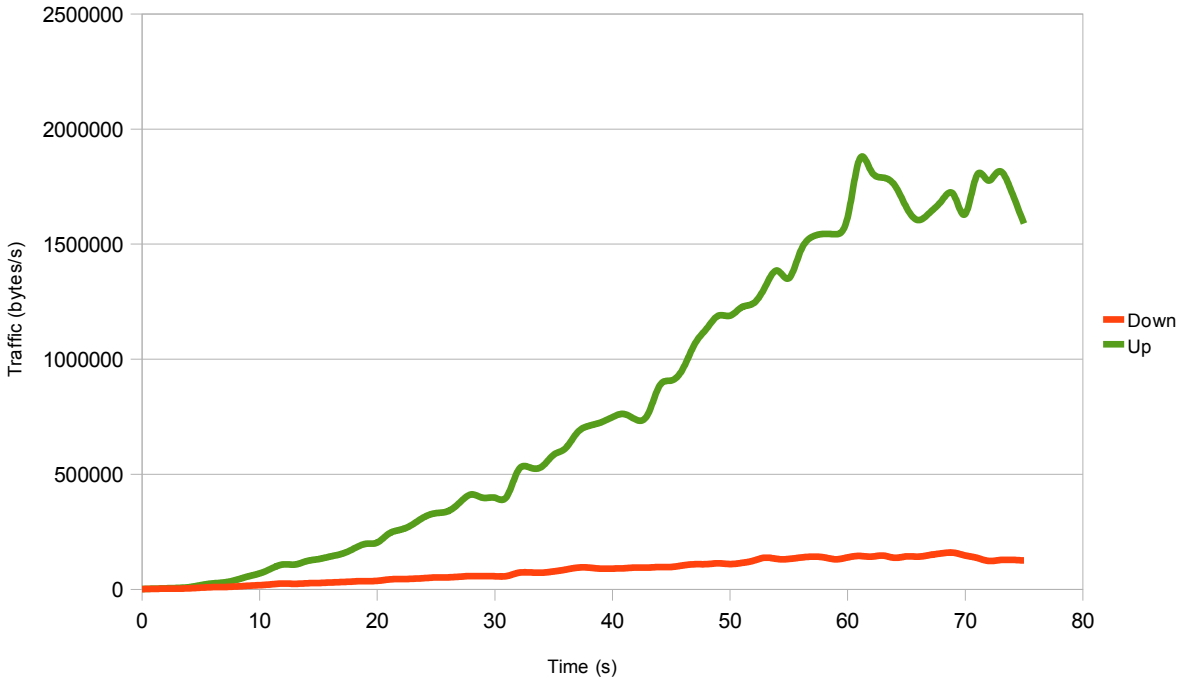

Figure 5.4: Server traffic with new clients joining every 2 seconds, up to a total of 32 clients at 64 seconds

Table 5.2: Peer traffic depending on the total number of players

# Players	Total (B/s)	Up (B/s)	Down (B/s)
2	2,841	1,422	1,420
3	6,260	3,131	3,129
4	10,434	5,286	5,148
6	19,302	9,621	9,682
8	28,295	14,121	14,174
12	48,109	23,996	24,113
16	70,471	35,977	34,494
24	123,948	61,805	62,143
32	191,991	95,888	96,103
48	353,216	176,762	176,454
64	554,774	277,124	277,650

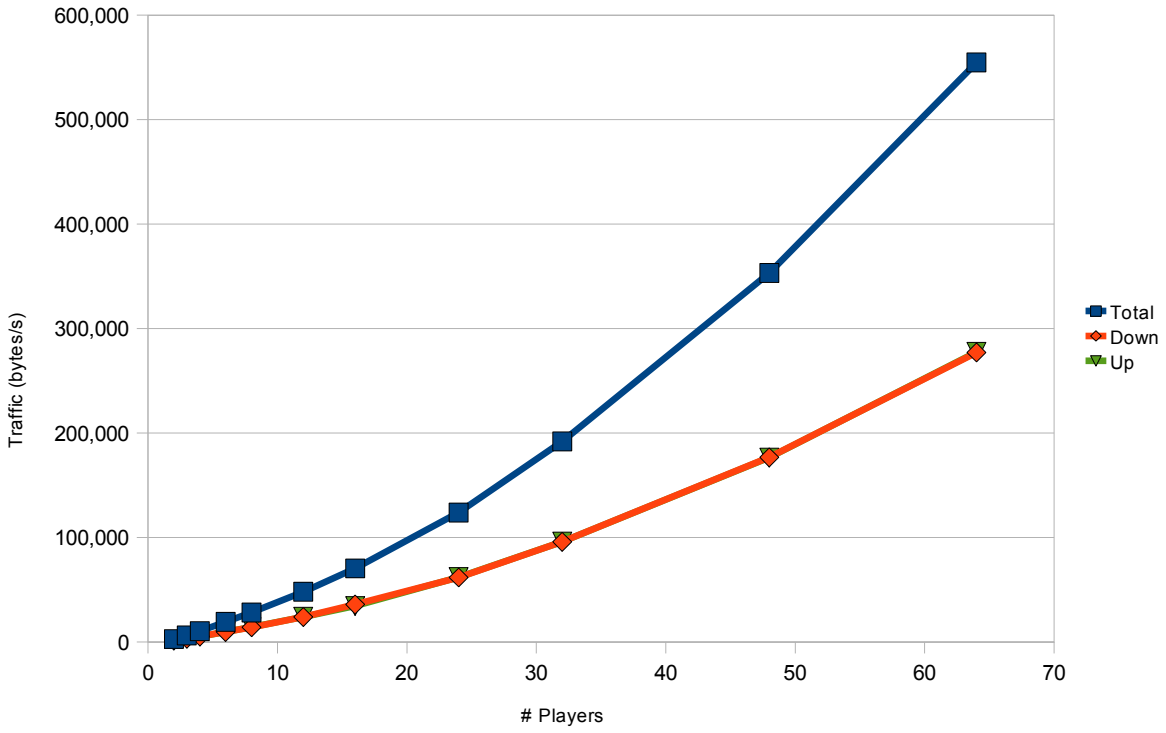

Figure 5.5: Peer traffic depending on the total number of players

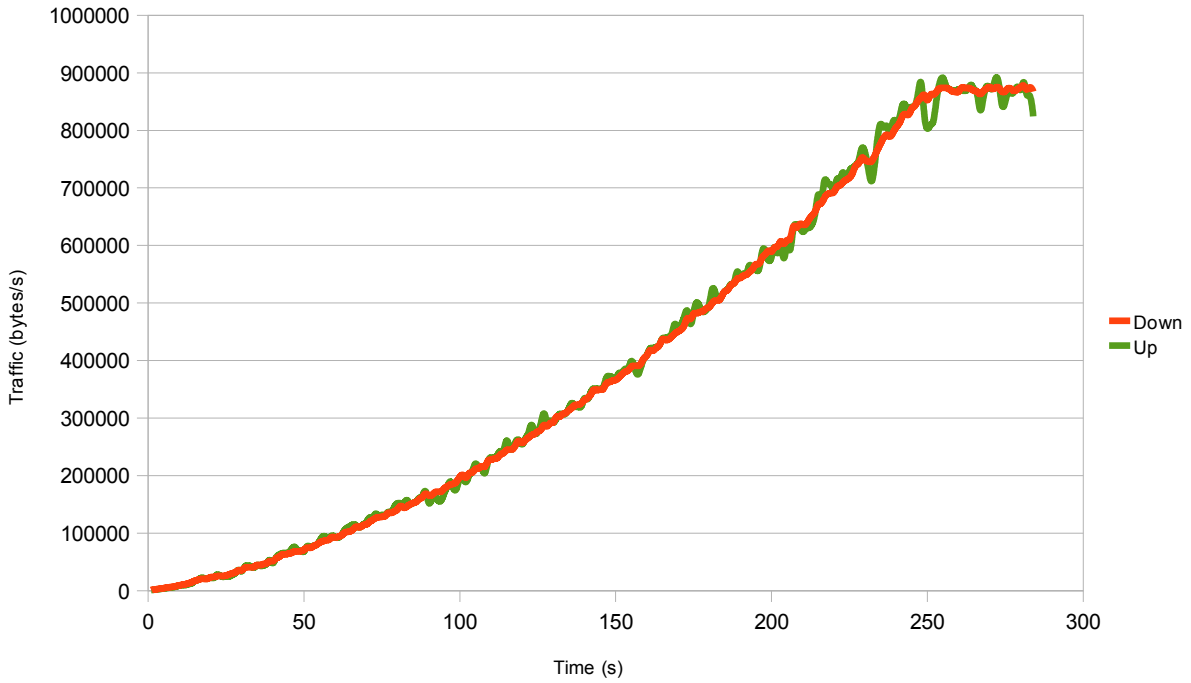

Figure 5.6: Traffic of one peer with new peers joining every 2 seconds, up to a total of 128 peers at 256 seconds

Clustering

For substantiating the claim that the quadratic behavior originates from the fact that all players are within each other' vision range, additional experiments are conducted with multiple clusters of players. The clusters' distance is set to double the vision radius to make sure that players from neighboring clusters are out of sight. So the connections between players are only within the clusters except for the sensor node connections. Figure 5.7 shows the results. The left plot shows the traffic with a fixed number of 32 players and a varying number of clusters into which the players are partitioned. The four data points represent the combinations 1 cluster of 32 players, 2 clusters of 16 players each, 4 clusters of 8 players, and 8 clusters of 4 players. The right plot keeps the number of players per cluster constant (size: 8), resulting in 1 to 4 clusters for 8 to 64 total players.

The two plots visualize that the most important factor affecting the traffic in the peer-to-peer network is the number of neighbors within the vision ranges. The left plot shows the decrease in traffic with increasing cluster count and thus decreasing cluster size. If the cluster size stays constant, the peer traffic is in fact not constant, as shown by the right plot. An increasing number of clusters still slightly increases the traffic. This can be scribed to the increasing sensor node communication, which is most notably when comparing 8 players (one cluster) and 16 players (two clusters). With only one clusters, there are no sensor nodes at all; with more clusters, an increasing proportion of the eight sensor nodes of each node is filled, accounting for the additional traffic.

Qualitative Analysis of the Algorithm

As the bot-only scenarios only stress the message exchange capabilities within vision range, additional qualitative tests were run to check the sensor node functionality. These have shown that nodes moving

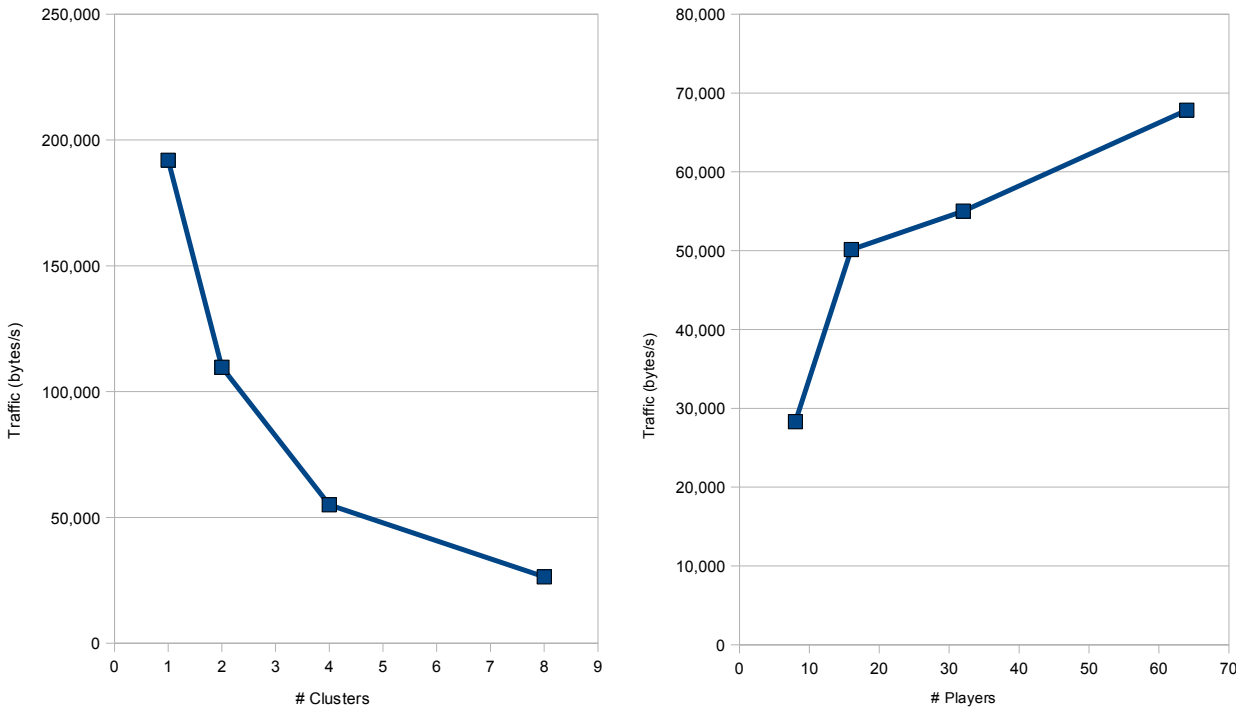

Figure 5.7: Total peer traffic depending on number of clusters (left) and depending on the number of players with constant cluster size (right)

outside each other's vision range and not being sensor nodes disconnect as expected. Through the sensor nodes, nodes quickly re-connect after re-entering each other's vision range. Only when sensor nodes crash (which is the same as a normal disconnect in the current version), the network partitions in some cases. This is a problem especially in low-density regions where only a few players are available as sensor nodes. One solution could be the integration with a conventional peer-to-peer overlay in which each peer regularly stores its position. When a peer detects the crash of one of its sensor nodes and it does not know an alternative, it queries for peers in the particular area to find a new sensor node.

5.1.3 Discussion

The measurements show that the peer-to-peer version easily supports a larger amount of players than the client-server version. Both the client-server and the peer-to-peer version are simplistic and unoptimized, so the results have to be taken with a pinch of salt. Current client-server implementations show that optimization may elicit much larger numbers of players than the implementation evaluated here. In client-server systems, the server is always the bottleneck, in this case especially concerning bandwidth. That means the server may be replaced by a more powerful machine and its connection bandwidth can be extended when necessary; both is a matter of resources. But the fact that the similarly unoptimized peer-to-peer version is already able to cope with a magnitude of order more players, shows the potential of peer-to-peer systems.

Still, the peer-to-peer version has high bandwidth requirements compared to today's consumer internet connections. Especially the upstream bandwidth is the limiting factor using typical asymmetric connections like ADSL. While the clients in the client-server version have low bandwidth requirements, particularly upstream, the peers' traffic grows linearly with the number of neighbors.

Table 5.3: Computation time and memory requirements for a simulation (peer-to-peer mode)

# Players	Memory (MB)	Runtime 30s (s)	Time dilatation
2	175	3.4	8.738
3	185	4.8	6.250
4	196	6.4	4.688
6	216	9.9	3.030
8	235	14.7	2.036
12	276	29.1	1.031
16	324	50.1	0.599
24	429	118.4	0.253
32	558	224.8	0.133
48	865	620.7	0.048
64	1,250	1,302.6	0.023
96	2,352	3,931.5	0.008
128	3,822	8,113.4	0.004

The measurements presented here are only a starting point and for gaining first impressions. More extensive tests require further work concerning

1. improved gameplay modes and more realistic player behavior,
2. further optimized networking systems that are really trying to get the most out of limited bandwidth, and
3. tools that allow for a deeper and more detailed analysis of the game events and generated traffic.

These three items should also allow for measurements of protocol quality, i.e., how well the network protocols perform with delays and limited bandwidth.

5.2 Performance Measurements

Especially for the simulation of large systems, the simulation performance and memory requirements are of major concern. This section discusses measurements of runtime and memory consumption of a peer-to-peer network simulation with a varying number of players. The scenario is the same as described in 5.1. The simulation run time is measured for a 30 second simulated game. One bot is equipped with a rendered viewport. To save memory, no textures are loaded except for the viewport instance. The ship's textures are completely omitted even in the viewport instance.

All performance measurements are run under a 64-bit Linux (2.6.30, x86_64) on a system with an AMD Athlon 64 X2 4600+ at 2,4GHz and 6GB RAM. Since the simulator is single-threaded, only one core of the CPU is used. As a comparison, a single-player standalone instance (with all textures loaded) of the game requires around 210MB of RAM. The Standard ML part of the program is garbage collected, thus the memory requirements vary over time. But the measurements have shown that these variations are less than 10%. Measured run times include loading of resources, setting up the game and connecting to the network for each instance. The simulation time dilatation is calculated by dividing 30 seconds (simulated time) divided by the simulation run time (wall time). Thus it is the factor indicating how much faster (or slower) the simulation runs relative to real time.

Table 5.3 and figures 5.8 and 5.9 show the results. Both memory consumption and computation time show about a quadratic growth. Reason for that is the above mentioned clustering of players in a

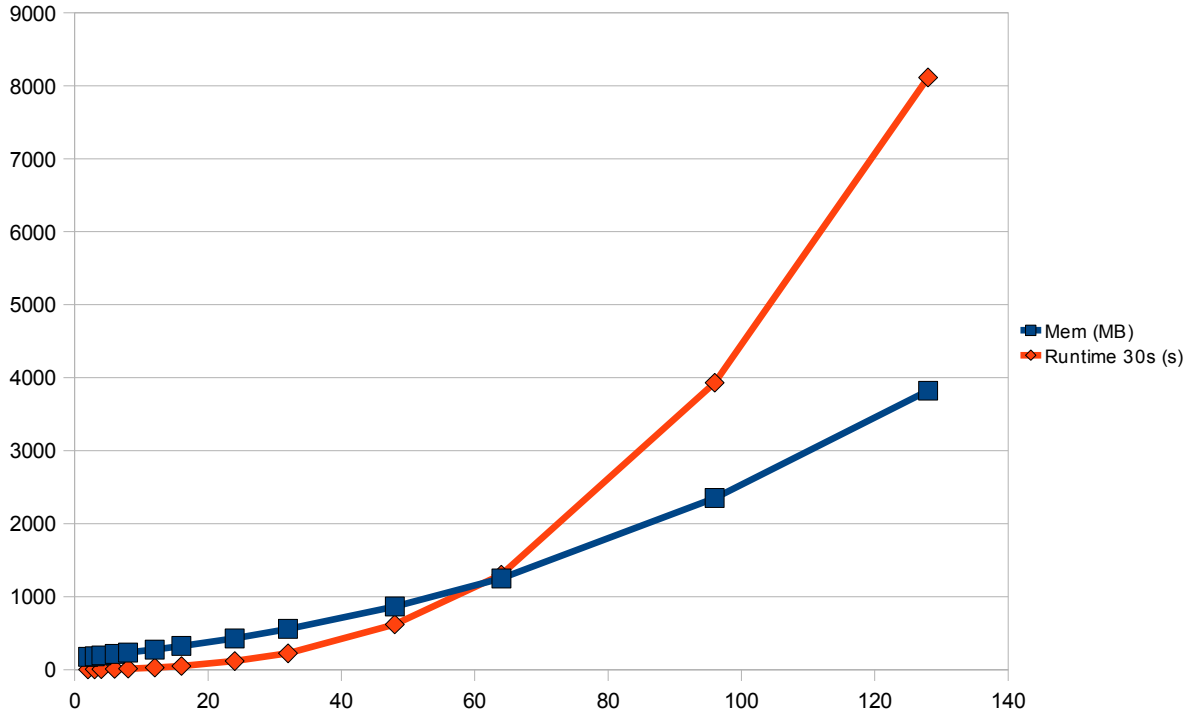

Figure 5.8: Computation time and memory requirements for a simulation depending on the number of players

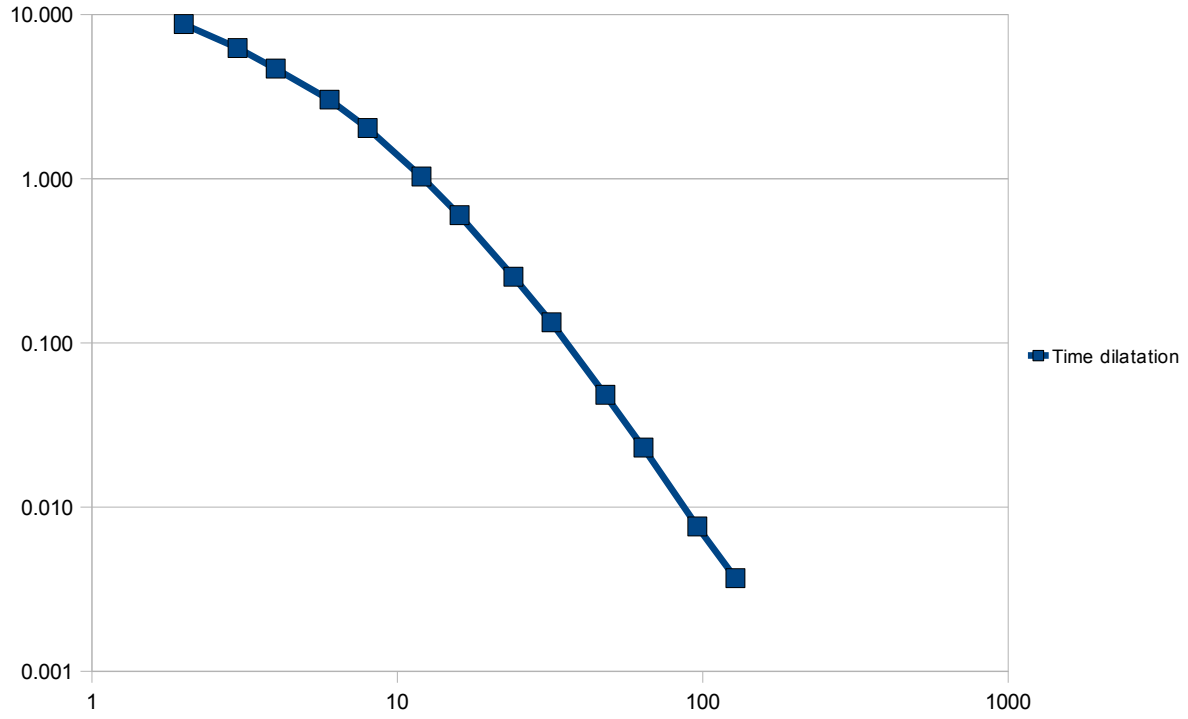

Figure 5.9: Simulation time dilatation depending on the number of players

Table 5.4: Computation time and memory requirements of a 32-player peer-to-peer game simulation with player clustering

# Clusters	Players/cluster	Runtime 30s (s)	Memory (MB)
1	32	225	558
2	16	133	510
4	8	99	478
8	4	57	457

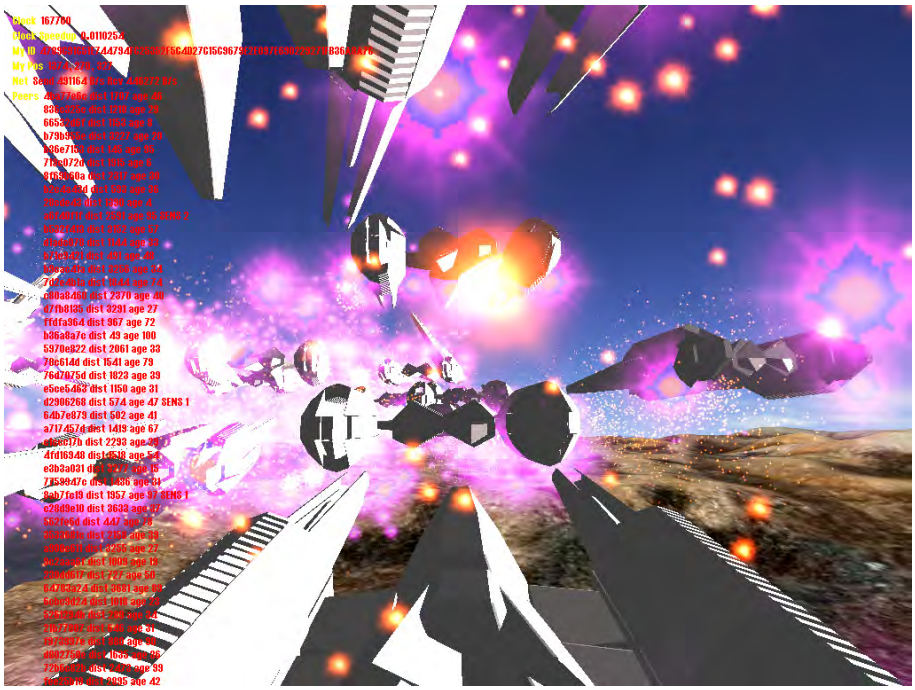

Figure 5.10: Screenshot of a simulation run with a high player density (spaceship textures disabled)

small space. As each player can see all others, everyone needs to process information about everyone else, resulting in the quadratic consumption. This is the same effect that also appears in the traffic measurements. And like there, it has to be mentioned that the growth will be much lower if the average number of nodes in the vision range stays constant. Similarly to the peer traffic, the simulation time is reduced if the players are separated into multiple clusters. Table 5.4 shows the significant impact of the player distribution on the simulation.

The simulation measurement are made with transport encryption turned on, thus all data sent over the simulated network is AES-encrypted and each connection attempt requires complex public key encryption operations. Encryption in the simulator is obviously superfluous. A later version of CUSP allows to easily turn off encryption through the API, resulting in a speedup of the simulation. Also, the simulator tested here is a 64 bit build which did not yet have an optimized cryptographic suite. The now available optimized version is much faster. Still, all these optimizations will not change the trends shown in the measurements above.

5.3 CUSP

This section briefly discusses the experience of using CUSP. First off, the reliability and correctness of the protocol implementation itself has been very satisfying from the beginning. There were only a few minor bugs to fix that appeared in use cases that had not been tested before, but this is negligible.

However, the correct usage of CUSP's asynchronous API with C++ has shown to be challenging. Obviously, the API is more complex than the Berkeley sockets API. Asynchronous I/O using callback functions is generally inconvenient in C++. The callback handlers passed to the I/O function usually require some state for futher processing of the results. In C++ there is no other way of passing that state than explicitly creating an object and passing it to the function. Languages supporting closures (like Standard ML, for instance) allow executing the callback functions in the context of the calling function, not requiring the explicit state encoding. C++ programs using CUSP are thus significantly longer than a comparable Standard ML program or a C++ program only using synchronous I/O.

Besides the inconveniences, the asynchronous I/O has certain pitfalls that may cause bugs resulting in sporadic crashes of the program. Callback functions that are passed to an I/O function may either be invoked later from the main event loop (e.g., when data has been received) or they may be called directly from the I/O function (e.g., when a connection has been reset and it is clear that no more data can be received). While debugging the game, one case appeared very typical for that problem: To multicast a message to list list of known players, that list is iterated and the corresponding send function is called in each iteration. If one of the connections has been reset from the remote end, the send function may immediately call the reset handler, which in case of the buggy implementation removed the corresponding player from the list of known players. As that list is being iterated, it is not allowed to simply remove elements from the list as the loop may then overrun the end of the list, eventually causing a segmentation fault in most cases. Such problem can only be solved by explicitly taking care of the situation, e.g., by creating a copy of the list before iterating.

Still, the asynchronous API seems to be a good choice. CUSP is designed for holding a large number of connections which is inefficient using synchronous I/O. A general pattern avoiding many problems caused by preemptive callbacks is to always call the asynchronous I/O function at the and of the current function, preventing side effects with code after the I/O function call. Concluding, CUSP is a reliably working protocol with an API that initially requires some time and experience of the programmer to become aware of all consequences, but supports efficient networking applications.

6 Conclusion

The core task of this work was the development of a simple client-server and, more importantly, a peer-to-peer network engine for the game Planet π4. Surrounding that topic, a set of other tasks was tackled, all of them providing support for the core networking topic.

The completely reworked network architecture of Planet π4 now allows for a simple replacement of the network engine. Two rather opposite networking approaches, a group-based client-server system and an unstructured peer-to-peer system, were both successfully applied to that architecture, proving its flexibility. The network system is fully pluggable, thus it is even possible to select/switch the network engine at runtime.

Other parts of the game were also made exchangeable. Most importantly, the main loop can be replaced, particularly allowing to run the complete game in an discrete-event simulator. This unites the two basically contradicting approaches of the busy loop that games usually run and the event based simulator's main loop that is not under control of the application which is running in the simulator. For decoupling the various game components, the internal eventing infrastructure was extended and is now used as the central communication point between components.

Since all current and future networking models have to be evaluated, there is the need for workload generation. The most obvious way of workload generation is just letting human players play the game. But that is too time-consuming in many cases, especially for large numbers of players. Thus, an AI component was developed, providing a bot player that plays the game according to simple rules. The simple bots do not generate a completely realistic workload, but the nonstop best-effort activity generates more workload in average than a human player would do, so it can be seen as a worst-case scenario.

Being the first customer of CUSP, important feedback was provided, especially in CUSP's late development and testing phase. Particularly the peer-to-peer networking implementation represents a typical use case that CUSP aims at. Also, the development and evaluation of the C++ bindings for CUSP are an important contribution, since only a small fraction of applications which may potentially use CUSP are written in Standard ML. The C++ API is thus assumingly to become more relevant for external applications than the original Standard ML API. The evaluation has shown that CUSP is a powerful and reliable protocol but introduces some challenges particularly for the C++ programmer.

Three variants of networking code were implemented and evaluated. While the two client-server versions are similar in their functionality and primarily comparing the applicability of TCP and CUSP, the peer-to-peer version introduces a new concept. The CUSP client-server and peer-to-peer implementations were evaluated concerning scalability with a focus on the traffic requirements. Although the absolute values are likely to change with further protocol optimization, the trends are definitely meaningful. Those show that the client-server version is strongly limited by the server capabilities. The generated traffic grows quadratic with the number of players. This behavior can also be observed in the peer-to-peer version, but on a lower level. And, more importantly, the peer traffic (almost) only depends on the number of players in the vision range; all players outside the vision range do not affect the peer at all except for the maximum of eight sensor nodes. Thus, the pSense implementation has shown to be in principle qualified for massively multiplayer games of any size as long as the player density is rather low.

The peer-to-peer version of the game is able to run in the CUSP simulator, allowing for a simplified and more deterministic analysis on certain aspects. The number of players is bounded by the available memory and CPU power (or time constraints). Up to 128 players were successfully simulated; 4GB of RAM, the typical memory configuration of today's desktop PCs, is enough for that. Nevertheless, it is desirable to optimize further for simulation performance in order to achieve greater player numbers and still acceptable simulation times.

6.1 Future Work

As pointed out in several places throughout this document, there is a lot of continuing work that can be done in the different topics addressed within this work. Those include several aspects of the game like gameplay modes, particularly for large numbers of players, further improvements in the networking implementations, and evaluation and simulation tools.

Currently, the game Planet $\pi 4$ has only a simple deathmatch gameplay mode. This mode is feasible for some testing and demonstration of networking models, and can be played by a simple bot implementation. But in order to generate workloads conforming to those of real games, it is necessary to develop gameplay modes that resemble those of popular online games. This is not only important for the representativeness of the workload itself but also generated opportunities for collecting experience by letting possibly large numbers of human players play the game. And that is only feasible if the game is attractive to the players.

Instead of a deathmatch with everyone fighting against each other, the players could be assigned to teams which have to compete. An example of a common gameplay mode is capture-the-flag (CTF) where each team tries to steal a flag from the base of the other team while protecting its own flag. For a larger number of players it is necessary to develop more sophisticated gameplay modes because simple modes like CTF scale only to a certain number of players/teams. The common model in role-playing games (MMORPGs) is to provide quests that small groups of players solve together, while the massive number of players forms a society (economic and social) but is not directly involved in most of the game activities of a single player. For shooter games like Planet $\pi 4$, there are no massively multiplayer gameplay modes yet, thus they still need to be developed.

For supporting attractive gameplay modes, Planet $\pi 4$ also needs to be equipped with a bunch of basic features that are still missing at the moment. One of them is a collision detection among spaceships and the terrain. The terrain needs to be expanded and enriched with features that make it feel more like a real world and that support certain gameplay modes (e.g., team bases for CTF).

A very impacting new feature would be a general-purpose object model which is required for almost every new gameplay mode. Currently, the only information that is exchanged over the network is for the players' ships and missiles. Game objects can be used e.g., for mutable terrain features (breakable parts, doors, etc.) and objects directly related to the gameplay (the flag in CTF). The whole topic of game objects filling the gap between the fixed terrain and the players' avatars, as introduced from a technical perspective in 2.4.3, is not covered yet.

For supporting enhanced gameplay modes and generating more realistic workloads in simulations, the AI implementation should to be improved. But the particular strategies that need to be implemented depend on the focused gameplay modes. Another approach for workload generation would be the recording of traces of games played by humans and replaying the games in simulation runs. But it always has to be concerned that the properties of different network engines also influence the game, feeding back to the players' behaviors. Thus it may not be feasible to just replay the same game for comparing network implementations.

Further improvements are achievable in the game' software architecture. This work made important steps towards a low-coupled and flexible software system. But there are still some remains of code that requires special handling, reducing replaceability and extensibility. One example is the `Avatar/Spaceship` construct that at some point needs to waive abstraction. Those improvements have to go hand in and with the design of new game features like those described above.

Certain changes may also be necessary in the game mechanics concerning communication. The simple IHITYOU/IDIED message scheme is unsafe against both cheating attacks and unreliable network transmission. The scheme is also very specific for the purpose of modelling shooting. If further interactions between players are introduced, it may become desirable to design a general purpose mechanism that may (reliably) handle various types of interaction.

New game features and improvements of the game mechanics do of course have an effect on the network engine requirements. Especially new features like an object model introduce completely new challenges for peer-to-peer network engines. pSense only manages the players' avatars but does not handle other types of game objects. General-purpose overlays (DHTs or unstructured systems like BubbleStorm) appear to be an appropriate base for distributed game object management, as shown by several approaches dealing with that topic (see section 2.3). These overlays may not only provide object management capabilities but may also serve as a backup infrastructure for the network maintenance in cases where the primary low-latency overlay fails.

Of course, since the goal is to compare different approaches particularly of peer-to-peer systems for games, there is task of implementing further of these approaches. Some alternatives have already been published, but with the experience gained from the Planet π4 implementation it is likely that new approaches can be developed. And even the existing implementation of pSense may be further improved, e.g., by extending the sensor node mechanism to the 3D space, possibly applying the suggested scheme based on platonic solids.

For the purpose of simulation particularly of large systems, it is desirable to optimize the game further for low resource usage in simulation mode. The omission of texture loading already saves a significant amount of memory. But still, the whole Irrlicht engine is working even if no visual output is rendered. Being an integral part of the game mechanics, the Irrlicht engine cannot be simply removed. But there may be solutions making a better tradeoff between the required infrastructure and resource consumption in simulation mode. The simulator itself should also be further improved to support more scenarios and further logging and tracing capabilities for a more detailed analysis. But that is an extra topic and not discussed here in more detail.

An important tool that should be developed for evaluation purposes is a mechanism for measuring protocol quality. Abstractly, the protocol quality is a measure for the differences in the game world states on different participating nodes. The most important source of (temporary) discrepancies of game state between nodes are the network latencies. The challenge while measuring the discrepancies in a distributed system is the fact that the communication channels used for synchronization of the measurements are affected by the same latencies. The whole situation is much easier when the system is run in a simulator; there may be simply a global component collecting all necessary information from the simulated nodes. But the simulation runs have to be validated by real-world experiments, thus a distributed mechanism for measuring protocol quality in real networks would be a highly valuable tool. Finally, distributing the game in a large scale and letting several thousands of players play over the internet, possibly gathering information about the protocol quality and player experiences, will greatly expand the knowledge about peer-to-peer massively multiplayer gaming.

A Source Code Extracts

A.1 Simulation: Game Main Code

Listing A.1: simnode.cpp

```
#include <stdio.h>

#include "anyoption.h"
#include "event.h"
#include "irrlichtdev.h"
#include "implgameinstance.h"
#include "irrlichttimer.h"
#include "bstimer.h"
#include "groupnetworkengine.h"
#include "cuspclientservernet.h"
#include "p2networkengine.h"
#include "simpleai.h"

// startNode() function export
extern "C" {
    void startNode(const char* peerAddr, const char* gatewayAddr);
}

struct MainContext
{
    IrrlichtDev* device;
    GameInstance* game;
    GameTimer* timer;
    EventManager* eventMgr;
    NetworkEngine* networkEngine;
    GameAI* ai;

    MainContext()
        : device(NULL), game(NULL), timer(NULL), eventMgr(NULL),
            networkEngine(NULL), ai(NULL) {};
};

void connectCallback(bool result, void* userData)
{
    assert(userData);
    MainContext* ctx = (MainContext*) userData;
    if (result) {
        // game
        printf("Init game...\n");
        ctx->game->init(ctx->device, ctx->timer, ctx->eventMgr);
```

```
        // AI
        printf("Init AI...\n");
        ctx->ai->init(ctx->game, ctx->timer, ctx->eventMgr);

        // init done
        printf("Game running.\n");

    } else {
        printf("Connect failed!\n");
        delete ctx->ai;
        delete ctx->networkEngine;
        delete ctx->eventMgr;
        delete ctx->timer;
        delete ctx->game;
        delete ctx->device;
        delete ctx;
    }
}

// node start function, called from SML simulator code
void startNode(const char* peerAddr, const char* gatewayAddr) {
    printf("Starting node (connect to \"%s\").\n", peerAddr);

    static bool first = true;

    // create context object
    MainContext* ctx = new MainContext();

    // create objects
    printf("Setting up...\n");
    ctx->eventMgr = new EventManager();
    ctx->device = new IrrlichtDev(ctx->eventMgr, first);
    ctx->game = new ImplGameInstance();
    ctx->timer = new BSTimer();
    ctx->networkEngine = new P2NetworkEngine(8585, peerAddr, gatewayAddr);
    ctx->ai = new SimpleAI();

    // connect to network
    printf("Connecting to the network...\n");
    ctx->networkEngine->init(ctx->timer, ctx->eventMgr, ctx->game);
    ctx->networkEngine->configureUser(ctx->device->getDevice());
    ctx->networkEngine->connectAsync(connectCallback, ctx);

    first = false;
}
```

A.2 Simulation: Game Connector Code

Listing A.2: simulator.sml

```
(* fake transport_open and transport_close for C++ bindings *)
fun nothing () = ()
val () = _export "transport_open" : (unit -> unit) -> unit; nothing
val () = _export "transport_close" : (unit -> unit) -> unit; nothing

(* call C++ node start code *)
fun start (peer : string, gateway : string option) =
    let
        val startNode = _import "startNode" : string * string -> unit;
        val () = Log.log(Log.INFO, "starting node " ^ peer)
    in
        startNode (peer, Option.getOpt (gateway, ""))
    end

(* create nodes *)
val () = GlobalLog.print ("Creating nodes.\n")
fun makeNode partner node =
    SimulatorNode.setCurrent (node, fn () => start (partner, NONE))

val bootstrapNode = Vector.sub (nodes, 0)
val bootstrapAddr = Word32.toString (valOf
    (MasterConnectionProperties.staticAddress
    (SimulatorNode.connection bootstrapNode)))

val () = GlobalLog.print ("Preparing nodes.\n")
val () = Vector.app (fn x => ignore (Simulator.RawEvent.schedule
    (Time.zero, fn _ => makeNode bootstrapAddr x))) nodes

(* stop event *)
fun stopSim _ =
    ( GlobalLog.print "Stopping Simulator.\n"
        ; OS.Process.exit OS.Process.success
    )
val _ = Simulator.RawEvent.schedule (Time.fromSeconds 300, stopSim)

(* run *)
val () = GlobalLog.print "Running simulator.\n"
val () = Simulator.run ()
```

Bibliography

[1] E. J. Berglund and D. R. Cheriton. Amaze: A multiplayer computer game. *IEEE Software*, 2(3):30–39, 1985.

[2] Ashwin Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. *SIGCOMM Comput. Commun. Rev.*, 34(4):353–366, 2004.

[3] Ashwin Bharambe, John R. Douceur, Jacob R. Lorch, Thomas Moscibroda, Jeffrey Pang, Srinivasan Seshan, and Xinyu Zhuang. Donnybrook: enabling large-scale, high-speed, peer-to-peer games. *SIGCOMM Comput. Commun. Rev.*, 38(4):389–400, 2008.

[4] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: a distributed architecture for online multiplayer games. In *NSDI'06: Proceedings of the 3rd conference on Networked Systems Design & Implementation*, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.

[5] Luther Chan, James Yong, Jiaqiang Bai, Ben Leong, and Raymond Tan. Hydra: a massively-multiplayer peer-to-peer architecture for the game developer. In *NetGames '07: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 37–42, New York, NY, USA, 2007. ACM.

[6] The SpoVNet Consortium. Spovnet: An architecture for supporting future internet applications. In *Joint EuroNGI and ITG Workshop on Visions of Future Generation Networks (EuroView2007), July 23-24 2007*, 2007.

[7] Associated Content. Why does world of warcraft cost a monthly fee? `http://www.associatedcontent.com/article/584795/why_does_world_of_warcraft_cost_a_monthly.html`, 2008.

[8] James Cowling, Dan R. K. Ports, Barbara Liskov, Raluca Ada Popa, and Abhijeet Gaikwad. Census: Location-aware membership management for large-scale distributed systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, USA, 2009. USENIX.

[9] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26, 2004.

[10] A. K. Datta, M. Gradinariu, M. Raynal, and G. Simon. Anonymous publish/subscribe in p2p networks. *Parallel and Distributed Processing Symposium, International*, 0:74a, 2003.

[11] Jauvane C. de Oliveira and Nicolas D. Georganas. Velvet: an adaptive hybrid architecture for very large virtual environments. *Presence: Teleoper. Virtual Environ.*, 12(6):555–580, 2003.

[12] Blizzard Entertainment. World of warcraft(r) subscriber base reaches 11.5 million worldwide. `http://eu.blizzard.com/en/press/081223.html`, 2008.

[13] Bryan Ford. Structured streams: a new transport abstraction. In *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 361–372, New York, NY, USA, 2007. ACM.

[14] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[15] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi. *ACM Trans. Graph.*, 4(2):74–123, 1985.

[16] Shun-Yun Hu and Guan-Ming Liao. Scalable peer-to-peer networked virtual environment. In *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, pages 129–133, New York, NY, USA, 2004. ACM.

[17] Shun-Yun Hu and Guan-Ming Liao. Von: A scalable peer-to-peer network for virtual environments. In *IEEE Network, vol. 20, no. 4, Jul./Aug. 2006*, pages 22–31, 2006.

[18] Patric Kabus and Alejandro Buchmann. Design of a cheat-resistant p2p online gaming system. In *International Conference on Digital Interactive Media in Entertainment and Arts 2007*, 2007.

[19] Patric Kabus, Wesley Terpstra, Mariano Cilia, and Alejandro Buchmann. Addressing cheating in distributed massively multiplayer online games. In *ACM SIGCOMM workshop on Network and system support for games*, 2005.

[20] Bjorn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. volume 1, 2004.

[21] Limewire.org. Gnutella protocol specification wiki. `http://wiki.limewire.org/index.php?title=GDF`.

[22] Michael R. Macedonia, Michael J. Zyda, David R. Pratt, Donald P. Brutzman, and Paul T. Barham. Exploiting reality with multicast groups. *IEEE Comput. Graph. Appl.*, 15(5):38–45, 1995.

[23] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *1st International Peer-to-Peer Symposium (IPTPS 2002)*, pages 53–65, 2002.

[24] Katherine L. Morse, Lubomir Bic, and Michael Dillencourt. Interest management in large-scale virtual environments. *Presence: Teleoper. Virtual Environ.*, 9(1):52–68, 2000.

[25] Times Online. Computer games to out-sell music and video. `http://business.timesonline.co.uk/tol/business/industry_sectors/technology/article5085685.ece`, 2008.

[26] Jeffrey Pang, Frank Uyeda, and Jacob R. Lorch. Scaling peer-to-peer games in low-bandwidth environments. In *IPTPS: International workshop on Peer-To-Peer Systems*, 2007.

[27] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, 2001.

[28] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[29] Michael Scharf and Sebastian Kiesel. Head-of-line blocking in tcp and sctp: Analysis and measurements. In *Global Telecommunications Conference, 2006. GLOBECOM '06. IEEE*, pages 1–5, 2006.

[30] Arne Schmieg, Michael Stieler, Sebastian Jeckel, Patric Kabus, Bettina Kemme, and Alejandro Buchmann. psense - maintaining a dynamic localized peer-to-peer structure for position based multicast in games. In *IEEE International Conference on Peer-to-Peer Computing 2008*, 2008.

[31] Randall R. Stewart. Stream control transmission protocol. RFC 4960 (Proposed Standard), 2007.

[32] 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, New York, NY, USA, 2001. ACM.

[33] Wesley Terpstra, Christof Leng, Max Lehn, and Alejandro Buchmann. Channel-based unidirectional stream protocol (CUSP). In *The 29th IEEE International Conference on Computer Communications*, San Diego, USA.

[34] Wesley W. Terpstra, Jussi Kangasharju, Christof Leng, and Alejandro P. Buchmann. Bubblestorm: resilient, probabilistic, and exhaustive peer-to-peer search. In *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 49–60, New York, NY, USA, 2007. ACM.

[35] Wesley W. Terpstra, Christof Leng, and Alejandro P. Buchmann. Brief announcement: Practical summation via gossip. In *Twenty-Sixth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2007)*, pages 390–391, New York, NY, USA, 2007. ACM Press.

[36] Tonio Triebel, Benjamin Guthier, and Wolfgang Effelsberg. Skype4Games. In *Proc. of the 6th Annual Workshop on Network and Systems Support for Games: Netgames 2007*, Melbourne, Australia, 2007.

[37] Tonio Triebel, Benjamin Guthier, Richard Süselbeck, Gregor Schiele, and Wolfgang Effelsberg. Peer-to-peer infrastructures for games. In *NOSSDAV '08: Proceedings of the 18th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 123–124, New York, NY, USA, 2008. ACM.

[38] Bruce Sterling Woodcock. An analysis of mmog subscription growth. `http://www.mmogchart.com`, 2008.

[39] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical report, Berkeley, CA, USA, 2001.