Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Semester Thesis SA-2006-31
Summer Semester 2006

Scalable Real-time Multiplayer Game Network Support for Orxonox

Benjamin Grauer, Patrick Bönzli
grauerb@ee.ethz.ch, boenzlip@ee.thz.ch

Tutor:	Károly Farkas	farkas@tik.ee.ethz.ch
Supervisor:	Prof. Bernhard Plattner	plattner@tik.ee.ethz.ch

Abstract

Computer games are based on many different cutting edge technologies like efficient and realistic graphical simulation of a world, physically correct interaction with the game world, network based multiplayer gaming and many more. Game development therefore is a very complicated topic and involves the application of knowledge from many different specialty fields like programming, linear algebra, numerics, physics, signal processing, communication networks and requires broad artistic skills. People who like to work in the game development industry often feel overwhelmed by the huge amount of knowledge this work requires.

In this semester thesis a group of students were introduced to the art of developing games based on a framework called Orxonox. The students got an introduction about how the Orxonox engine works and afterwards they extended the framework with a project of their own. In the end they presented their work in the form of a presentation. During their work we assisted the students to get the informations they needed, helped them solving programming, design and logical issues and kept them motivated.

Current online games are evolving towards massive multiplayer online games (MMO). Such games normally require multiple servers to handle the high amount of concurrent simultaneously playing gamers. Even nowadays networked games are still based on client-server architecture and are therefore not scalable due to limited network and processing resources. In the second part of our semester thesis we developed a scalable multiplayer game architecture for Orxonox. We enhanced the Orxonox network framework by adding proxy servers, which not only distributed the load, but also added redundancy in case of server failure.

Acknowledgments

No thesis is a solo venture. It's a collaborative effort requiring the support, guidance and advice of many. We would like to thank all of these people who were so generous with their time, insights, inputs throughout this process.

Special thanks to our tireless semester thesis advisor, Károly Farkas. Thank you for your inputs, great ideas and inspirations.

Thank you to our students that worked so well with us together in the Practical. You did a very well job, we never expected so much dedication from you.

Contents

Chapter 1

Introduction

Computer game programming involves a variety of speciality fields reaching from numerical analysis to mechanics. It therefore offers students a broad area of application for what they learn in the lectures. As a part of the students education at the Department of Information Technology and Electrical Engineering (D-ITET) [1] at the Swiss Federal of Technology Zurich (ETH Zurich) [2], the students have to attend to practical exercises called PPS (Chapter 3).

In the first part of our semester thesis we organized such a PPS for ten students. The students learned how to use open source tools to develop open source software, got a deeper introduction into the object oriented C++ language and they were introduced to some standard open source graphics and sound libraries. In the main part of our exercise they engaged themselves in a project of their choice. Chapter 2 gives a more detailed introduction into Orxonox [3], the software we ware using as a base for the student's projects. Chapter 3 describes the organization of the PPS and all student's projects.

In the second part of our semester thesis we developed a scalable multiplayer network architecture. This architecture not only meets the trend towards massive multiplayer online games (MMO) but also introduces redundancy in case of a server failure, keeping the network more stable. Our approach distributes server logics to proxy servers, therefore removing the bottleneck of a central server. We implemented this architecture into Orxonox as a prove of concept. Chapter 4 describes the implementation and goals in greater depth.

Chapter 2

Orxonox

In this chapter we first give an insight into how Orxonox came into existence and how it has evolved until today. Then we provide a quick guide on installation and configuration. At the end of the chapter we presents an overview of the general framework and its modules and how the main parts of the engine are built together, concluding with a section on development in Orxonox.

2.1 Project Orxonox

2.1.1 Description

Orxonox [3] is an open-source 3D action game that has been developed mostly by students of the Department of Information Technology and Electrical Engineering (D-ITET)[1] at the Swiss Federal of Technology Zurich (ETH Zurich)[2]. Orxonox serves as the application layer for the networking part (Chapter 4) of the thesis, and was also the base application for the practical part of the thesis (Chapter 3).

Built on top of many well known opensource libraries, like the Simple Directmedia Layer (SDL)[4], the Open Graphics Library (OpenGL)[5] and others. Orxonox is released under the GNU Public License (GPL)[6]. As such it is an open source project and everything created in it is free of charge and all the audio-visual resources and code-sources can be examined and used freely by the public.

2.1.2 Team

At the moment Orxonox has two maintainers, about fourteen developers, two designers and three musicians.

- Maintainers: Overlook the entire program code, assure module compatibility, decide on what code goes in the project, and what stays out. Additionaly write most of the engine code.
- Developers: Create and extend game engine modules. They also create story and entities for the game and its worlds.

- Designers and Musicians: Patch up graphics and sounds to make the worlds the developers generate seem realistic and authentical.

Twentyone part-time participants may sound as quite a lot of people, the resources needed to build a game from scratch are enormous and must not be misjudged. Section 2.2 contains more informations about this topic.

2.1.3 History

Orxonox was founded by the Patrick Bönzli and Benjamin Grauer in May 2004, roughly two years ago. The idea came up after we had finished the second intermediate diploma, when we had again time to fullfill old dreams.

At the beginning fighting with limited skills of C++ and some minor object oriented programming the limiting factor was the small resources in knowledge and manpower. In the hopes for more support and a greater insight into the world of programming we organized a convention where all the developers, also students from the our department and newly interested people were invited to join and discuss the future work related to Orxonox. It was a great success. Talking a lot about new ideas, concepts and sharing knowledge, we found new talented developers, in an environment that provided a good social experience burning the midnight coal, and eating what was frizzling on top of it.

As the months passed we organized a second reunion where we found new artists for visual and audible content and again we exchanged a lot about the game concept and further techniques involved with programming.

Since all previous meetings went so good we tought about giving a practical exercise at the ETH for younger students so that they could learn and profit from the project and certainly to further extend Orxonox resources.

So we launched the first approach of the practical exercise in the winter semester of 2005/2006. The result was an amazing achievement both in the leaps the engine went forward and the way the students seemed to appreciate it.

We talked about giving another practical exercise, but both, our study and the practical exercise were too much to handle together. So we decided to make a semester thesis on the Orxonox practical exercise and through it combine the fun and the determination with work.

The product of this second practical exercise and the involved work on Orxonox is presented in this thesis.

2.2 Game Development Model

This section begins with an overview on complexity of developing games, where some of the many aspects are displayed. Then we present how to align the development process into a phases, concluding with a description of the current state.

2.2.1 Complexity of Developing Games

Developing games is a very complicated process, and many different aspects have to be taken into consideration. Besides manufacturing source code there is also need to attend to many other factors, some of them are mentioned below.

Publicity in the Open Source Community

In the open source scene it is necessary to find people to help freely, so one needs to have a good reputation, a webpage, forums, maintain a server with easy access, provide documentation about the core concepts, attend to mailing lists, chat with new, interested users and much, much more.

Designing Environments

A game without content and design is like a lion without teeth: Quite cute, but not impressive. On the design level, it is a constant struggle to find people that are willing to help. Since most programmers see the beauty in code itself rather than in the visual representation a totally different kind of people has to be found, that concern themselves with graphics, music and design, namely designers. There are also many other aspects that have to be considered, like the universe the game plays in, the characters, and many many more.

Scientific Work

Saying that developing games is not science ends when starting to develop one.

There are many modules within a game all with a high level of complexity. For once there is a *model loader*, where one needs *material properties*, which again need *textures* and *shaders*, to make simple entities look interesting, there is the *sound engine* that gives the level the audible atmosphere, and makes visual events realistic in a three dimensional environment. A *collision detection system* with an efficient intersection algorithm is required for entity interaction, a *resource manager* to handle all the loaded resources, a *physics engine*, a *network module*, *graphical user interfaces* and an extended *math library*, just to take a few.

All these modules have to be interconnected by a controlling unit.

Only when all of these needs are satisfied it is possible to talk about “Developing Games”, and without the support of many, the few would never succeed.

2.2.2 Development in Phases

Orxonox has a huge framework that extends and expands in all the directions a modern 3D game as well as many other applications need. To guarantee the success of an ongoing development a modular framework that can be developed in parallel at all the different building blocks is the main requirement.

But it is not only the sourcecode that must match the demands of modularity. As depicted in Figure 2.1 there are many different layers with totally different aspects that can be worked on.
To keep a constant overview on the state, and the development involved with the stage, we split the work on Orxonox into three main phases, that are listed below.

Phase One: Basis

At the beginning all the participants came together discussing the ideas and basic concepts that would flow into the product. After gathering enough data

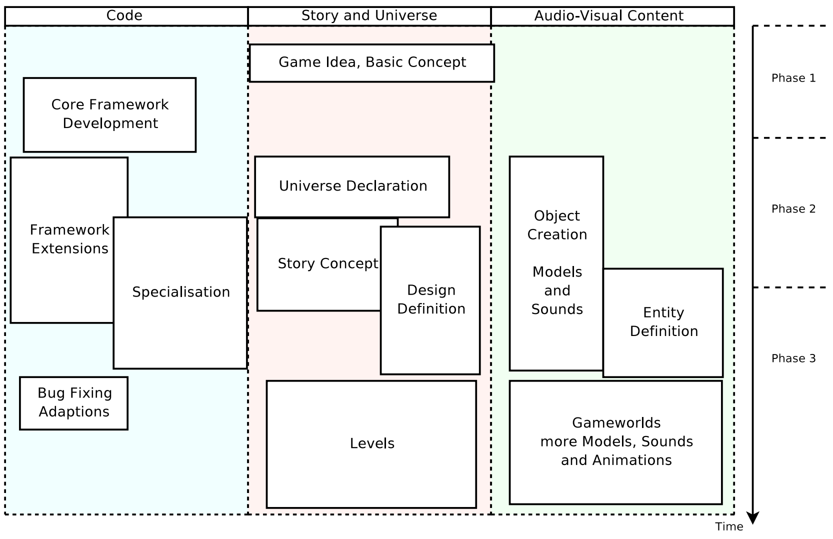


Figure 2.1: Parallel Game Development

on what direction the game would take, the core framework development was started, where all basic modules like positional nodes, the graphics engine, and many more were implemented.

Due to the high level of abstraction and complexity of the code foundation it was almost impossible to split the work into more than two parts. Also multiple iterations over the code sometimes forced us to work alone at adaptions making development a single threaded process.

At the end of this phase the basis of the project was founded, waiting for input on a less general level.

Phase Two: Specialization

While phase one was mainly a work on fundamentals where a only a few people were involved, phase two switched to a more parallelized workflow.

Making a game realistic means, that at first there has to be an environment, that supports that kind of realism. That is why at the beginning of this phase there was again an idea gathering, where Orxonox' universe and its basic story concepts were discussed.

Simultaneously the framework was extended to match the needs of the game's universe. On this part many more people were involved (see Chapter 3).

From the design side there was now also the need to create entities and sound effects, that fit into the newly defined environment, so all kinds of artists were searched and motivated to work on the project.

Following the concept development in all of these sectors, a more finegrained specialization has to be worked out to make a real game.

Phase Three: Enter the Game

After putting together many different models, a code framework, concepts and a complete storyline, everything that remains is to create levels on top of it, that match the atmosphere the definition of the universe provides.

The final stage is intended to purely extend the content and make code fixes only where bugs or a real need occur.

To represent the game this is the most important phase, because it alone is what attracts endusers.

2.2.3 Current State

At the moment Orxonox development is situated between phase two and three, where still some code has to be adapted and invented, but also far enough to start developing realistic worlds and integrating parts of the story into levels. As can be predicted, it is not always easy to draw a clear line between the phases and sometimes it is very hard to keep track of all the different directions the engine, graphics and concepts are aiming at. That is why it is inevitable to meet from time to time, gather the thoughts of everyone and join them together, so that the project may flourish from the accumulated data and not be smashed under it.

2.3 Engine Overview

In this section an overview of the main framework modules and the game itself is given.

2.3.1 Orxonox API

Orxonox API (Application Programming Interface) is broad, and it is beyond the scope of this thesis to go into all the details, so here we give only an abstract model overview of the engines main modules. On the projects wiki page [7] and also in Appendix A the different classes and modules are described in more detail.

When entering a gameworld - which could be either a game level or the game menu - the desired level is loaded, all *entities* like enemies, ships and environmental objects are placed into the scene and the music gets ready to be played.

After the loading stage the level is started and set into *run* mode, that is essentially an infinite loop called the ”main loop”. In the main loop, all the modules are updated according to their updating policy as depicted in Figure 2.2. For a smooth animation it is desired, that one cycle must not be longer than the 25th part of a second.

To illustrate the performance needed for one cycle the main modules are described in shortly below. A more module-oriented view is presented in Appendix A.

1. Input Handling: Grabs all user input, and input from the window manager. The input is then relayed to the objects, that are registered to receive it, and handled by them.
2. Network Synchronization: Collects all the changes since the last time and relays the difference through the network. This is described in more detail in Chapter 4.

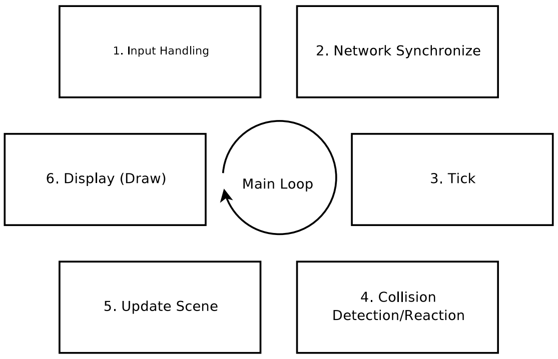


Figure 2.2: Orxonox Main Loop

3. Tick (Timeslice Propagation): In this stage all the time dependant entities, like spaceships, missiles and particles are propagated. To approximate the timeslice that passes to calculate the next frame the averange of the last ten frame calculation times is used. In this way the illusion of an even motion is produced.

4. Collision Detection/Reaction: For interaction between entities inside of the gameworld the intersections of each entity with all others are checked and acted upon. For example a projectile can collide with an enemy. Thus the projectile must explode, and the enemy should get hurt which is then graphically represented with a particle fauntain, and audible accompanied with the sound of an explosion from the direction the collision occurred.

5. Update Scene: After everything is handled, moved and reacted upon the scene must be updated and made ready to be displayed. For this to work all transformation data of all moved objects are recalculated. Also many other things like music buffers are refilled and updated.

6. Display (Draw): Finally the most important stage for a graphical application, the rendering is executed. All currently visible entities within the gameworld are drawn according to their drawing disciple using OpenGL.

As seen above, there is quite a lot of processing to be done and a lot of hard work to be invested to make all these modules as modular and optimized as possible.

2.3.2 Gameplay

Since there are multiple game modes[1] supported by Orxonox such as vertical scrolling, free flight or first person shooting it is not possible to give a uniform explanation on how the spaceships, persons and vehicles are operated. We are committed to implement all the controls as intuitive as possible so that learning how to master entities comes as easy as possible.

In general there are four different game modes and a short description on the usage of each one is given below.

[1]Game mode: a way to play the game

Game Mode 1: The Game Menu

In the game main menu the user can traverse through the options with the mouse, selecting from various levels, campaigns and other options. A screenshot of the game menu can be seen in Figure 2.3.

Game Mode 2: Free Flight

The free flight mode of the game is activated when the user travels inside of a spacecraft with the camera behind it. Here the user can use the configured buttons (default Up = 'w', Down = 's', Left = 'a', Right = 'd') to fly through the level, look around by bringing the mouse into motion and firing the weapons with the 'fire button' (default = 'Left Mouse Button'). A picture of the mode can be seen in Figure 2.4.

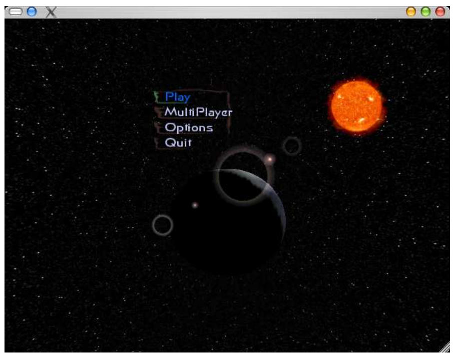


Figure 2.3: Game Menu

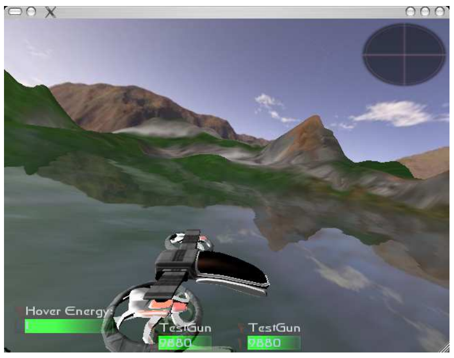


Figure 2.4: Free Flight Mode

Game Mode 3: First Person Shooter (FPS)

In the first person view the user is represented as a person through whose eyes one can see the world. The person can walk on the ground, enter vehicles and shoot with guns in his hands. A view of such a mode is represented in Figure 2.5

Game Mode 4: Arcade

In the final game mode the camera is positioned above the spacecraft, that carries the player. The background and enemies flow from the top of the screen to the bottom, and one must destroy as many enemies as possible while staying alive dodging from alien attacks. Such a 'from top' view is depicted in Figure 2.6

2.3.3 Installation

Orxonox is portable and runs on many well known platforms like Linux, Windows and Mac OSX, although we mainly support Linux as in our opinion it is the most powerful development environment for open source software.

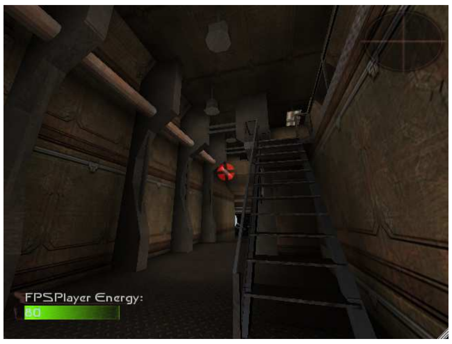


Figure 2.5: FPS Mode

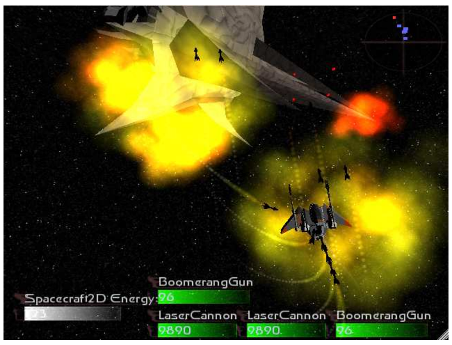


Figure 2.6: Arcade Mode

Since Orxonox is still not mature enough to release binary versions users have to compile the source on their own. Assuming, that a development environment suited for Orxonox is given the following procedure will lead to a successful installation:

1. Download source and unpack
2. Compile

   ```
   ./autogen.sh && ./configure && make
   ```

3. Download system-independent data
4. Install

On our wiki page [7] all the information on and around installation and the setup of a development environment can be found by following the *Installation* link.

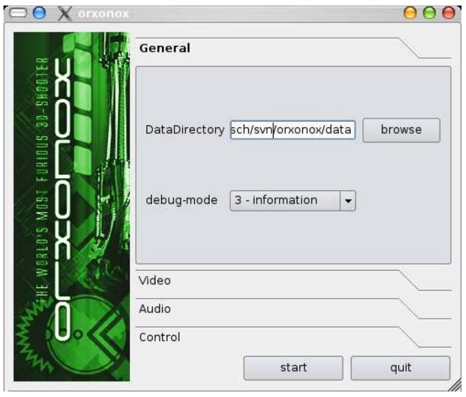
2.3.4 Configuration

For the configuration of Orxonox a graphical users interface (GUI) can be used by starting Orxonox out of the console with the option '-g' or by starting up for the first time.

One important option that must be set correctly is the DataDir in section General, otherwise the application does not know where the game resources can be loaded from.

Here we also provide a short overview on what can be found in the main configuration sections.

- General: System independant data and debugging level (manly for developers);
- Video: Graphic options for finetuning the performance and resolution setup;



DataDirectory: sch/svn/orxonox/data

debug-mode: 3 - information

Figure 2.7: Orxonox GUI

- Audio: The volume of the ingame music and sound effects can be setup here;
- Control: Configuration menu for the keyboard, mouse and joystick control.

Chapter 3

PPS - Practical Exercise for Students

3.1 Introduction

The Orxonox practical exercise was a one semester course, where students learned the basic concepts involved with "developing games". The course was supervised by Parick Bönzli and Benjamin Grauer (the authors of this thesis) with the main goal to manage, maintain and supervise the work of the students.

3.1.1 Motivation

Motivation of the Supervisors

The motivation to make a practical exercise with the project Orxonox was twofold.

Firstly in the ever growing project the need for more programmers and designers arose. Implementing new modules takes a long time, designing, developing, testing and integrating are all very important tasks, and finetuning them to match the desired outcome requires even for an experienced programmer a huge endurance.
The second motivation factor was, to pass on the gathered knowledge and experience from coding the base of Orxonox to people who are also interested in programming computer games. Many students of D-ITET are enthusiastic in learning programming skills, but they never really get the chance in doing it. Giving the students the possibility to teach themselves while playing around with already existing code and taking credit(s) for it was a very good way to start programming.

Motivation of the Students

We think, that for the students the main motivation was, that they could build a module inside of a bigger project, learn C++, modeling, and how to work in a team to achieve a goal greater than what they could have done by themselves.

Working on a game might also be a motivation factor, because many of today's programmers start with wanting to program their own game. Unfortunately (but not for Orxonox), nowadays it is impossible to create a game alone, so joining a team is the best solution.

Another factor was that they received six PPS credit points for the work they did, but we all know that this would never satisfy their hunger to go further and work as hard as they did.

3.2 Objectives

As supervisors we had the role of teacher, maintainer and motivator. What we did in each of these topics is described in more detail below:

3.2.1 Teaching

It was very important, that the students got a good basic insight into the project. For this we arranged meetings where the basic concepts of game programming and also the core modules of the Orxonox API were explained.

Besides the concepts of game programming also more general concepts like STL (Standart Template Library) and the C++ object oriented programming were discussed.

3.2.2 Maintenance

The second objective was to manage a team of seven students and maintain the code and ideas they generated in a friendly but efficient manner.

People who are new in programming often do not see the need for creating modules. It was our job to look at the design and fragment the code they produced. Also we had to give constant feedback to questions concerning the already existing code, and how they should integrate their own modules.
To handle the groups in an 'as simple as possible' fashion, each of us maintained half of the projects.

3.2.3 Motivate the Students

The third objective was to motivate the students to spend even more time on design, code and content than what can be expected to get.

Fairly speaking, three hours a week are nothing when it comes to programming and content creation. It is important to keep the students constantly motivated and to show them, that the their efforts will be appreciated and used inside (and maybe outside) of a bigger project.
The way we managed to motivate the students was to let them have fun. If it was programming what they wanted or modeling new spaceships or levels, reaching the goal made it worthwile.

3.3 The Practical Exercise

3.3.1 General PPS

The PPS (Projects, Practicals and Seminars) in general is a fix part of the bachelor course at the department of Electrical Engineering and Information

Technology. Each PPS takes 14 weeks and serves to achieve the following goals:

1. To enhance skills in practical work, team work, preparation and presentation of one's work; acquiring of knowledge in learning and project methodologies as well as enhancing motivation to look into the basics and applications of Electrical Engineering and Information Technology.

2. Procurement of knowledge about building up systems as well as enhancement of general knowledge.

3. Procurement of skills in the area of Electrical Engineering and Information Technology that are useful for the remaining terms as well as during one's work life.

3.3.2 Orxonox Practical Exercise

In the Orxonox practical exercise the main idea was to give the students an insight into the world of programming, and to teach them how to make the most efficient use of open source utilities, used in most of today's open source development environments.

The students received six credit points for the three hours they had to be present each Wednesday afternoon.

The practical exercise was split up into a theoretical and a practical part.

Theoretical Part

In the theoretical part the students learned the basic concepts of the Orxonox framework, and of all the essential modules needed to work with Orxonox API.

Practical Part

The students should learn programming while being able to play around with an advanced, modular and extendable framework. The work involved the implementation of their own free-standing module, and the integration of their achievements into the framework of Orxonox.

At the end they had to incorporate the module into a gameworld and present it with slides at a final presentation.

Matching PPS Structure

To match the requirements of a PPS described in Section 3.3.1 a short interpretation of each point is given here.

1. The practical exercise was a teamwork, where all the students worked together on a big project, while acquiring the insight into the basic concepts of game and also more general programming.

2. Building up the own module, untouched by other modules, but using the underlying structure as learned in the theoretical part.

3. Learning the skills of programming, design and presentation, that will for sure be used in future projects and work in the area of Electrical Engineering and Information Technology.

3.4 Work Scheduling and Course Structure

One of the most significant things in a practical exercise is to make full use of all available time, and for this we created a time schedule what all the students had to follow.

As depicted in Figure 3.1 the work flow was split up into three major phases, each with a work and presentation part.

Into this form each group had to integrate its own work plan and time scheduling.

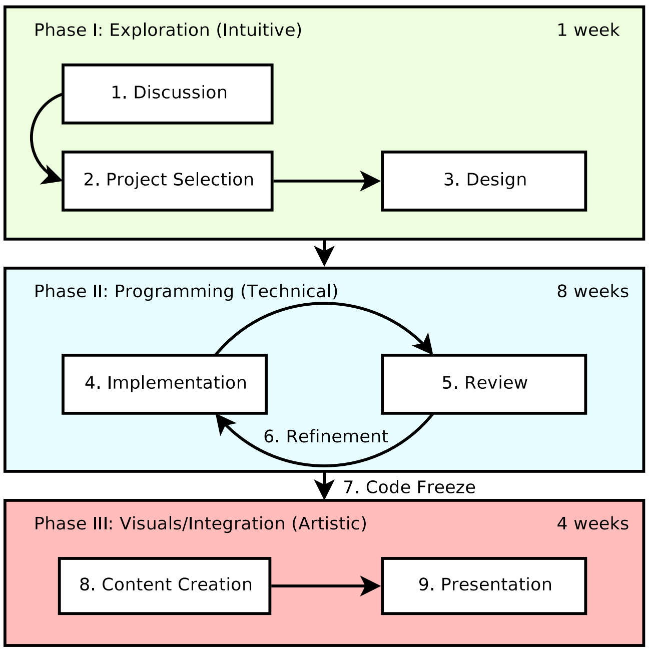


Figure 3.1: Work Flow of the Practical Exercise

1. Discussion - Where will we go together (two hours):
   In the first afternoon we discussed with the students what they expected from the project Orxonox, in what direction they wanted it to go, and how we all together imagined the future of Orxonox. Also all the already available PPS-projects were presented.

2. Project Selection (one week):
   During the first week the students had to decide what topic they'd like to work on. On the wiki page of the PPS[8] under section 'Projects' there is a list of all the available projects. On the grounds of what was discussed the week before, everyone selected a project, that met the constraints of the PPS and the main goal of the Orxonox project.
   The list of the modules that were chosen by the students can be found in Section 3.5

3. Project Exploration (one week):
   The following week each group had to create a timetable and adapt it

to the scheduling explained here. Furthermore an UML-diagram (Unified Modeling Language diagram)[9] of the chosen project had to be generated. At the end, the concept ideas were reviewed and finalized together with the supervisor.

4. Implementation (four to five weeks):
The next step was to implement the conceptual idea from the exploration phase in C++. The module had to be as self-contained as possible to avoid overlapping. In this phase the students were supposed to learn the importance of coding clear interfaces.

5. Review Phase (one afternoon):
It is important for less experienced programmers and people new to any project that they receive feedback. In this phase the supervisor examined the work. The code was read through, and minor design flaws were rethought.

6. Work Refinement (two weeks):
From here on the integration into the existing framework of Orxonox started and the fixed flaws from the review phase were reimplemented.

7. Code Freeze
In this stage the feature development was stopped meaning that nothing new was to be implemented anymore. From this time foreward only severe bug-fixes were to be dealt with.

8. Content Creation (three to four weeks)
To impress external gamers and interested viewers this was the most important part. Since most people will not read through the big amount of code the students have created, they had to generate a world to demonstrate their achievements in a visual manner.

   This was done by creating a new world and integrating the new features into the scene.

9. Presentation (one afternoon)
Finally the students used a presentation toolkit to create a presentation about their work and had to rephrase what they wanted to say, and together with their content part demonstrate it. The final presentation took place on the 5th July.

3.5 Results

The results of the PPS were astonishing and much more than we expected. All the students invested at least double if not tripple the time they had to spend on the PPS, writing code, designing new spaceships, creating levels, writing documentation and talking about the concept of the whole game.

Every single module was a full success, and the students had (at least most of the time) a lot of fun implementing and learning the not so trivial concepts of programming and design.

3.5.1 Story Concept Paper by *Benjamin Knecht*

The concept paper was an approach to implement the basics of game story and design.

The story spanned from the definition of a science-fiction universe over the sentient races, their technology and social structures down to details on how conflicts merge and resolve during the course of the game and how the player experiences the whole story.

The second part of this project was a design definition on how the ships and races should look like. It also included ideas on style of weaponary and big technology like hypergates.

3.5.2 Scripting Engine by *Silvan Nellen*

The scripting engine is a powerfull extension to the Orxonox framework which allows to write scripts that can control the behavior of all entities inside of the Orxonox world. The backend is the Lua scripting language [10] that is highly portable and wildly used both in games and professional applications.

3.5.3 Weather Engine by *David Hasenfratz and Andreas Mächler*

The weather engine (Figure 3.2) is a graphical extension to Orxonox allowing it to very easily bring a realistic environmental atmosphere into outdoor levels. The many effects created in this work cover rain, snow, fog and an animated cloudy sky.

All these effects have a default setting, and are highly configureable through either the scripting language, or dynamic level loading.

The rain and snow engines are a perfect example, that building a new module on top of an already existing module, namely the particle system, is possible in one semester.

3.5.4 Water Surface Simulation by *Stefan Lienhard*

Water surface simulation (Figure 3.3) enables a realistic rendering of ingame water. The surface is based on the new shader language GLSL (Open Graphics Shading Language)[11].

The process is therefor split into two parts: One is an entity, that renders a plane at the right position inside of a level, the other one the shader, that renders reflections, refractions, and wave translations.

3.5.5 Binary Space Partitioning by *Claudio Botta*

Binary space partitioning (Figure 3.4) is an algorithm, that splits up space and vertices inside of it into a binary tree. With this sorting it is possible to decide efficiently if an object collides with another object or a wall. The sorting also allows for a three dimensional culling algorithm, that automatically hides all object, that are either fully hidden behind another object, or that are behind the camera.

This module made it possible to create indoor levels with arbitrary size for the first person shooter part of the game.

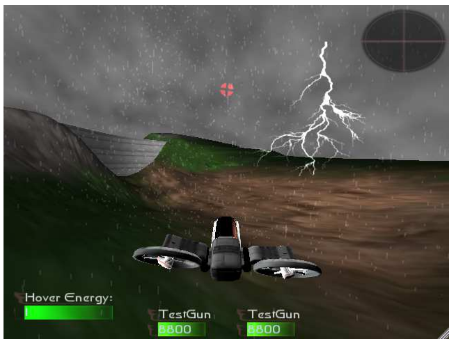


Figure 3.2: Weather Engine

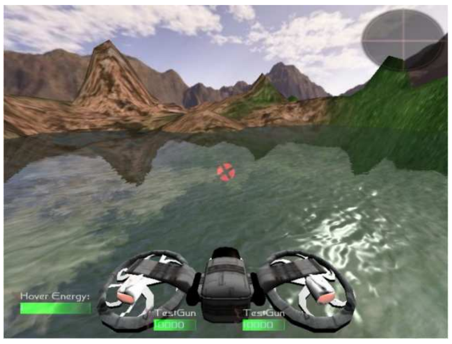


Figure 3.3: Water Surface

3.5.6 Network by *Christoph Renner*

Every serious game has network support to enable multiplayer games over the Internet or the local area network. We managed to build a simple network framework for Orxonox based on the TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) protocols. It proved to be very extensible (see Chapter 4), easy to use and stable. For testing purposes we manufactured a first person shooter (FPS) arena and a space simulator map. (Figure 3.5)

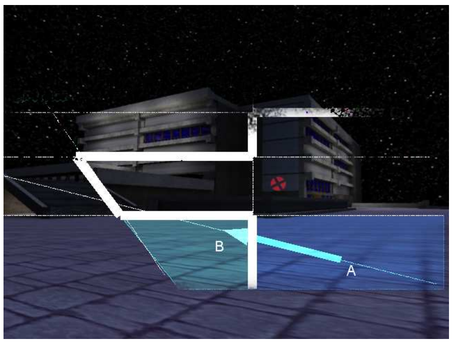


Figure 3.4: BSP

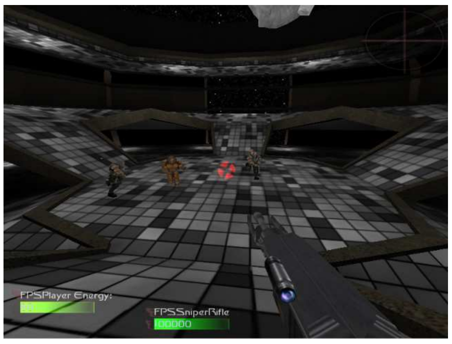


Figure 3.5: Network

3.6 Conclusions

What we have learned from one year of instructing students is, that with good motivation you can get a lot out of six PPS points. If it is possible, to show students, how interesting programming can be, and that what they are doing is important for the bigger picture, they are more than interested in working hard, and often, to reach their final goals.

Another very important aspect is to prevent students from developing interdependencies between blocks. Nothing can be more frustrating, than using a module that does not work, or that is not finished, or constantly changing. Because of this we realized, that giving the students standalone projects, that were all modules in themselves and independant of each other is the basic building block for time-limited work.

Last but not least, the students must be given the possibility to do what they want. That means, choosing their own project, going earlier one day, but staying longer another evening and so on.

We are very happy, that giving the students that kind of freedom helped boosting their workflow, and that they did not abuse it.

Although providing a casual environment for the students, they realized that their work was appreciated, and that through overcoming obstacles they learned more, and finally could be proud to present their achievements.

3.7 Future Work

As future work there will be a practical exercise in the winter semester 2006/2007, that is much more centered around content creation and less on programming. This does not mean, that the students will never see C++ code, but much more, that the students will learn to use the existing code to build levels and new entities.

The next practical exercise will be supervised by students of this semester's Orxonox course, and the previous supervisors will only be present as advisors.

We sincerely hope, that this constellation will come to happen, and through it the project will prevail, and the enthusiasm of the students will never vanish.

Chapter 4

Proxy Server Architecture

4.1 Introduction

Nowadays networks provide a fast and performance intensive infrastructure for multiplayer computer games. The subjective game performance depends heavily on the network responsiveness (network delay) and fault tolerance. The client-server and peer-to-peer topologies are today's commonly used game management models. Both suffer from poor scalability. This becomes increasingly critical for fast paced games like first person shooters. The bottleneck of the client-server topology limits current games to about 64 participating players at most. The limiting factor at such player numbers is the network bandwith at the server and its processing power.

4.2 Objectives

There is a number of goals in any software design. The first two topics describe the main goals of this semester thesis part. The latter two goals are of general nature and will not be explained in any detail but rather in terms of how design goals can be applied to this thesis.

4.2.1 Scalability

There is need to make multiplayer games more scalable. Scalability is the ability of a distributed game architecture to maintain service when the number of participating players increases.

Game developement is evolving towards large scale multiplayer games. The more players are able to join a game and play together, the more fun it is. From industry's side of view there is much interest in making multiplayer games more scalable.

4.2.2 Redundancy

The client-server topology has a single point of failure: in case of a server failure, the whole network will cease operation. Such failures often are the reason for

long and costly network downtimes.
To minimize the risk of such downtimes one should add some redundancy that dynamicaly reorganizes the network in a way, that operatability can be assured with a minimum of human interaction.

4.2.3 Modularity

Modularity in design means, that the design should strive to be architecture independant, in the sense, that it should be easy to integrate in any other (networked) game.

4.2.4 Reusability

Reusability goals are tightly coupled with the modularity goal stated above. The implementation should not be platform dependant. Keeping the APIs clean is an important goal that guarantees future reuse of the modules. Documentation is another important part that guarantees easier reusability.

4.3 Orxonox Network API

In this Section we give a short description about the Orxonox network API and the most important modules, datastructures and classes related to it.

4.3.1 History

The base network framework of Orxonox was developed during the PPS. It was designed and implemented by Christoph Renner and Patrick Bönzli. The development started in October 2005, in the following four months we managed to design and implement a framework which was able to synchronize data in a client-server environment using TCP protocol.
The framework we came up with did not yet fully satisfy our needs, so we did a major redesign and reimplementation starting in March 2006 for another four months. It runs on top of the UDP protocol and therefore shows better performance than the previous implementation. This part of the semester thesis is based on the new framework.

4.3.2 Overview

The goal of the Orxonox network framework is to synchronize an Orxonox game state from one network node to another. A game state is made of variables describing the actions the player is currently doing, the reactions of enemy players and the graphics, sound and input states of the game engine. Not all of these state variables need to be synchronized over the network in order to play together. We categorize the state variables in three types:

1. Variables that need to be synchronized over the network, we call them *SynchronizeableVar* (eg. player position and orientation). Their state is determined directly by user inputs which are only known to the local host.

2. Variables whose states are indirectly influenced by inputs of a local user and therefore can be deduced from the state of *SynchronizeableVars* (eg. position of objects attached to the player, a gun for example).

3. Variables that don't need to be synchronized over the network (eg. number of frames per second).

The goal in this categorization is to obtain a minimal set of variables needed to be sent over the network in order to minimize the required network bandwith. According to this categorization it is sufficient to only synchronize the *SynchronizeableVars*.

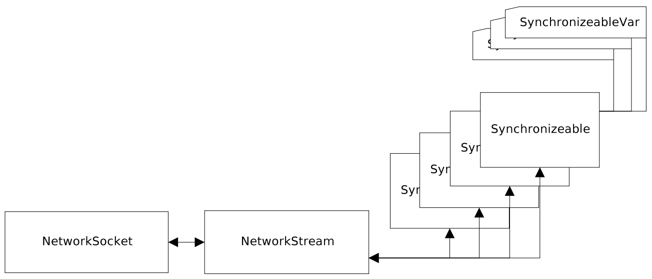


Figure 4.1: Orxonox Network Modules

We call the C++ class that contains such variables *Synchronizeable* (see Figure 4.1). It has got a list of *SynchronizeableVars* and an identification number called *uniqueId* which serves as an identification address in an Orxonox network. This number must be unique within one Orxonox multiplayer network session. Once such a container is created and a *uniqueId* is assigned to it, this object will automatically be created on every remote host connected to the same server (and they all got the same *uniqueId*). From this moment on the content of this *Synchronizeable* will be kept synchronized until it is destroyed (more about the synchronization process in Section 4.3.3). The *NetworkStream* references to a list of *Synchronizeables*. To get the current game state, it gets all variable values from them and makes a game state image out of it (a binary stream containing all variable values sequentially). This image is then passed to the *NetworkStream* which constructs one or multiple UDP/TCP packet(s) out of it and transmits the packet(s) over the network to the other hosts. The remote hosts are identified by a *hostId* which is assigned by the server.
The following sections describe the network functionalities in greater detail.

4.3.3 Synchronization Process

This process synchronizes an Orxonox game state from one network node to another, as it was already mentioned before. The main participants in this task are *SynchronizeableVars*, *Synchronizeables*, the *NetworkStream* and the *NetworkSocket* (see Figure 4.2).

If a synchronized variable gets changed the corresponding *SynchronizeableVar* will detect this. Next time the *NetworkStream* wants to create a state update this change will be contained in it. The *NetworkStream* is assembling an

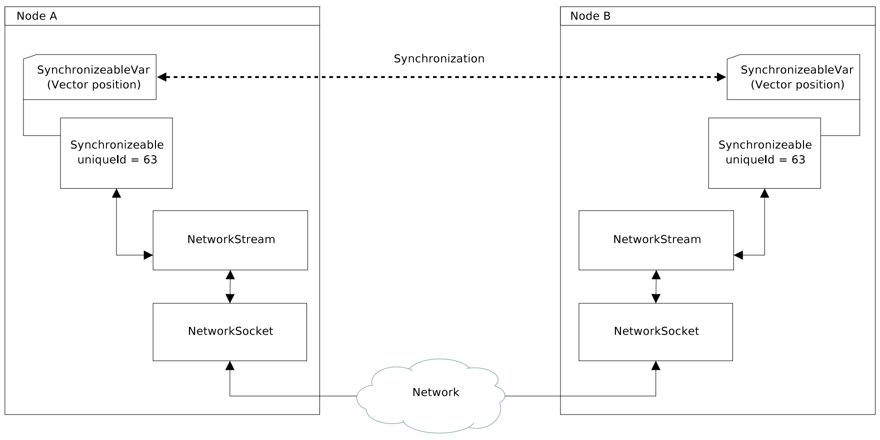

Node A
Node B
Synchronization
SynchronizeableVar (Vector position)
Synchronizeable uniqueId = 63
NetworkStream
NetworkSocket
Network
SynchronizeableVar (Vector position)
Synchronizeable uniqueId = 63
NetworkStream
NetworkSocket

Figure 4.2: Orxonox Network Synchronization Process

update by iterating through all of its active *Synchronizeables* and sequentially writing the *SynchronizeableVar* values in to an array. This array represents the state of the game at the current time (see also Figure 4.3).
But this state image is quite big, to gain more efficiency, Orxonox extracts the differences to the last state image. This is done by calculating the mathematical difference of the two images (as it is often done in video compression). Now there this is a sequence of `1`s and `0`s (`1` if the bit has changed, `0` if not). Such a binary array can efficiently be compressed using zlib [12], since it contains long series of `0`s and `1`s. After adding some state information (some state identifiers) to this array, this data is passed on to the *NetworkStream* and included in a UDP or TCP network packet. The *NetworkStream* sends the packet over the network to the other host. Should the packet get lost during this transmission the packet will be resent later. In case of TCP packets the error handling is covered by the protocol itself. Sooner or later the remote host receives the state image and uncompresses it. Then it creates the current state using its last state (remember we transmitted only the state difference). This state update is then applied to the local state.
Example: Imagine a new player connecting to the game. The server will locally create a player character (a space ship or a creature) for the new player. Since this player character is a *Synchronizeable* the object gets created on all other network nodes automatically. If the player starts to move, the position and orientation of the object will change (they are *SynchronizeableVars*) and therefore be written to all other network nodes. So the player character will move on all other hosts as well.

4.3.4 Packet Creation

If you are looking at an Orxonox network packet, it is almost impossible to read any information out of it, since it is a compressed differential image. If you disable compression, the images will be more readable since the overall structure of the state image remains.

An Orxonox packet consists of a main header with length and state information and a list of sections for each *Synchronizeable*. Each Section contains a sub-header with information about the *Synchronizeable* itself and the list of variable values belonging to the *Synchronizeable* see Figure 4.3

This synchronization process normaly runs with approximately $60Hz$ if the server is fast enough. This frequency can be altered if needed to lower/increase network delay and bandwith usage.

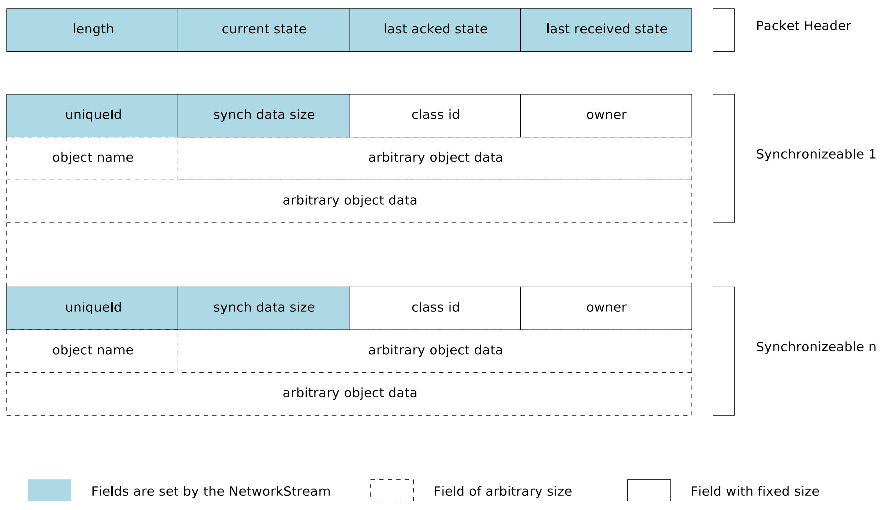


Figure 4.3: Orxonox Network Packet

4.3.5 Start a Connection: Handshake

Before a client can join a game, it has to accomplish a handshake with the server. Orxonox has its own handshake, including:

- a comparison of the network framework version numbers to assure compatibility;
- the *uniqueId* of some central *Synchronizeables* that the client needs from the server;
- the server assigns a *hostId* to the client;
- the preferred nick name for the player.

After this handshake the client is ready to receive the current game state and then to join the game. In detail the *Handshake* class is a *Synchronizeable* and its attributes are exchanged via *SynchronizeableVars* so it fits itself perfectly into the Orxonox network framework. The game map name and objects contained in the game world do not need to be created via the handshake since they are all *Synchronizeables* and therefore will be initialized and created automatically. Theoreticaly this handshake can initialize any information one would want to be initialized before a network game can start.

4.3.6 Sending Messages

Another way to communicate between hosts is by sending messages. The *MessageManager* is a special *Synchronizeable* that offers the possibility to send such messages to any node in the network. This works similar to message queues in the Unix interprocess communication. Only the receiver's *hostId* needs to be known. There are some special receiver addresses:

`ALL` a message sent to this id will be received by all network nodes in the whole network.

`ALL_BUT_ME` this is the same as `ALL` except, that the local host does not get the message.

`ALL_BUT_HOST` this message will be sent to all hosts in the network except a host that can be specified as another argument.

`SERVERS` a message sent to this id will be received by all servers in the network.

`HOST` such a message is only sent to one specific host.

The message body can be of arbitrary length. Messages are used to signal new or leaving clients, changing nick names and so on. For the *proxy server architecture* this module was heavily extended.

4.3.7 Modules

In the following we give a list of the most important network modules and data structures with each a short description of what they're doing. The UML diagrams of these classes do only reflect the most important functions. Appendix A presents the modules in full detail.

All modules are described using an UML diagram that serves as a short overview. All diagrams are used in the following way (Figure 4.4):

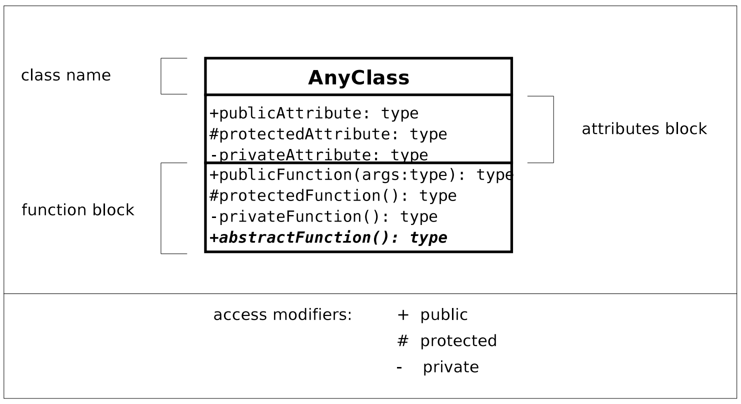


Figure 4.4: Legend of Symbols in the UML Diagrams

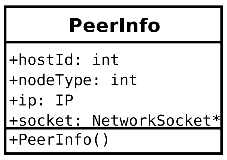


Figure 4.5: PeerInfo Class

Peer Information

The *PeerInfo* structure represents a remote host the local node is connected to. This structure is managed by the *NetworkStream* (see Figure 4.5).

The *PeerInfo* contains an identifier for each host called `hostId`. Each network node can have a special function (client, *master server* or *proxy server*), this information is stored in the `nodeType` variable. The remote host can also be identified via an IP address `ip`. If there is already a connection setup, it can be referenced through `socket`.

Synchronizable Variables

The *SynchronizeableVar* class is the smallest part in the synchronization process. It represents a variable that needs to be synchronized over the network (Figure 4.6).

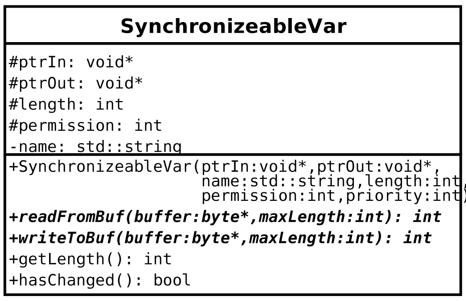
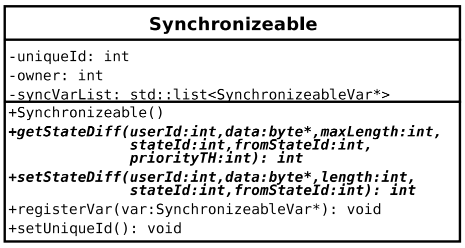

SynchronizeableVar
#ptrIn: void*
#ptrOut: void*
#length: int
#permission: int
-name: std::string
+SynchronizeableVar(ptrIn:void*,ptrOut:void*,
name:std::string,length:int,
permission:int,priority:int)
+readFromBuf(buffer:byte*,maxLength:int): int
+writeToBuf(buffer:byte*,maxLength:int): int
+getLength(): int
+hasChanged(): bool

Figure 4.6: SynchronizeableVar Class

A *SynchronizeableVar* contains a reference to the variable (`ptrIn` and `ptrOut`) that needs to be synchronized over the network. These variables are written to the state image using the `writeToBuf(...)` function. If there is a variable update coming in, it will be read from the image by the `readFromBuf(...)` function. A *Synchronizeable* can check if a variable changed by accessing the interface function `hasChanged()` which returns `true` if there is a local change. This class registers itself to a *Synchronizeable*, from then on it will be automatically updated with the interface functions that were just described. Each variable has a specific permission mask which specifies which other hosts are able to over write its value. For debuging purposes each *SynchronizeableVar* has an additional `name` indicating the function of the variable in the game.

Synchronizables Objects

The *Synchronizeable* class represents a container holding variables that need to be synchronized. Most of the Orxonox classes that are drawn in the world actually are *Synchronizeables* (see Figure 4.7).



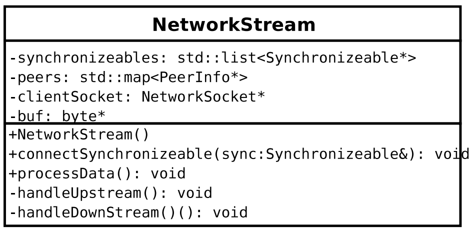
```
Synchronizeable
-uniqueId: int
-owner: int
-syncVarList: std::list<SynchronizeableVar*>
+Synchronizeable()
+getStateDiff(userId:int,data:byte*,maxLength:int,
              stateId:int,fromStateId:int,
              priorityTH:int): int
+setStateDiff(userId:int,data:byte*,length:int,
              stateId:int,fromStateId:int): int
+registerVar(var:SynchronizeableVar*): void
+setUniqueId(): void
```

Figure 4.7: Synchronizeable Class

The *Synchronizeable* manages a list of *SynchronizeableVars* (`syncVarList`) and is responsable to create its part of the state image. The *NetworkStream* will get its state image by calling the `getStateDiff(...)` function and vice versa will write an image to it by `setStateDiff(...)`. Only those variables will be affected, that have the right permissions. New state variables can be added to the *Synchronizeable* class using the `registerVar(...)` function. Once a *Synchronizeable* object with a valid *uniqueId* is created on a host, it is automatically created on all other hosts as well. They all get the same identification number. From this moment on, the *SynchronizeableVars* will be synchronized continuously until their *Synchronizeable* is destroyed.

Network Stream

The *NetworkStream* is the most central class in the Orxonox network framework (Figure 4.8). This class also assembles the state images and passes incoming images from the network socket to its *Synchronizeables*.



```
NetworkStream
-synchronizeables: std::list<Synchronizeable*>
-peers: std::map<PeerInfo*>
-clientSocket: NetworkSocket*
-buf: byte*
+NetworkStream()
+connectSynchronizeable(sync:Synchronizeable&): void
+processData(): void
-handleUpstream(): void
-handleDownStream()(): void
```

Figure 4.8: NetworkStream Class

It manages a list of *Synchronizeables* (`synchronizeables`) and a list containing all hosts connected to this node (`peers`). If this node is a server it will also listen to incoming connection via the `clientSocket` network socket. State images are written and read in a local buffer called `buf`. The Orxonox main-loop calls the `processData()` function with a certain frequency. This function handles the down- and upstream separately in the specific functions

`handleUpstream()` and `handleDownstream()`. They create, assemble or read the state images.
The *NetworkStream* can also start a network connection to another server by calling `connectToServer(...)`. This function automatically creates a new *PeerInfo* for this host and starts the handshake process. If the node is in server mode it can be initialized with the function `createServer(...)`.

Network Socket

This class represents a network socket and serves as a low level interface to the operating system's hardware (Figure 4.9). Network operations are executed using the sdl-net library [13] to assure interplattform compatibility. The *NetworkSocket* is an abstract base class for the *UDPSocket* and *TCPSocket*.

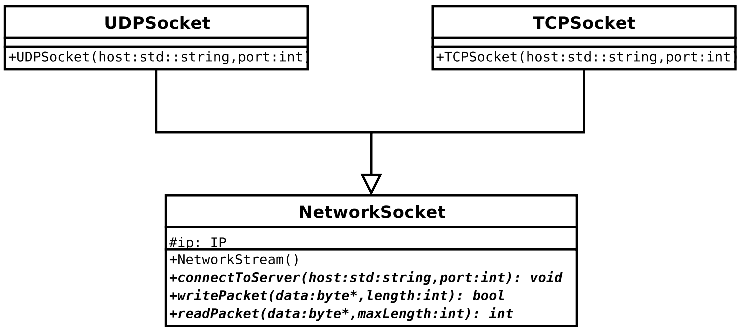

UDPSocket
+UDPSocket(host:std::string,port:int
TCPSocket
+TCPSocket(host:std::string,port:int
NetworkSocket
#ip: IP
+NetworkStream()
+connectToServer(host:std:string,port:int): void
+writePacket(data:byte*,length:int): bool
+readPacket(data:byte*,maxLength:int): int

Figure 4.9: NetworkSocket Class

In this class the network packets are manufactured and sent over the network (`writePacket(...)`) and received from the network (`readPacket(...)`).

4.3.8 Further Information

There is a full UML diagram of the network module on our Orxonox wiki page [7]. Here the programmer is able see all modules and their interdependencies. As a programming reference we suggest to use the doxygen pages [14] or to directly dive into the source code which is documented very well.

4.4 Network Topologies and Scalability

To ensure scalability a suited network topology has to be found. Therefore we give an overview about the different network topologies used today, their assets and drawbacks. The following analytical analysis, partly based on "GSM: A Game Model for Multiplayer Real-time Games" [15].

4.4.1 Peer-To-Peer Topology

Ten years ago most games used to have a peer-to-peer topology (Figure 4.10), because it was easy to program (every node had exactly the same information and was could therefore execute the same algorithms) and very stable. Even if one of the peers disappears the network is able to continue operation. The first versions of Doom [16] fully based on peer-to-peer networking. The commonly played RTS (Real Time Strategy) game was also programmed using this topology.

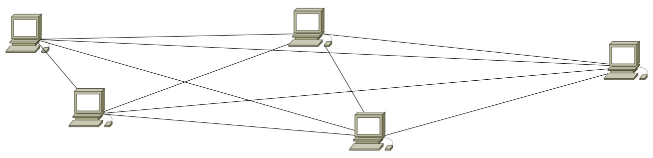


Figure 4.10: Peer-to-Peer Topology

4.4.2 Client-Server Topology

This is the commonly used topology for nowadays FPS games (Figure 4.11). Even nowadays RTS games are switching from the peer-to-peer to client-server topology because of the huge amount of network traffic created by this game type. This topology is well suited for games with up to some tens of players. Such topologies are specially vulnerable to server failures.

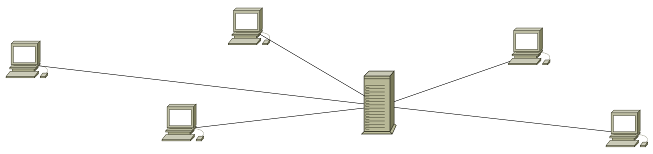


Figure 4.11: Client-Server Topology

4.4.3 Proxy Server Topology

This topology has not yet been used by any fast paced networked game on the market right now (Figure 4.12). It shares the required network and processing power by distributing the load to multiple servers. Both the peer-to-peer network topology and the client-server topology are combined to achieve this goal.

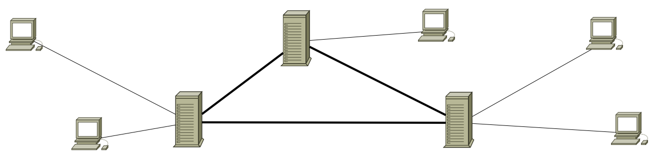


Figure 4.12: Proxy Server Topology

According to GSM a measure for the traffic load on a server is given by the following formula:

$$D^{in}_{proxy}(l,n,m) = \underbrace{\frac{n}{l}d_{cin}}_{\text{actions of locals}} + \underbrace{(n-\frac{n}{l})d_{cin}}_{\text{updates of remotes}} + \underbrace{(m-\frac{m}{l})d_{rsu}}_{\text{remote game entities}},$$

$$D^{out}_{proxy}(l,n,m) = \underbrace{\frac{n}{l} d_{cout}(n,m)}_{updatesoflocals} + \underbrace{(\frac{n}{l} + \frac{m}{l}) d_{rsu}}_{alllocalentities},$$

where

$m \in \mathbf{N}$	number of entities in the game world;
$n \in \mathbf{N}$	number of clients ;
$l \in \mathbf{N}$	number of proxies ;
$d_{cin} \in \mathbf{R}$	incoming message size from directly connected clients;
$d_{rsu} \in \mathbf{R}$	incoming message size from clients connected via proxies ;
$d_{cout} \in \mathbf{R}$	outgoing message size.

We have changed the formula slightly since there is no traffic multicasting available for the semester thesis.
As we can see, with increasing n the network traffic is only increasing with the speed proportional to $\frac{n}{l}$. To gain scalability, we have to add more proxy servers to keep the $\frac{n}{l}$ low, this suffices our scalability requirement. By adding proxy servers we do also increase the inter-server traffic $(n - \frac{n}{l})d_{cin}$ which becomes dominant as soon as $\frac{n}{l}$ gets small.

4.5 Design

This section describes the design of our *proxy server architecture*. First we describe the network topology and its components, then we show how these components are used by describing a usage scenario. Afterwards the main modules used in the usage scenario are identified and described. As a last point we present the most important communication scenarios in depth.

4.5.1 Network Topology and it's Components

The proxy server topology is a mix between the client-server and the peer-to-peer topology. It introduces scalability by distributing game logic to several servers. One server is selected to manage the network, it is called the *master server*. The other servers are referred to as *proxy servers* and will only handle the inputs of the locally connected clients. All servers are interconnected in a peer-to-peer topology, this assures, that information is spread in the fastest way possible.
This topology introduces a lot of advantages: On one hand it certainly is more scalable. On the other hand it implicitly introduces the redundancy we need. In case of a server failure the client would only need to be reconnected to one of the other servers. Even if the *master server* fails, another *proxy server* can take its role over.
However this indirection also has a disadvantage: By introducing more servers, we also introduce a new level of indirection. The client-to-client communication delay is therefore increased by the communication delay between the servers. However experiments in gaming communities show, that with a communication delay of about 100-120 ms a FPS game is playable, anything beneath it is not actively felt as delay anymore by the human mind (see *A Traffic Analysis of a Busy Counter Strike Server* [17]). It seems that games are still playable if the

delay is not higher than 180 ms.
The most important components in the proxy server topology are:

Master Server

The *master server* is the server in command. It is responsable for network organization. Its common tasks are:

1. Handling new clients: New clients connect to the *master server*. If the *master server* has no free slots available anymore, the client is redirected to a *proxy server* with free resources.
2. Handling leaving clients: Clients that left the game will have to be removed from the network plan and their game states will need to be removed from the game.
3. Dynamical client redirection: Clients can be dynamicaly reassigned to another server during the game.
4. Network growth: When the number of clients is increasing the *master server* will elect *proxy servers* to handle new clients.
5. Network shrinkage: The *master server* can decide to remove *proxy servers* from the network to shrink the network.
6. Server failure: If a server fails another server takes its place over.

Since game state information is not uniformly distributed in the network (because of link delays) game decisions based on a local game state will not have the same result on all hosts. Therefore important game decisions should be by only one node in the network, the *master server*. Some of its special responsabilities are:

- Game goals: The game mission is controlled by the *master server*;
- Damage: Collisions are handled by every node locally but the damage resulting from a collision is evaluated by the *master server* only. Also player deaths are handled here;
- Creating players: new clients will each be assigned a new player. The player is created and initialized by the *master server* only;
- Removing *Synchronizeables*: Objects in the game world can be removed by the *master server* only.

Proxy Server

The *proxy servers* manage their local client pools. They can be suspended and activated. If they are not yet elected by the *master server* to accept clients they are in *passive mode*. A *proxy server* chosen to handle clients will switch to the *active mode* and start the synchronization process (more details about this process in Section 4.5.4). Such as the *master server*, the *proxy servers* need to keep track about the network plan. They will listen to all joining and leaving messages and update their local network plan accordingly. This is very important, since a proxy server needs to be able to switch to master server mode in case of a server failure.

Client

The client just forwards the player inputs to the next server, which evaluates them and sends a state update back to the client and the other servers.

4.5.2 Application Scenario

Setup of the Scenario

An administrator sets up a dedicated Orxonox server, the so called *master server* (*MS*). During the setup process he will specify a pool of other dedicated servers, which will function as *proxy servers* (*PS*s) in case, the number of players on the master server exceeds the local maximum number of players. The *MS* and the *PS*s are started, while the *MS* loads the game world and starts processing client inputs the *PS* will just wait in an inactive state for an activation command from the *MS*.
The players will connect to the master server as one would expect until the master server reaches the maximum number of players. Up to this point, our network framework doesn't work differently from a normal client-server network except that in a normal client-server network new clients would be rejected from this moment on.

Proxy Server Election

In our multiserver network the *MS* elects one of the *PS* from its server pool and activates it. This election is based on individual network bandwith and delay properties of the servers. Then the main server forwards the current world state to the new *PS* so that it knows which level and objects to load and where the players are positioned at the moment.
The *MS* requests the client to reconnect by giving it a list of alternative *PS*s. The client decides on its own which *PS* is suited best. Since there is only one *PS* up in this scenario the client chooses this one. From now on the network consists of one *MS* and one *PS* each with their clients. Should the number of players still grow, the *MS* elects another proxy server and suggests new clients to connect to it. For performance reasons the next *PS* is woken up when the $(n-1)$th client connects to the *MS*. So the *PS* got some time to load and synchronize and the connection delay for the next client can be kept as low as possible.

Inter-Server Synchronization

There are two network topologies working together now: the client-server star topology and the peer-to-peer topology of the servers. Each server handles two different synchronization traffics: The input based traffic from the clients and the world state updates from the proxy servers. Both traffic have to be merged to one new coherent world state and will be distributed again over the network to the clients. Only the world state changes from the local clients of each server will be sent to the other servers.

Network Shrinkage

When the clients leave the game and the number of clients drops below a certain threshold, the clients have to be reorganized: one *PS* will give its clients to the other servers and switch back to inactive mode. From there, the *MS* can reactivate it as soon as there are more clients available again. The threshold value depends on statistical properties like the mean time to next client connection. Since switching clients and turning up/down proxy servers are a very costly operation in terms of time, its always better not to restructure the network too fast.

4.5.3 Modules

Overview

According to the application scenario there are some core functionalities involved with this *proxy server architecture*:

- the *master server* controls the *proxy servers*;
- the *master server* assigns clients to servers;
- all servers keep track about the current network topology (the network plan);
- network configurations need to be centrally accessible.

From these requirements we can now derive the modules we need for the implementation. The first two tasks are both about controling the network, so we put them together in one module called *ProxyControl*. To keep track of joining and leaving network nodes there is another class, the *NetworkMonitor*. To centrally access the network configurations the *NetworkSettings* class can be created (see Figure 4.13

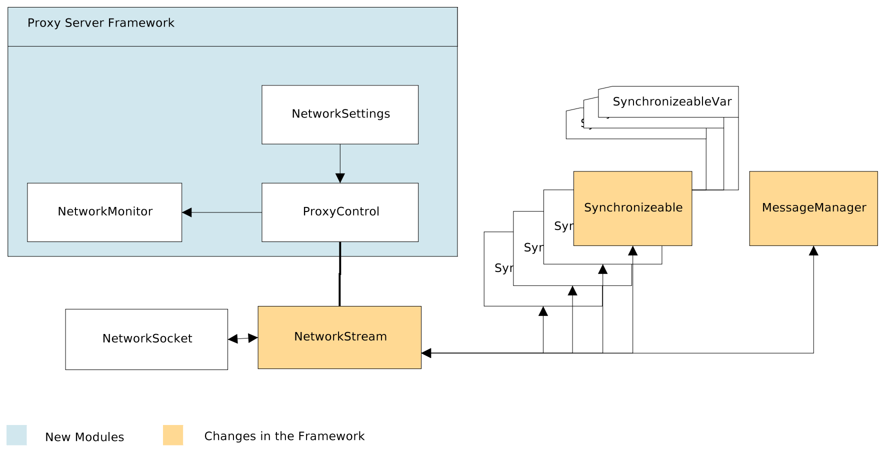


Figure 4.13: *Proxy Server* Modules

Proxy Control

The *ProxyControl* module is responsable for all inter-server communication. Messages are all created, sent and received in this module. Furthermore the *ProxyControl* decides about when and how network reorganizations take place. Therefore it always needs to know the current state of the network, which can be via the *NetworkMonitor* module.
This module can run in two different modes: it only will work actively if the server is a *master server*. Otherwise the module won't do anything.

Network Monitor

This module monitors the network and keeps a prevailing network plan. It interfaces directly with the *ProxyControl* and the *NetworkStream*, which notify this module about changes in the network. The network plan is represented in the form of a network graph.

Network Settings

This module serves as an interface to the network configuration files. These files specify the player limit and the *proxy server* pool. The *NetworkSettings* only interface with the *ProxyControl* module.

4.5.4 Network Processes

After knowing the main modules, we describe how these modules interact to accomplish their work.

Handling new Clients

A new client wants to join the game. As usual it first starts the handshake process with the *master server*. We extended the old handshake, with additionaly synchronizing a list of active *proxy servers* and a redirection flag (indicating if this client must reconnect to another *PS*). As long as the *master server* has enough capacity to handle new clients, it will accept them. But as soon as the *MS* has no more free slots, the client will be indicated to reconnect to another *PS*. Clients can decide on their own which *PS* to take. Reasons for one or another *PS* are:

Delay The clients measures the link delay to the server. As we stated in Section 4.5.1 the link delay in general gets worse in a proxy server network (because of the additional hop). Therefore this attribute needs to be minimized before all others. This attribute depends on the type and size of network between the hosts, the local and remote network buffer and the network polling frequency in Orxonox.

Bandwith The link bandwith needs to be maximized so that state images are transfered faster.

To evaluate these attributes a connection is created to the server to measure its delay and bandwith. With this information we can assign a quality value to the link with the following formula:

$$quality = c_1 * \frac{1}{delay} + c_2 * bandwith,$$

where

$c_1 \in \mathbf{R}$	constant indicating the importance of the network delay;
$c_2 \in \mathbf{R}$	constant indicating the importance of the network bandwith;
$delay \in \mathbf{R}$	the network link delay;
$bandwith \in \mathbf{R}$	the network link bandwith;
$quality \in \mathbf{R}$	the network quality.

Clients will choose the link with the best (meaning the highest) *quality*. All attributes of this equation can change frequently. It would be better to not only measure the quality at one point in time, but to also measure it over a longer period of time (see Section 4.9 for more information). As soon as the client has decided which *proxy server* to prefer, it will connect to it by going through a full handshake. In the moment the handshake is finished the *PS* will send a broadcast message to all other servers indicating that a new client has joined the game. The *MS* will then create a *Playable* (a playable avatar) and assign it to the new client (see Figure 4.14).

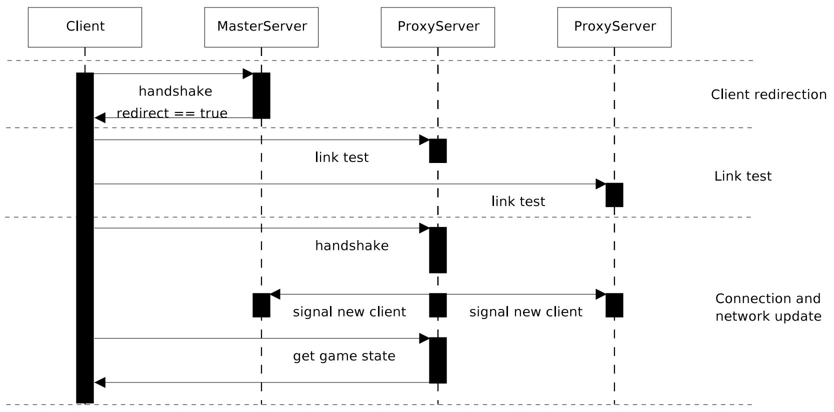


Figure 4.14: Connection Process

All servers receiving the broadcast message will update their network plan via the *NetworkMonitor* to reflect the new state.
A *proxy server* can also reject a client (for example because it just used the last slot for another client), in this case the client will connect to the *proxy server* with the second best link quality. The client will keep on doing this until it either finds a free slot or runs out of *proxy servers*. In the latter case the client will reconnect to the *master server* to get a new list of *proxy servers*.

Client Leaving

There are three ways a client can leave a game (Figure 4.15):

1. By signaling: The client sends a goodbye message.
2. By timeout: The client does not send any traffic for a specific amount of time and therefore times out.

3. By banning: The game administrator can kick a player off from the game.

The first way is how clients normaly disconnect from a network game. If the user decides to leave, he will click on the exit button, which will send a goodbye message to all servers. But what happens if a client computer crashes or a user decides to just kill the Orxonox application? In this case there is no signal sent and therefore the server will not recognize that the client has left the game. After a timeout the server will automatically assume, that the client has left the game without signaling it.

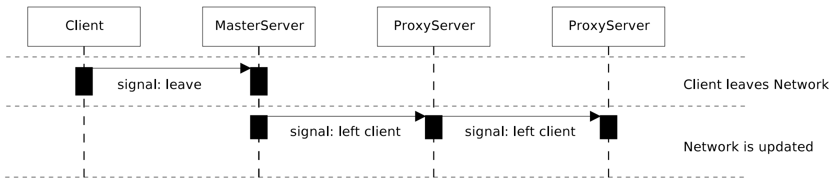


Figure 4.15: Disconnection Process

As soon as the server detects the client disconnection, it immediately removes its *Synchronizeables* and broadcasts a message to all nodes in the network to do the same. All servers will also remove the client from their locally managed network plan to reflect the change.

Dynamic Client Redirection

Once clients are connected to the multiplayer game they can be redirected to other servers if need should arise (for example if the network has to be reorganized). Only the *master server* can send such a reconnection command (see Figure 4.16). There are two different reconnection commands possible:

Hard Reconnection The client disconnects from the old server and reconnects to the new server by initializing a handshake. The old game state is lost and the player will join the game again as a new player (memoryless).

Soft Reconnection The client does not really disconnect from the game. It initializes the connection to the new server without handshake by just changing the synchronization server.

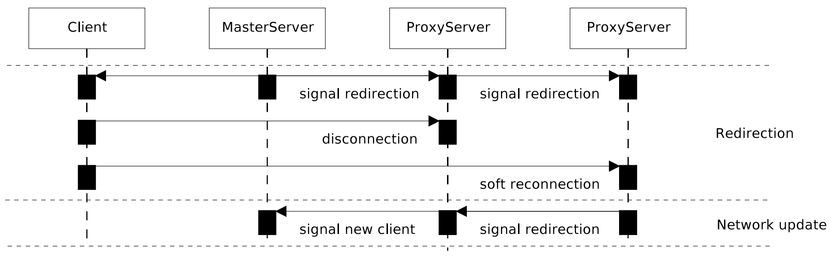


Figure 4.16: Reconnection Process

In both cases, the *master server* broadcasts a reconnection command to servers and the client to be reconnected. When the other servers receive the

reconnection command they remove the player from their network plan and add it again as soon as the client reconnects to the new server. Let's take a look at the first type of reconnection, the *hard reconnection*. In this case, the client will disconnect from the server and reinitialize a connection to the new destination server. The client will join the game again at this server as a new player, his old game state is therefore lost.
In the second case the client will not disconnect until it has a new connection setup with the new server. This new connection is handled without handshake. As soon as the new connection is working, the old link is closed. The player is never leaving the game and except for a short re-synchronization time will not see anything of this reconnection process. When the client is reconnected to the new server, the server will broadcast a message to all servers indicating, that a new client has connected to it.

Network Growth

The *master server* manages a pool of *proxy servers*, defined in the network configuration file. At the beginning all *proxy server* are in passive state, which means, that the servers are up and running but are not yet connected to any other server. If need arises to extend the network the *master server* will elect one of the servers from its pool to be activated (Figure 4.17).

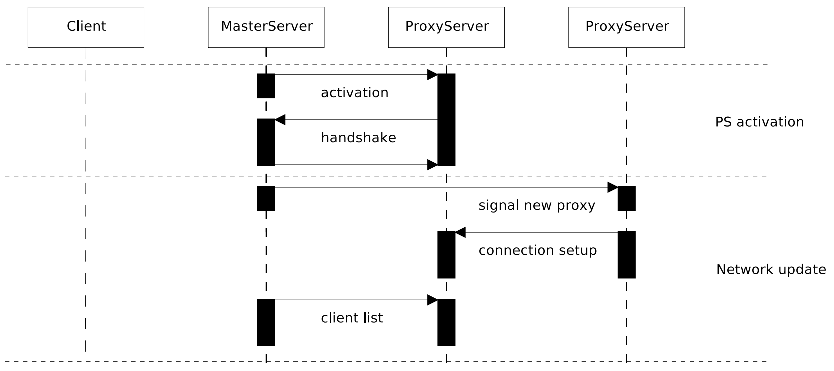


Figure 4.17: Network Growth Process

The *master server* decides when time is ready to select a new *proxy server* for activation. At the moment the server does this decision in a very simple way. More advanced algorithms would be preferable (see Section 4.9).

$$\underbrace{(l+1)*n_{max}-n}_{\text{number of free slots}} > n_{max},$$

where

$n \in \mathbf{N}$	number of clients ;
$n_{max} \in \mathbf{N}$	player maximum per server ;
$l \in \mathbf{N}$	number of proxies .

The equation just states, that the minimal number of free slots in a game should be bigger than the maximum amount of players one server can handle. This means, that as soon as the *master server* starts it will activate a *proxy server*. This is for redundancy reasons, if this inequality holds there will always be enough free slots to compensate one server loss.

The activation process is very simple: it's just a normal connection establishment in reverse order. So the *master server* initializes a connection to the *proxy server* and starts the handshake (in all other cases the client starts the connection). After the handshake finishes the *proxy server* has a complete game state and is ready to accept new clients. The *master server* sends a signal to all other servers, indicating that a new *proxy server* was activated. As soon as the other servers receive this signal, they update their network plan and initialize a connection to the new *proxy server*. At this moment the network plan of the new server contains all other servers. To complete the plan the *master server* sends a list with all clients connected to the servers. Now the *master server* adds the *proxy server* to the proxy list that is automatically synchronized during the handshake process. Once this is done the network is stable again.

Network Shrinking

If clients leave the game, not all servers in the network will be fully occupied anymore. The network will start to fragment, if there are no new connections to fill the free slots. The network will reach a state, that all clients could also be handled by less servers if they would be rearranged.

Figure 4.18 shows a client connected to a *PS* that is been selected for suspension.

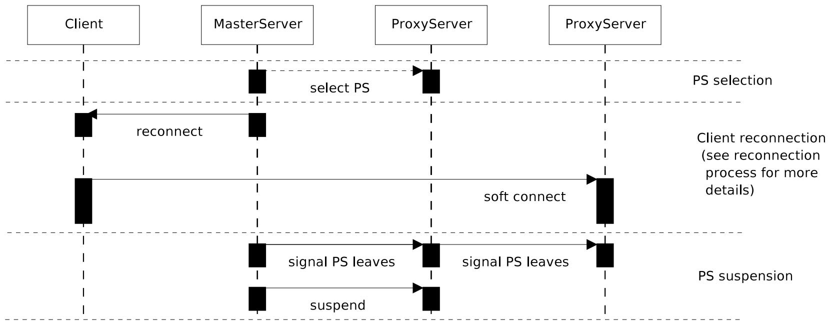


Figure 4.18: Network Shrinkage Process

The *master server* waits with any reorganization of the network until the number of clients in the game is beneath a certain threshold (normaly a multiple of the server's player limits). Then the *master server* will shrink the network by selecting a *proxy server* to be put in sleep mode again. It removes the server from the internal server list so no new clients will be able to connect to the server anymore. The *master server* will force clients from this *proxy server* to reconnect to other servers, by sending a reconnection command (Section 4.5.4). As soon as all clients reconnected to the other servers the *master server* will broadcast the new network state to all servers. The *proxy servers* will then terminate all network connections and switch to passive mode again.

The shrinkage threshold definitely is something that should be handled in a more sophisticated way, since there is much overhead involved with a network reorganization. We discuss more ideas about a better algorithm in Section 4.9.

Server Failure

There are two scenarios our *proxy server architecture* can deal with:

- *Master server* failure;
- *Proxy server* failure.

In both cases the reaction of the network is well defined and will remain stable. The clients will just reconnect to another server. This will happen after the connection timeout. In case of a master server failure the *proxy server* with the highest *hostId* will be chosen to replace it. The new *master server* will then send a message to all hosts indicating this change, all servers will then update their network plan (see Figure 4.19).
After this the network is stable again, the *master server* will activate a new *proxy server* from its pool according to the algorithm stated in Section 4.5.4. According to the network growth algorithm there should always be enough place to reconnect all clients from one server.

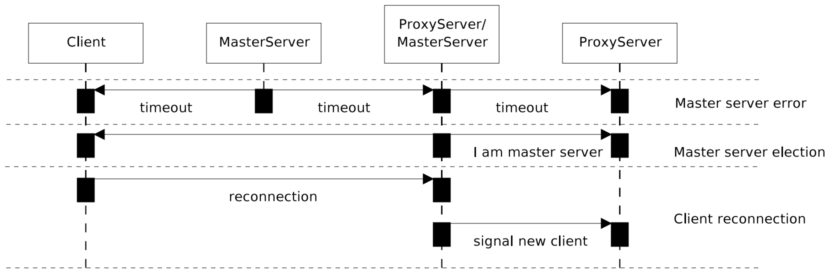


Figure 4.19: Network Failure Process

A server that looses connection to the network because of a failure will not be able to reconnect at the moment (see Section 4.9). Clients involved with the server failure will lose their game state since they will do *hard reconnection* to another server, in the future this could also be handled in a better way (see Section 4.9).

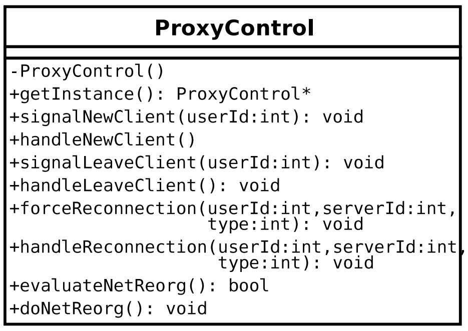
4.6 Implementation

Here we give some insight about the implementation of the modules described in Section 4.5.

4.6.1 Proxy Control

This module controls the whole proxy server network (see also Section 4.5.3).

ProxyControl is a singleton class meaning, that there is only one instance of this class (Figure 4.20). The static `getInstance()` function takes care of this requirement and returns the instance. Because of this singleton model the constructor is also private. The rest of its member functions are used for the



```
ProxyControl
-ProxyControl()
+getInstance(): ProxyControl*
+signalNewClient(userId:int): void
+handleNewClient()
+signalLeaveClient(userId:int): void
+handleLeaveClient(): void
+forceReconnection(userId:int,serverId:int,
                   type:int): void
+handleReconnection(userId:int,serverId:int,
                    type:int): void
+evaluateNetReorg(): bool
+doNetReorg(): void
```

Figure 4.20: ProxyControl Class

communication scenarios stated in Section 4.5.4. `signalNewClient(...)` is called when a new client connects locally, it sends a signal to all *ProxyControl* modules on every server node in the network. When this signal arrives at the other nodes the `handleNewClient()` function is called. This function updates the network plan managed by the *NetworkMonitor*. When a client closes connection to a server `signalLeaveClient(...)` is called on this node. This function also sends a signal to all server nodes in the network to let them know about this change. The other servers receive this message which will automaticaly trigger the `handleLeaveClient()` function. This function will again update the network plan to reflect this change.
The last function is for client redirection. Recall that there are two reconnection modes: *hard reconnection* and *soft reconnection* (see Section 4.5.4). The reconnection type can be specified as an argument in the `forceReconneciton(...)` function. This function reconnects a client from one server to another (identified by the *userId* of the node). Only the *master server* can call this function for security reasons. This function is called automaticaly if the network has to be resized. Evaluating if the network needs reorganization, can be done using the `evaluateNetReorg()`, which will return `true` in case that there is need. The `doNetReorg()` will then reorganize the network using the functions to force the clients reconnecting to other servers. The other nodes receive this signal and automatically call the `handleReconnection` function. The target will therefore be reconnected to the new server and the new server will reserve enough resources to accept the new client.
All these signals are sent via the *MessageManager* (described in Section 4.3.6).

4.6.2 Network Monitor

The *NetworkMonitor* is responsable for keeping the network plan up to date (Figure 4.21). This plan includes all network nodes, their types (*master server*, *proxy server* or client), and connections to other hosts. Each node is represented by a class called *NetworkNode* which includes all attributes stated above. An alternative representation of the described information is the *PeerInfo*, it can also be accessed via the *NetworkNode* interface.

NetworkNodes can be added or removed using the `addNode(...)` and the `removeNode(...)` interface functions. This call does only create a new node in the network plan, if this node does not already exist. In the latter case there will

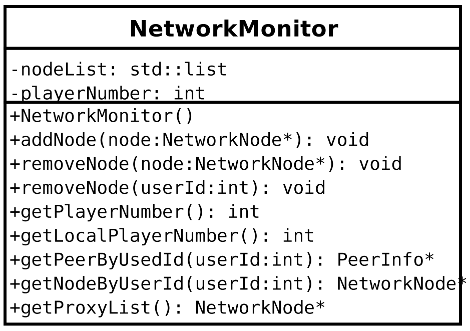

NetworkMonitor
-nodeList: std::list
-playerNumber: int
+NetworkMonitor()
+addNode(node:NetworkNode*): void
+removeNode(node:NetworkNode*): void
+removeNode(userId:int): void
+getPlayerNumber(): int
+getLocalPlayerNumber(): int
+getPeerByUsedId(userId:int): PeerInfo*
+getNodeByUserId(userId:int): NetworkNode*
+getProxyList(): NetworkNode*

Figure 4.21: NetworkMonitor Class

be only a connection added to this node. Any other module can get the current player number by the function `getPlayerNumber()`, which will return the player number in the whole network. If one needs to know the players connected to the local host one would have to use the `getLocalPlayerNumber()` function. Any node information can be accessed by the generic functions `getPeerByUserId(...)` and the `getNodeByUserId(...)`. Both return a reference to the nodes registered under the given `userId`.
To get a list of all *proxy servers* in the network the `getProxyList()` function can be used. This is usually done during the handshake process from the *master server*.

4.6.3 Network Settings

This module represents the settings of the network (Figure 4.22). It contains a *proxy server* list, the player limitations for each server. Also the *master server*'s address can be specified, which is not necessary for the network to work.

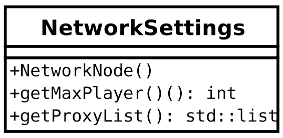


Figure 4.22: NetworkSettings Class

It just parses the network configuration file and offers this information to the Orxonox framework. The *NetworkMonitor* of the *master server* is initialized with this information.

4.7 Preliminary Measurement

4.7.1 Environment

To test the performance of the previous and the new network implementations the infrastructure of the ETH computer support group ISG [18] was used. Although they have very good machines both in graphics and networking performance, two major problems had to be faced:

First, since the network at the ETH is 100Mbit and not at all saturated by other traffic, it was almost impossible to measure a real-world example as it would be over real internet-connections.

Second, to measure network traffic, a root access is required, leading to measurements only on notebooks with limited graphical performance. Even worse for scanning was the fact that the measurements were taken on a switched network, making it impossible to sniff out packets.

4.7.2 Scenario

To compare the performance of the new proxy server topology against the old single client-server architecture, the following setup was used.

Client-Server based Scenario:

As depicted in Figure 4.23 the old setup is as expected one server, and multiple clients joining the session during startup.

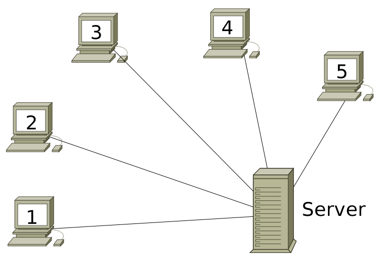


Figure 4.23: Client-Server Scenario

The traffic in this scenario was measured viewing out of one of the clients, and also as seen from the server.

Proxy-Server based Scenario:

The process of connecting 5 clients to the network is depicted in Figure 4.24. The first three clients get connected to the Main Server. Afterwards, newly connected clients get redirected to the Proxy Server.

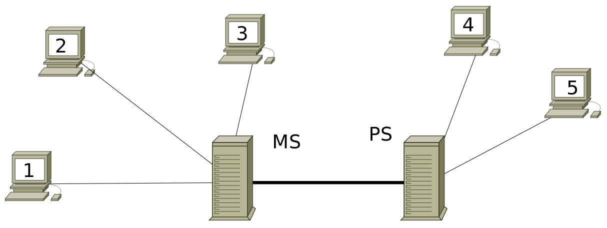


Figure 4.24: Proxy Server Scenario

4.7.3 Measurements

For the following measurements the software Wireshark [19] was used. With this software it was possible to split up the traffic as seen from one node from and to all the other nodes connected to it. In the following graphs the total traffic is in black, the upload from the node is in red and the download is in green.

Old Client-Server Architecture

Laid out here is the measurement of the client server topology as seen from the first client in Figure 4.25, and as seen from the server node in Figure 4.26.

As can easily be seen, the packets on the server side increase lineарely with each joining client.

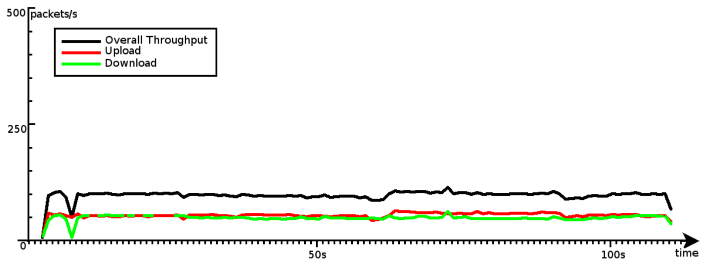


Figure 4.25: Traffic at a Normal Client

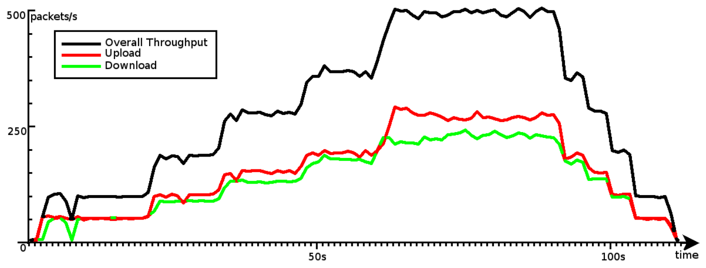


Figure 4.26: Traffic at a Normal Server

New Proxy Server Architecture

In the new proxy server topology the measurements were taken from three different angles. The first is the client's measurement depicted in Figure 4.27, the second from one of the servers (here the master server) in Figure 4.28 and the third shows the interserver communication, filtered out from one of the servers shown in Figure 4.29.

Interesting here is, that the packet throughput does not increase once the client limit on one server is reached and traffic is relayed over one of the proxy servers.

The clients throughput is approximately the same, as only the handshake phase is different, afterwards it talks to the server as if it was a normal client server network.

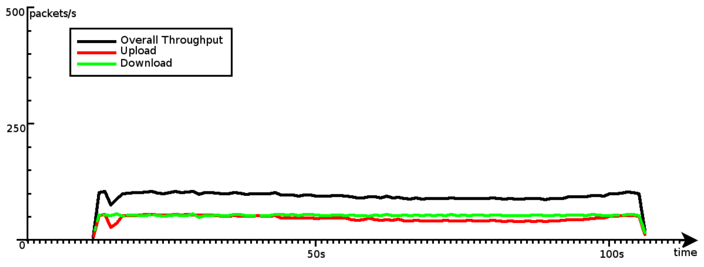


Figure 4.27: Traffic at a Proxy Client

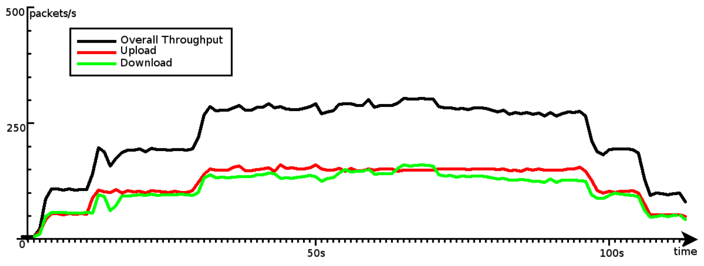


Figure 4.28: Traffic at a Proxy Server

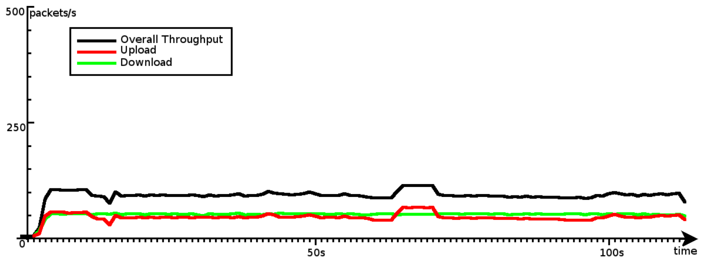


Figure 4.29: Traffic between Proxy Servers

4.7.4 Enhancements

The measurements presented above were not taken completely without problems. For once to take the measurements only two people could move the players, which could have been simulated using an artificial intelligence using the scripting engine.

Also it would have been interesting to measure the traffic in a more saturated network to measure network bandwidth and delay, maybe by using a network simulator. For this to work a network connected by hubs instead of switches would have been more appropriate.

4.8 Conclusions

We have reached the goals, stated in Section 3.2. According to the preliminary tests the implementation proved to work the way we expected. Orxonox now is the first open source game that implements a proxy server architecture based network.
There is still much to do in terms of efficient bandwith usage and more stable network organization.

4.9 Future Work

Evaluation The whole architecture needs to be evaluated correctly.

Better Server Selection Algorithm It would be better to monitor the link quality for a longer time period to be able to choose the best server on a better decision base. The link quality could even be monitored during the game itself and in case of a better alternative should be switched to the another server.

Better Network Shrinkage Threshold Currently the network is reorganized if the client number is beneath a certain threshold. Since there is much overhead involved with a network reorganization it would be better to analyze a network topology and to find out the best moment for a network shrinkage.

Better Network Growth Threshold The *master server* decides when to activate a new *proxy server*. This decision is very critical because if a decision is made wrong, it's possible that a client needs to wait until a *proxy server* is activated. Another reason is that the network always needs to have enough free slots to compensate one server error.

Individual Player Limits Each server should have the ability to specify its own player limit according to its link properties.

Modifying the Proxy Server List during Runtime It should be possible to modify the list of passive *proxy servers* during runtime.

Bibliography

[1] D-ITET, "Department of information technology and electrical engeneering." http://www.ee.ethz.ch.

[2] ETHZ, "Swiss federal institute of technology zurich." http://www.ethz.ch.

[3] Orxonox, "Orxonox - an open source 3d action game." http://www.orxonox.net.

[4] SDL, "Simple directmedia layer." http://www.libsdl.org.

[5] OpenGL, "Open graphics library." http://www.opengl.org.

[6] GPL, "The gnu general public license." http://www.gnu.org/licenses/licenses.html.

[7] Orxonox-Wiki, "Orxonox sevelopment page." https://dev.orxonox.net.

[8] Orxonox-PPS, "Orxonox pps main page." https://dev.orxonox.net/wiki/PPS_main.

[9] UML, "Unified modeling language." http://www.uml.org.

[10] LUA, "A powerful light-weight script programming language." http://www.lua.org.

[11] GLSL, "Open graphics shading language." http://www.opengl.org.

[12] Zlib, "Massively spiffy yet delicately unobtrusive compression library." http://www.zlib.net/.

[13] SDL-NET, "Sdl network library." http://www.libsdl.org/projects/SDL_net.

[14] Doxygen, "A documentation system for c++." http://www.stack.nl/ dimitri/doxygen/.

[15] Jens Müller, Sergei Gorlatch, "GSM: A Game Scalability model from Multiplayer Real-time Games,"

[16] Doom, "One of the first first person shooters from id-software." http://www.idsoftware.com/.

[17] Wu-chang Feng, Francis Chang, Wu-chi Feng, Jonathan Walpole, "Provising online-games: A traffic analysis of a busy counter-strike server," tech. rep., OGI School of Science and Engineering at OHSU.

[18] ISG, "Internet support group." http://www.isg.ee.ethz.ch.

[19] Wireshark, "Wireshark network analyzer." http://www.wireshark.org.

Appendix A

Orxonox Framework Description

Figure A.1.4 gives a general overview over the modules of Orxonox.

In chapter 2 most modules are described in a timedependant fashion. Here the modules are listed as they are logically connected.

A.1 Elements

A.1.1 GameWorld

The *GameWorld* is the main binding block of the API, it connects all the engines together, and as depicted in figure 2.2 handled accordingly.

GameWorlds are StoryElements. StoryElements can be loaded, and the loading process, can be different for each element. If it is a *Campaign*, a sequential order of *GameWorlds* is loaded, if it is a *GameWorld*, *WorldEntities* are created and loaded according to their loading disciple.

Every *GameWorld* holds a list of its allocated entities in its ObjectManager. The ObjectManager is used by many engines (eg. GraphicsEngine, CollisionDetector, etc.) to perform the actions described in section 2.3.1.

A.1.2 Engines

The engines are the driving blocks of the Framework. They all have an interface partner (eg. *EventHandler* has *EventListeners* etc.), that any object can Extend to interface easily with the Engines.

A.1.3 World Entities

WorldEntities are Obejects that can move, interact and draw themselves into the scene. All entities are registered with the *GameWorlds ObjectManager*.

A.1.4 Resources

Resources are objects, that are loaded from a storage media. The *ResourceManger* keeps track of allocated resources, to save memory and loading time.

A.2 Coding Tips

A.2.1 Network

Host Identifiers

It's very dangerous to mess around with *hostIds*, since there are some very special reserved numbers that you will never be able to use. Before working on this make sure to read and fully understand the *network_stream.cc* source file and especialy how handshakes are handled in the *handleUpstream()* and *handleDownstream()* function (handshakes fake a different *hostId* on startup).

Configuration Files

It is very important, that the configuration files on all network servers are synchronized and especially do have the same network player maximum. This is very important because the *hostId* can't be set correctly if the player limit is not synchronized.

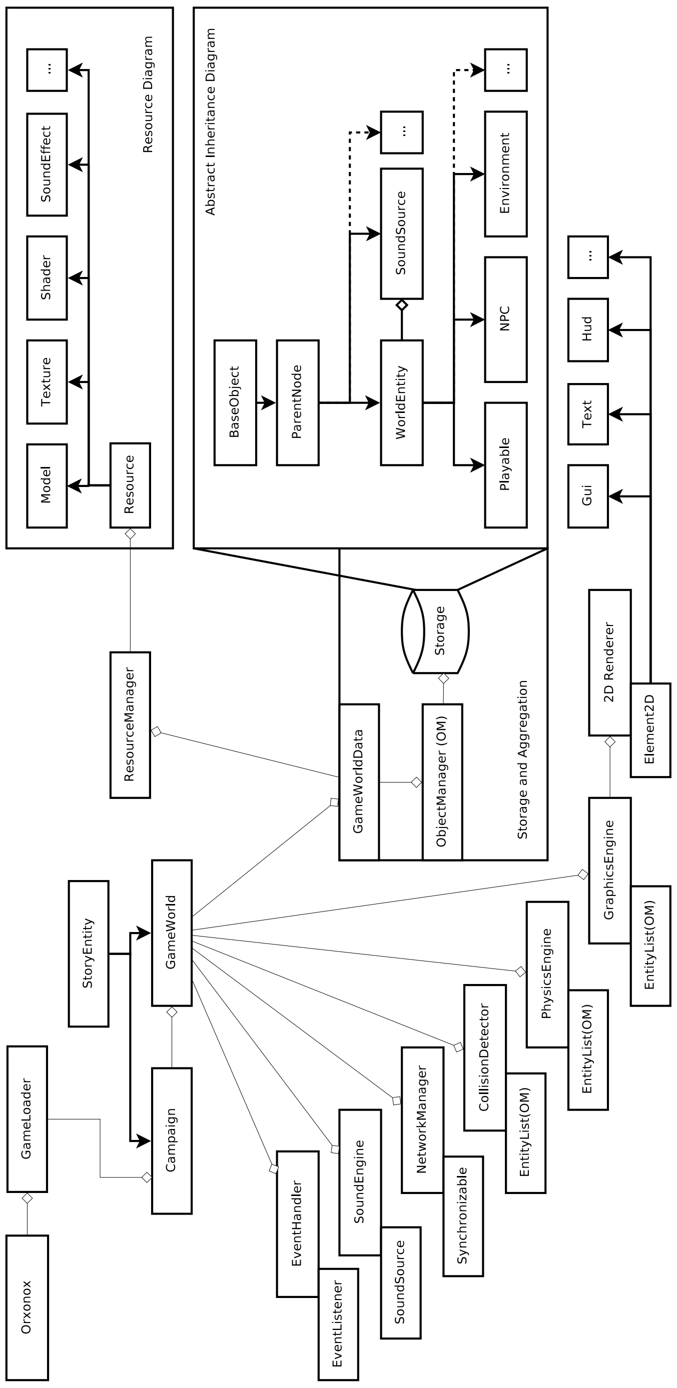


Figure A.1: Orxonox Framework Overview