UNICORN COLLEGE

Katedra informačních technologií

BAKALÁŘSKÁ PRÁCE

Multiplayer mód v počítačových hrách

Autor BP: David Miler

Vedoucí BP: Ing. David Hartman Ph.D.

2015 **Praha**

UNICORN COLLEGE

Department Information Technologies Academic year 2014/2015

ASSIGNMENT OF FINAL BACHELOR THESIS

Name Surname:	David Miler
Study programme:	System engineering and information science
Field of study:	ICT Project Management
Title of bachelor thesis:	Multiplayer mód v počítačových hrách

BACHELOR THESIS GOAL:

The bachelor thesis will cover the topic of multiplayer modes in computer games. The goal is to analyze different approaches in various multiplayer modes in computer games and to describe their respective strengths and weaknesses as well as provide an overview of their architecture and implementation.

The practical section will extend an existing team project codenamed Sublockus. Sublockus is a single-player computer game of the platformer genre played in 2 dimensions including a large set of RPG elements and an automatically generated environment. The source code is written in the C# language, using the .NET framework, especially its XNA libraries.

The concrete aim is to analyze options for implementing a multiplayer mode into this game, design an appropriate solution and implement it. Furthermore, benchmarking of the implemented solution will be designed, run and analyzed.

BACHELOR THESIS PLAN:

Introduction

Theoretical section
1. Overview, setting the goal
2. Description of different approaches and designs of multiplayer
3. Analysis and choosing the appropriate solution for Sublockus
4. Design of the selected solution.

Practical section
1. Documentation of the implemented solution
2. Benchmarking and its analysis
3. Analysis and possible solutions to other problems emergent from the implemented solution.

Conclusion
Appendices(source codes, license)

RECOMMENDED LITERATURE:

1. Smed, Jouni - Hakonen, Harri: Algorithms and Networking for Computer Games. Wiley, New York 2006. ISBN 978-0470018125
2. Cawood,Stephen - McGee,Pat: XNA Game Studio, Creator's Guide . McGraw Hill, New York 2009. ISBN 978-0071614061

Bachelor thesis consultant:	David Hartman
Place of work:	V Kapslovně 2767/2, 130 00 Praha 3
Entering date:	21.07.2014
Thesis deadline:	According to rector's decision

Prague, date 10.07.2015

[illegible]

doc. Ing. Jan Čadil, Ph.D.
rector

Unicorn College
UNICORN COLLEGE
V Kapslovně 2767/2, 130 00 Praha 3

Čestné prohlášení

Prohlašuji, že jsem svou bakalářskou práci na téma Multiplayer mód v počítačových hrách vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím výhradně odborné literatury a dalších informačních zdrojů, které jsou v práci citovány a jsou také uvedeny v seznamu literatury a použitých zdrojů.
Jako autor této bakalářské práce dále prohlašuji, že v souvislosti s jejím vytvořením jsem neporušil autorská práva třetích osob a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb.

V……………………. dne ………..   ……………………………………

(David Miler)

Poděkování

Chtěl bych poděkovat Ing. Davidu Hartmanovi Ph.D. za vedení mé bakalářské práce, cenné rady a odborný dohled. Děkuji také celému týmu Velvet Studios – Romanu Jergonovi, Pavlu Pokornému, Ondřeji Pluskalovi a Zuzaně Šebestové, bez jejichž nasazení a nadšení, které vložili do projektu Sublockus, by tato práce nevznikla.

Multiplayer mód v počítačových hrách
Multiplayer mode in computer games

Abstrakt

V této práci prozkoumáváme specifické problémy spojené s implementací online počítačových her pro více hráčů a snažíme se poskytnout přehledný a ucelený pohled na tuto komplexní oblast. Na základě studia odborné literatury vytváříme katalog jednotlivých žánrů her více hráčů a popisujeme specifické postupy, kterými se vyrovnávají s těmito problémy. Ve druhé části popisujeme návrh a implementaci multiplayer módu v naší vlastní hře. Vytvořili jsme a zdokumentovali knihovnu pro síťovou komunikaci, kterou jsme využili v této hře. Tato knihovna využívá open-source kinhovnu Lidgren a poskytuje jednoduché rozhraní pro vytvoření klient-server architektury a systém pro posílání libovolných zpráv mezi propojenými aplikacemi. V poslední části jsme experimentálně ukázali, že využití kombinace spolehlivě a nespolehlivě doručovaných zpráv vede ke zvýšení výkonu rychlých online her více hráčů.

Klíčová slova: multiplayer, počítačová hra, Lidgren

Abstract

In this work we explore the many challenges involved in implementing an online multiplayer computer game and provide a holistic perspective of this complex field. Based on the study of existing literature we catalogue different multiplayer game genres and describe the unique approaches they use to deal with these challenges. In the second section we discuss in detail the design and implementation of multiplayer mode in a platformer game of our own making. We build and describe a networking library used in this game that provides a simple interface to establishing a client-server architecture and a system to send custom messages between connected applications easily. This library utilizes the open-source Lidgren networking library. In the last section we demonstrate how using a combination of reliable and unreliable methods of message delivery is beneficial to the performance of fast-paced online multiplayer games.

Keywords: multiplayer, computer game, Lidgren

Table of Contents

Introduction

Video games have in the past 30 years established themselves as a progressive branch of the entertainment industry. With the advent of the Internet and the online multiplayer aspects of gaming, the video game market has quickly risen to compete with other more established media and art forms.

The topic of multiplayer modes in computer games has a great appeal to us, personally, as we have been working on a computer game project, called Sublockus, since 2011. Sublockus is a 2D platformer game and more about it and its creators, a small group of enthusiasts called the Velvet Studios, can be found in chapter 2.1. In this thesis we build upon our previous work and extend the project to encompass a multiplayer mode. The multiplayer mode allows players to fight each other in an enclosed environment while using various skills and abilities. With the addition of multiplayer mode we hope that the final game will be significantly more attractive to players and thus better able to find its footing in the very competitive computer games market.

In this thesis we implement a multiplayer mode using a popular open-source library called Lidgren upon which we build our custom networking framework. As a result, we are not focusing on the technical details of TCP/IP but rather on the aspects of networking on the application level.

One general issue we have to keep in mind while studying this topic is that it is mainly driven by private game development studios that do not readily reveal the details of the implementations of their games. As a result, there is a limited amount of sources that cover the subject of developing multiplayer games. Most of the publications that exist usually narrowly focus on individual topics or describe the use of specific frameworks. One of the aims of this work is to provide a general overview of as many aspects of online multiplayer game development as possible and also to provide an example implementation of a simple networking framework.

The general aspects of online game development are discussed in section 1. In this section we provide an overview of the underlying challenges in multiplayer game development and then give a list of examples how these challenges are overcome in different multiplayer game genres. Especially, we focus on the ways latency is masked in fast-paced games.

The bulk of this work is covered in section 2 which describes the inner workings of Sublockus multiplayer mode with focus on the interface of our networking framework. We describe the architecture of the framework and the messaging system we use in Sublockus multiplayer mode.

In section 3 we conduct a series of measurements that explore the effects of network latency and packet loss on different message delivery protocols. Specifically, we seek to prove that a combination of both reliable and unreliable message delivery protocols is desirable for a typical game-generated traffic.

As a whole, we hope this work will serve as a valuable guide to anyone starting out with online multiplayer game programming. While far from being a definitive source, it provides the necessary overview of the topic, with the bibliography section pointing towards numerous useful sources that provide further insight into the various aspects of online games development.

1 General principles – theoretical preparation

In the first chapter we will explore and discuss the options available for designing and implementing a multiplayer game. We will describe different multiplayer game categories and the techniques they use to overcome their respective challenges. We shall try to provide a broad perspective with particular attention to our own solution implementing multiplayer into Sublockus. Therefore we will occasionally refer to certain C# and XNA specific technologies, however, similar approaches are generally available for other platforms as well.

1.1 The main issues

According to Ernest Adams, "online gaming is a technology rather than a genre, a mechanism for connecting players together rather than a particular pattern of gameplay." (1) This technology's main goal is to enable multiple players using different computers to play in an identical and shared game-world where they can interact, compete and co-operate. This is done by replicating and synchronizing the game state across multiple computers and thus creating an illusion of a single shared game space.
The chief concern of online game development is to relay the data from one computer to another. For a truly synchronized and identical game-state across multiple remote computers that dynamically changes in real-time we would require instant and infallible data transfer technology. Sadly, due to laws of physics and Internet infrastructure, we have neither of those. Therefore the second and far more complex concern of online game development is to cover up for these limitations.
There are three main issues we have to keep in mind when designing and programming online games: latency, jitter and packet loss.

1.1.1 Latency

Latency refers to the time it takes for a packet of data to be transported from its source to its destination. In many networking texts, you will also see the term Round Trip Time (RTT) in reference to the latency of a round trip from source to destination and then back to source again. In many cases the RTT is twice the latency, although this is not universally true (some network paths exhibit asymmetric latencies, with higher latencies in one direction than the other). In online gaming communities, the term *lag* is often colloquially used to mean RTT. (2 p. 69)

1.1.2 Jitter

Variation in latency from one packet to the next is referred to as jitter. There are many different ways to define jitter as a function of latency, for our purposes it is sufficient to know that latency can fluctuate slowly or rapidly from one packet to next. Jitter is of great concern to game developers, as it greatly affects the gameplay experience. Empirical studies such as (3 p. 174) show that players tolerate even moderately high levels of latency, however, the sudden changes associated with jitter are a great annoyance, as is also referred to in (2 p. 94).

1.1.3 Packet loss

Packet loss refers to the case when a packet never reaches its destination. It is lost somewhere in the network. Depending on the transport-layer protocol used it will either be sent again (causing a spike in latency) or lost forever. It is critical that the developer identifies when packets need to arrive at their destination unconditionally and when a lost packet is acceptable. (4 p. 492) In future chapters we shall see different protocols, both on the transport layer and the application layer that either require or ensure that packets are always delivered to their destination (or break the connection if it is impossible to deliver). In other scenarios (such as a position update of a unit) one packet gone missing is no big deal as the next packet will arrive with newer information that overrides the last one.

1.1.4 Debugging and testing

Another specific issue associated with online game development are the increased demands on debugging and testing. Since the conditions in network may vary greatly both spatially and temporally, many errors may manifest only in certain very specific scenarios. This accentuates the need for a thorough, systematic design, careful test planning and the ability to simulate different network behaviours.

1.2 Transport layer protocols to use

One of the main technical decisions that can have a great impact on the final product is the choice of the transport layer protocol. Unfortunately, the drawbacks of choosing the wrong protocol for a given task typically manifest far down the road once the application is under heavy load or users are connected via high-latency connections. Basically there are three options to build upon: TCP, UDP and a combination of both.

TCP is the more robust of the two, it provides reliable delivery of packets and also guarantees that they arrive in the same order they were sent. This is generally a good thing, as we can clearly demonstrate that losing packets or getting them in the incorrect order could easily disrupt the gameplay (e.g. a player pushes a button to make an action, but no response is given because the packet was loss before arriving to the server, or, even worse, it was accepted and resolved by the server but lost on the way back causing the server and client state to be de-synchronized). However these good properties come at a cost of potentially higher latencies, caused by larger headers or resending lost packets. A very detailed analysis of the use of TCP in online games is given by Chen et al. (5)

UDP, on the other hand, does guarantee none of the above. If a packet is lost along the way, it is lost for good. The good part, however, is that there is very little overhead and one lost or delayed packet does not in any way affect other packets, therefore the average latency for UDP packets is smaller. We can use this for our benefit if we realize that a large portion of the data sent in a multiplayer online game (especially in some genres as we will see in following chapters) is relevant only for a small period of time and is soon rendered obsolete by a following update. For example the updates sent about the movement of a player are relevant only until the next update arrives with more up-to date data. Therefore it makes little sense to require that packets arrive in an orderly fashion.

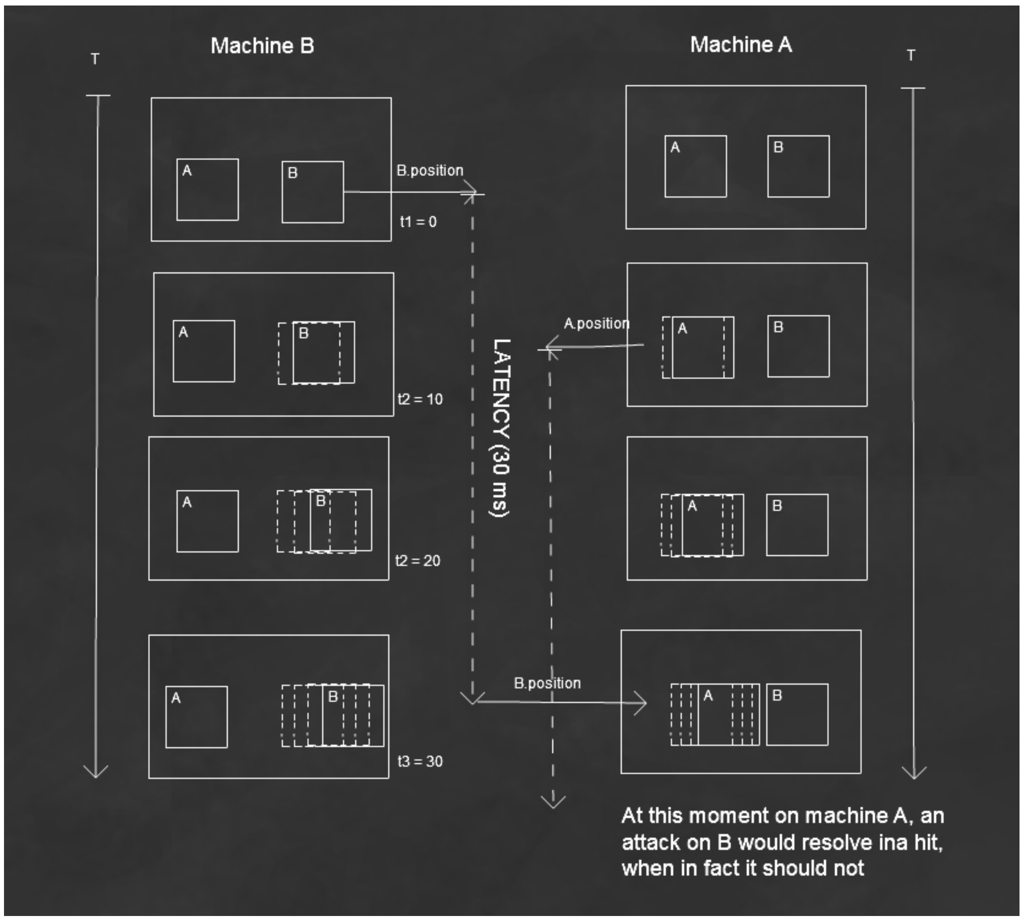
Still, there are other types of updates, for example about attack moves, that we cannot allow to be lost, therefore a seemingly sensible solution might be to use a combination of both UDP and TCP. It is even possible to open multiple TCP connections and send different types of data over separate connections, so that unrelated updates do not have to wait on one another. However, as Glenn Fiedler explains (6) the UDP packets are adversely affected by concurrent TCP communication, which makes this quite inefficient.

A better solution, as he proposes, is to implement the reliability and orderliness features of TCP on the application layer using UDP as that gives us a much better control over the communication and bypasses some of the less desirable features of TCP. As we shall see in section 2, this is the approach we have opted for in our game.

1.3 Messaging and possible conflicts

Let us take a high-level look at the sort of information (further called messages) that is sent between connected instances of a multiplayer game.
There are in essence at least two different levels of data messages. The first level, represented chiefly by position information, is information that can be somewhat inaccurate and still allow for unhindered gameplay. That is, for the sake of gameplay we do not need to have each instance rendering each player´s avatar on exactly the same spot to the precision of individual pixels. Basically as long as the data message is used to only show a certain state to the player, i.e. is "read-only", we can tolerate the lag that comes with networked communication.
On the other hand, we have information where no difference can reasonably be tolerated among connected instances. This demand logically arises from the need for synchronized simulation of the game state across multiple instances. This category contains all other messages that cannot be proven to fit into the first category, where differences can be tolerated. For example whether a player hits another player cannot be resolved differently on different machines, since it can lead to each player showing different health or even one machine showing a dead player, while the other having a player alive. This surely leads to disaster. Henceforth we see that we need some sort of authority that decides whether a player has been hit.
You might ask how it is possible that one machine would resolve that a player has hit another player and the other machine might resolve that the attack has been a near miss instead. For example if we do not synchronize player positions on every frame it is easy to imagine that when one machine thinks a player is in position A and the other machine believes him to be in $A + (x, y)$, where (x, y) is an arbitrary vector, the hit resolution might end up differently. There are several reasons why (x, y) might be different from (0,0) and they are covered in the following chapters.
In Figure 1 we can see how a conflict occurs as a result of non-zero latency.

Figure 1: Effect of latency on colision resolution



Source: own creation

1.3.1 High latency

The main reason behind latency is that while each instance tries to run the same simulation, it sends its input information on the same frame as it uses this input information in the simulation and renders it for the player. That means that this information will arrive to the other machine with some latency, at a moment when that other machine may already be resolving the next frame. In fact, when we work with long-distance connections, the latency will probably span the time of many frames rendered on the distant machine.

1.3.2 Unstable sampling rate

Even if we through some magic managed to have accurate and recent (position) data from the remote machine, any difference in the sample rate can lead to different simulation outputs. While we can setup each instance to run at the same set sample rate (e.g. 60 ticks per second) we can never reasonably ensure that we will have enough computing resources to maintain this rate for the whole run of the simulation. Once the CPU is incapable of running so many samples, it will increase the time between samples and this greater granularity can easily lead to different simulation outputs as seen on the following image.

1.3.3 Misprediction

Thirdly, when we try to make up for the latency with any sort of prediction, we can never be sure that the position we predicted for remote players is correct until we receive a message from the given frame from the remote machine that confirms or corrects our prediction.

1.3.4 The need for authority

Thus we see that if we let each connected instance to resolve these critical events on its own, we would soon end up with an asynchronous mess. Therefore we need some sort of authority that decides whether such critical event has happened (or how and when) or not.

There are multiple strategies to implement this authority, but the most obvious is an authoritative server. Such server runs the whole simulation with the same inputs as all connected clients and if the simulation differs between the server and the client, we assume that the server is “right” and its results must be respected by the clients. That at least is the high-level logic, implementation details will be left out for now and we shall look at them in more detail in chapter 1.7.

1.4 Network topologies

In this chapter we will describe different network topologies and see which one is the best fit for our game and why.

1.4.1 Peer-to-peer

We have already established that some sort of conflict resolution will be necessary. If we choose our topology to be peer-to-peer that is each instance has a connection to each other and messages are sent along each of these connections, each communicating pair can have their own conflict that needs to be resolved. Moreover, the resolution needs to be communicated to every other peer (who can also be in conflict with this resolution). We can clearly see that conflict resolution in such a system is very complicated and would basically require some sort of central co-ordination, which defeats the purpose of the peer-to-peer topology.

Another strong argument against peer-to-peer is the scaling of network communication. Since every peer needs to update every other peer, there is linear rise in the use of bandwidth for every peer that joins the game. With a client-server architecture, it is only the server that has to deal with this increase and each client is using a constant bandwidth no matter how many clients are connected (not taking into account that more data might be sent because the game state expands as a result of adding the extra players).

Generally speaking, a client-server model is the simpler of the two despite the intuition that writing two different applications (or modes of operation within the same application) would be more difficult than writing one piece of code for every peer[1].

1.4.2 Client-server

Therefore we have opted for a client-server topology, as we shall see in section 2, where the main role of the server is to arbitrate potential conflicts among the server and clients and among different clients. For our small game where only a handful of players are expected to be playing in a single game, the bandwidth issues are not crucial.

In low latency scenarios it might be feasible to implement a system where player input is resolved (e.g. player´s avatar moves as a result of a press of a button) only once it reaches the server and the server resolves it authoritatively. However, using long-distance connections, where one has to deal with latencies over 100ms, this is becoming problematic, and outright useless for very fast-paced games. This so-called “command-lag” (as the player perceives a lag before his command is executed) is very annoying to the player and is present in all situations, not only when resolving conflicts as it would when utilizing other approaches. However, certain game genres may cope with this sort of delay rather gracefully and this defensive approach may be a valid option, as we shall see in chapter 1.8.2.

[1] Further throughout this work we shall use the term peer to denote either client or server, when we do not mean to differentiate between the two.

1.5 Gameplay session initialization

One of the areas that need to be discussed is the way we initialize individual multiplayer sessions. This includes giving the player a possibility to choose whether he wants to join a game or start a new one, which game to join etc. We shall focus on the technical details. Peer/Server discovery is done differently on LAN and over the internet, where the initiating peer or client has to query a concrete IP address directly. This has a bearing on how the UI for discovering a running session one can join works.

On LAN peer discovery can be done using broadcast which is a simple way to ask every node in the network whether it is a server you could connect to. However, this mode of communication is not supported by WAN (you cannot reasonably ask every computer on the internet), therefore such peer discovery has to be based upon its own system, where a central server gathers information about available peers and then conveys answers to clients looking for servers to join. In this system, a server that is being started registers with the master server and unregisters when it is being shut down. This system does not use broadcast and relies on unicast communication only.

Orthogonal to the issue of server discovery is the way a client joins a gameplay session. This could be either through a multiplayer lobby or directly using a join-in-progress system.

A multiplayer lobby is a virtual space where all peers participating in a game session gather before the session itself starts. This approach is utilized in some game genres, such as real-time strategies or racing games where the game rules cannot sensibly allow a player to join in once the game has begun. This has a bearing on how the information about remote players and the initial state of the game state is distributed. Since these information can be gathered before the game starts, each peer then initializes the game state in the same manner and all changes to the game state once the game has begun can be communicated incrementally. It also allows for a smoother start of the game, as all peers can initialize the game state and wait for any stragglers, and only once every peer has performed all the CPU intensive initialization, the game itself can start.

On the other hand, a join-in-progress system allows a peer to join the game once it has begun. This requires that the connecting peer receives the current game state at the moment it is connecting. Typically the game state is sent by an authoritative server, which creates a snapshot of its current state and sends it to the connecting client. A potential challenge here is that the server state can be fairly large and it can take quite some time for the client to initialize. At the same time, however, the server has to keep the connecting client updated with the incremental changes occurring in its game state. It might be beneficial for the gameplay experience of the connecting player to grant him a grace period in which he cannot be affected by the game (e.g. killed by other players) while his client is still loading the game. Once the client has loaded and initialized the whole game state it can send a message for this grace period to end.

The join-in-progress is typically used by first-person shooters or MMOs, where it is possible to add players to the game state dynamically. Another benefit of this system is that it does not require the concept of game lobby, which simplifies the UI and player experience.

1.6 Synchronization strategies

In this chapter we shall take a look at how we can keep the game state of the clients synchronized with that of the server.

Command input synchronization is one of the options that was presented by Jan Svarovsky (7). The basic concept behind this strategy is that each players commands (i.e. their interaction with the game) is sent to each other peer and it is then resolved identically. This is a very elegant solution that minimizes the communication required between peers, however it puts very strict limitations on the rest of the architecture. Most of all it requires that each peer updates the game-state at exactly the same rate, with each command reaching the other peers before its execution. This means that the game is not going to be very responsive and is therefore not a very good option for fast-paced games, such as Sublockus. However, for certain game genres, as we will see in chapter 1.8.3, it is the correct choice.

Another case against this strategy is that while it enables us to have the same code governing the local and remote player entities, it forces us to recreate the remote player's UI. If we synchronize just the mouse and keyboard clicks of each player, we have to know how his UI (inventory, action bars, screen size) looks so that we can evaluate these input commands properly. This might not be an issue if we have no or very simplistic UI, however, for Sublockus this seems like a fairly poor choice.

Therefore we have to synchronize the commands on a more concrete level, after they have been interpreted by the client that is receiving these input commands. To illustrate,

instead of synchronizing the push of button X, we synchronize the “desire to cast a fireball”, which is the result of the pushed button X. That means the client has already interpreted the button click and already knows what it means in term of in-game entities. In more detail we might actually synchronize a fireball projectile entity with a certain position, direction and speed.
The alternative to command input synchronization is the entity synchronization, which synchronizes the attributes of actual in-game entities. This approach does not require us to have all clients run at the same sampling rate. In this case, the server is distributing the results of player actions and only the server interprets the commands. Therefore we can allow for minor discrepancies in the client states, as they can be corrected by the information sent by the server. For example if two clients calculate the position of an entity differently (e.g. because each uses a different sample rate), once an update of this entity arrives from the server these differences are removed.

1.7 Conflict resolution

One thing that we have to specify when designing any real-time online application is the way conflicts are being resolved. As we have demonstrated in chapter 1.3 conflicts arise when the state on a client is different from that on the server and as a result allows for a different outcome of player actions. A straightforward option is to use our own client state only once it has been conveyed to the server and confirmed by it. Then the client can run exactly the same simulation as the server, albeit slightly delayed. The main and crucial drawback of this approach is that every action taken by the player is rendered with a delay that is equal to the round-trip latency between the client and the server. As empirical evidence (8) (9) shows, such sort of delay when in the ranges of 100ms is a great nuisance to the players and becomes intolerable with higher latencies very quickly.
Therefore we need to consider ways to increase the responsiveness of the system. A possible solution is to have player movement instant (rendered on the client before it is confirmed by the server) and handle only some critical-level[2] actions need server confirmation, which makes the overall player experience significantly better. However, this then creates some discrepancies between what the player sees in his own version of

[2] We define critical-level actions as those that can in some way affect other players. For example attacking is a critical-level action, while moving is not (as long as players do not collide with one another).

the simulation on the client and what is actually run in the authoritative version of the simulation on the server. It is still a fairly easily implemented solution, which does not have to make any predictions we would have to roll back.
To achieve an even greater level of responsiveness we can go one step further and render even the critical-level actions to the player outright. However, since only the server knows the correct outcome of these actions, we are merely giving a prediction.
Let us assume we make the simplest prediction where we take the latest confirmed state (that we have received in a data message) and make the optimistic assumption, that the results of our simulations are most probably correct and therefore we will render the results to the player outright. Of course, we still have to account for the chance that our prediction is wrong. To that end, whenever we take a critical-level action (such as attacking the remote player) we send a message with the parameters of the action to the server and the server will decide what the outcome of that action is according to the server state of the simulation. It will then send back a confirmation message or a correction message.
If we receive back a confirmation message, everything is right and we can continue with our simulation happily. If we receive a "correction message", however, we will have to change the game state according to the server version of the simulation. Here we can expect jittery behaviour and various graphical glitches in our game. The execution of the correction message can be fairly complex, depending on the nature of the player action. We shall see this approach in a specific algorithm in chapter 1.8.4.

1.8 Multiplayer mode categorization

Having described the principal challenges inherent to online multiplayer games in the previous chapters, we shall now take a look at the differences found across a number of multiplayer games. Based on the study of a number of publications (10) (2) (11) (12) (13) (3) we have come up with a categorization of games, or rather game genres, with respect to their multiplayer implementation. Through assembling various techniques and frameworks we present several basic approaches to multiplayer mode in computer games. They vary mostly in their way of handling high latency, high volumes of data and server scaling. This is not a strict division, as some of the techniques are applicable to many (if not all) genres, but rather a general overview of the specific characteristics and challenges of each category that pertain to their multiplayer implementation. These

categories are as follows: hot-seat, turn-based games, real-time strategies (RTS), fast-paced games such as first-person shooters (FPS) and massively multiplayer games (MMO).
We see the usefulness of such categorization especially in that a developer can identify the correct category for the game they are creating and then look for existing solutions in that category. We shall focus mostly on describing the techniques used in RTS and FPS games. The first two cases are somewhat trivial, while MMOs are a greatly complex subject that is beyond the scope of this work.

1.8.1 Hot-seat

Hot-seat is a colloquial term for a multiplayer game played on a single machine, usually using multiple input devices and a shared or divided screen. While it is played by multiple players, it is not using a computer network and is technologically similar to a traditional single-player game. These types of game were mostly prevalent in the past when broadband internet connection was not commonly available and to some extent survives on home consoles in certain niche game genres. We will not focus on hot-seat games in this work.
Examples: Super Smash Bros. (Nintendo), Bomberman (Konami), Heroes III (3DO)

1.8.2 Turn-based games

Turn based games require the least sophisticated approach of all, as gameplay is generally very permissive of delays caused by network latency. In large strategies (such as Sid Meyer's Civilization) the focus would therefore be mostly on reducing the bandwidth used during initial synchronization (2 p. 96). Usually in turn-based games, only one player at a time is allowed to make any changes to the game-state and this initiative is passed only once all changes have been synchronized. For strategies in general it is unsuitable to synchronize all in-game entities all the time, rather each player's moves (orders) are synchronized and then interpreted identically on each machine.
Examples: Civilization V (Firaxis Games), Hearthstone (Blizzard), Chessmaster (Ubisoft)

1.8.3 Real-time strategies

The main difficulty with RTS games is the large number of in-game objects that make up the game-state, which need to be synchronized across multiple peers. As described in an article (14) by the creators of the famous strategy game Age of Empires (AoE), sending even just position information for thousands of units uses way too much bandwidth (this was especially true for old dial-up connections, but is still relevant today). Therefore the approach used is to run the same game simulation on every client, and only send control information (i.e. commands issued by players). These commands need to be interpreted by an authoritative server. A simple protocol that tackles these issues is called the lock-step protocol. Jan Svarovsky presents the following description of the lock-step protocol.

"As the name implies, a game using this protocol runs in lockstep with all computers involved. They start with the same games state, and game time is split into turns. With each turn, each computer models the game world to advance the action by one step. At the end of the turn, each machine tells every other machine what actions the user has performed. Since all of the computers know what all users are doing, they can collectively model the next game turn and stay synchronized." (7 p. 497)

As Svarovsky notes this protocol is easy to implement and requires low bandwidth that is unrelated to the size of the game-state. What is difficult is the task of keeping all machines synchronized, as any direct manipulation of the game-state (without the server first receiving the input) will probably result in an out-of-sync scenario. Also any randomly generated numbers need to be generated as a single sequence with a common seed. This requires a planned-ahead architecture and strict programming and code-review practices. (7) (14)
The most severe drawback, however, is that the number of game turns per second is limited by latency, since advancing the game turn is possible only once the information from all sites has arrived.

Improving the lockstep

One of the workarounds to this was used by AoE and that is separating *communication turns* from actual rendering frames and then adding a shift in the *communication turns*

so that they are executed 2 steps from the past. In AoE, when a command is issued, it is sent to the server, while at the same time a command from two turns back is being rendered on the screen. This creates a seamless stream of actions, but still creates a delay in execution of commands, however, for usual latencies (up to 500ms) this was found to be acceptable for this not-so-fast-paced genre. (14). This delay is called local-lag and is in detail analyzed by Mauve at al. in (12) and (15).
Furthermore, they created a *Speed Control* system that dynamically adapted the length of a single communication turn. The length of the turn was always determined by the slowest computer, so that it had enough time to process the simulation turn, render it and receive the inputs for the next turn (if other machines ran at the same pace without waiting for this slowest machine it would soon get out of sync and disconnect).
Examples: Age of Empires (Ensemble Studios), Starcraft (Blizzard), SimCity (Maxis)

1.8.4 Fast-paced games

In this chapter we will focus on FPS games and other fast-paced genres (Sports, Racing, Action, etc.) which is a group that contains most multiplayer games. Their main common characteristic is that they rely on very fast player reactions and the action on the screen is progressing rapidly. For example, in a fast-paced shooter game, the time it takes a player to move from one place to another (where the space their avatar occupies is disjoint from the space previously occupied) can easily be in the order of hundreds or even tens of milliseconds. It is then obviously critical, that the representation on each client machine has to be as accurate as possible, otherwise shooting a moving target becomes nearly impossible. There are three techniques that we shall describe in this chapter, each dealing with a slightly different issue, but all of them are most relevant to fast-paced games.

Dead-reckoning

The first technique falls under the broader category of client-side prediction - that is speculating on the correct state of the simulation without having received actual updates from the server. This technique is called dead-reckoning and it is used to calculate the position of a player (or any player-controlled entity) in-between updates from the server. This allows us to send updates less frequently and save bandwith, but also to predict the position of the oponent as it is represented on the server (2 p. 89). Dead-reckoning, which is inspired by the way sailors used to calculate their position, uses position and velocity to predict the current position of the remote player. That is we acknowledge the latency between clients and based on this latency we calculate a correction for the position that we receive in a data message. When P_{ct} is the corrected position at time t we use in our client-side simulation, P_0 is the position we have received in a data message, V is velocity. L is the latency between the server and the client and Δt the time passed since t_0.

$$P_{ct} = P_0 + V \times (L + \Delta t)$$

However, using this corrected position to render the remote player will lead to errors and jerky movement when we correct our mispredictions (3 p. 194). A possible strategy to alleviate this is to use interpolation and gradually converge to the correct position. However, for many scenarios it might be more desirable to correct the position as soon as possible and avoid the added imprecision and complexity of the gradual convergence. A more detailed description of dead-reckoning and prediction techniques is given by Armitage et al. in (2 pp. 87-93)

Time delay

Time delay, as discussed in (2 p. 93) and (12) is a technique aimed at creating a more level playfield for players with differing latency. In fast-paced games having the correct game-state information a fraction of a second before your opponents can make a huge difference. Imagine a typical FPS scenario where you emerge from behind a corner, spot an opponent and shoot them. If that opponent is further away from the server (in terms of latency) they will see you later than you might see them and in an extreme case they could be dead before even having a chance to react.

Since we cannot easily ammend the high latency of the remote player, we establish justice by adding extra delay to players with higher latency. That means sending update information to closer players later than to far away players, so that they get the same updates at roughly the same point in time. This effectively means that everyone is playing with the latency of the farthest player, so we are trading responsiveness for fairness.

The updates can be buffered either on the client or on the server, which is generally preferable since it avoids the possibility for cheating by modifying the client-side program and getting to the information that should be delayed. Another point to note is that the delay should not be re-evaluated very often as that creates extra variation in the percieved latency, i.e. jitter.

Time Warp

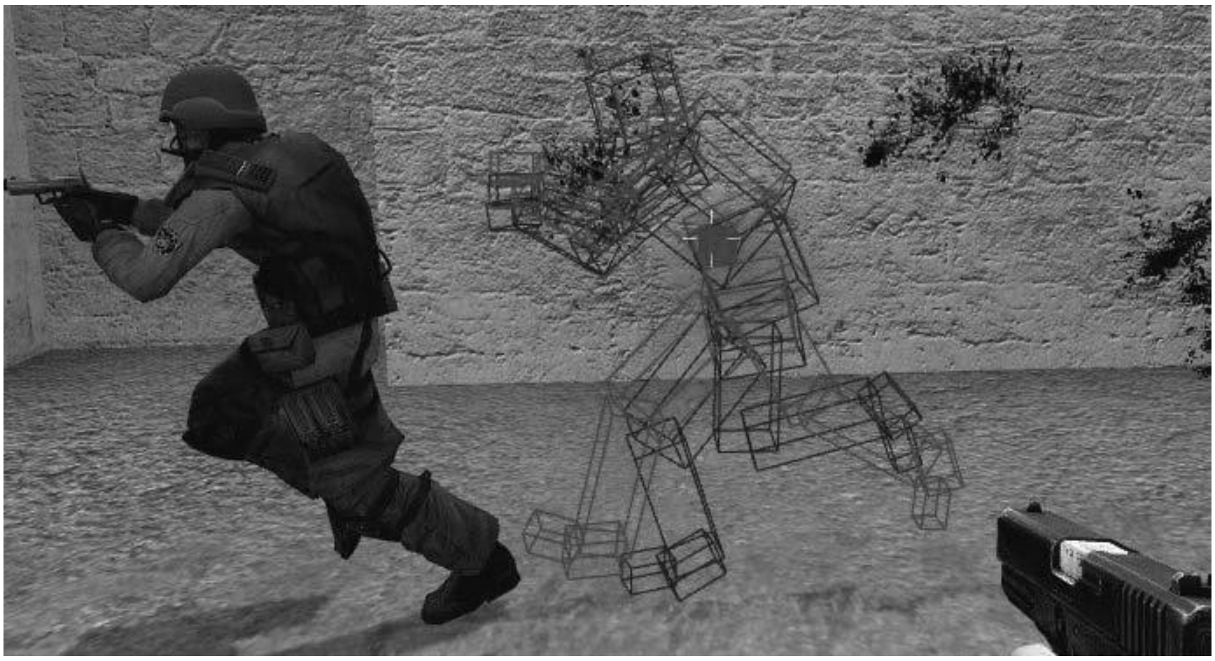
Time warp is the most complex of theses techniques, but probably also one of the most powerful at ensuring that players are given a fair, consistent and responsive experience. It has been first introduced by Martin Mauve in (16) and since then successfully implemented in games such as Valve's Half-life 2 (2 p. 95). Players provide input based on the actual state of the game at the client. Because of the lag between the client input and the server receiving the command, the state at the server may have changed. For example, player A shoots at player B at time t_0. By the time the shoot command arrives at the server at time t_2, player B had moved at time t_1 so he is no longer in the spot where player A is shooting. With the time warp algorithm, the server rolls back all events it had processed since t_0, when the client had sent the command. The server then re-evaluates all events including the shoot action from player A. This can easily lead to a different outcome. In this case, player B, who might have already escaped behind an obstacle, will be shot and killed. The bullets from player A turn around the corner and hit player B, as they should, since player B was shooting accurately at t_0. We can see that the actual unfairness has shifted from the player with high-latency (who would otherwise have to lead his fire to hit a running opponent) to a perceived unfairness on the side of the player who is being shot at (no matter their latency). However, for FPS games in particular this is a very good trade-off as this unfairness is perceivable only in a slight minority of cases, as opposed to the scenario without time warp.
The server-side algorithm as presented by Armitage et al. (2 p. 95) is as follows:

1. Receive packet from client
2. Extract information (user input)
3. Elapsed time = current time – latency to client
4. Rollback all events in reverse order to current time – elapsed time
5. Execute user command
6. Repeat all events in order, updating any clients affected
7. Repeat.

Important to note is that it is crucial that we measure the latency between the client and server precisely and as frequently as possible. It is also highly advisable that the latency calculation is done by the server and not the client, as that opens up doors for extremely potent exploits. A cheating player can lie to the server about the latency they are experiencing, thus tricking the server to rolling his actions further back in time than would be appropriate. That allows the cheater on the client-side to virtually see the future and react to it in advance.

For illustration we add a screenshot from Half-life 2's debugging mode, which shows the location of a player both at their current location and the location seen by another remote client firing at that player.

Figure 2: Example of Time Warp (Half-Life 2).



The target is on the left and is ahead (in terms of time) of the client. The boxes on the right represent the targets the client had when shooting (back in time) and that the server uses when time is warped back to determine a hit. **Source:** (2)

It is important to note that time warp is fairly complicated to implement (we need to add the rollback mechanism to every event and the ability to fast-forward again after resolving a command from the past) and also quite demanding on the server. The computational complexity of time warp is at least $O(Ln^2)$ where L is the average latency and n the number of players. A more detailed discussion of the computational complexity of time warp, as well as a more rigorous definition of the algorithm is given by Martin Mauve in (12).

1.8.5 Massively multiplayer games

Massively multiplayer online games (MMOs) are the most complex multiplayer games and we will not discuss them in depth in this work. MMOs host upwards of a thousand of players in a single instance of the game and most often offer a persistent world, where player characters and their progress are preserved between play sessions. The persistence is achieved by the use of a database, which brings the MMO server architecture closer to traditional 3-layered web application servers. The scaling aspect is accomplished by several different techniques. One of the ways to keep communication in check is using spatial indexing. That is, when sending messages from the server to a player, only updates on things that are in his close proximity are sent. Even though there are thousands of players in a single connected world, they do not usually interact in any way at long distance. Therefore we can save orders of magnitude of traffic by updating every player only with what is going on in an area around them (13). Another technique is using server clusters and splitting the game world, along with the players, among multiple self-contained servers. It is important that each server takes care of a geographically consistent area, so that there is minimal communication between servers. For example, if a game world consists of two continents, each continent would be serviced by a separate server. Even though players can take a ship from one continent to the other, which makes one server transfer control over the player to the other server, they cannot interact directly. (13) Further improvements on this architecture include dynamic server balancing, as described by Daniel Sanchez-Crespo in (13). Another key aspect of enabling the massive scaling is using simplified simulation on the server. In his article *A Flexible Simulation Architecture for Massively Multiplayer Games* (17) Alexandre Thor describes a framework where the server runs only the minimal necessary simulation and lets each client run the computation-intense physics and graphics simulations that are not required to keep a consistent game state. It is important to note that this approach has its limitations, as you can never trust the game client to run any critical calculations as that would open the floodgates to tampering with the client program and cheating.

1.8.6 Closing remarks

We have shown some of the fundamental techniques used to deal with limited bandwith and network-induced latency and the general trade-offs that they provide. It is up to individual developers to choose the right mix of these techniques that suits their game the best, as different games have different requirements. It is important to note that none of these techniques provide a silver-bullet solution to all the problems at hand and should be mixed and matched together for the best possible result.

There are other network-related optimization techniques that are discussed for example by Armitage et al. (2), Greer et al. (4), Terrano and Bettner (14) or Jan Svarovsky (7).

This is the end of the first section where we laid the foundations that we shall build upon in the second section, which deals with the design and implementation of a multiplayer mode in our own game.

2 Sublockus multiplayer design and implementation

2.1 About Sublockus

Sublockus is a video game under development by a small group of enthusiasts. We call ourselves the Velvet Studios and have been working on the project since 2011. The creative mastermind and the initiator of the whole project is Pavel Pokorný, responsible for most of the design aspects of the game. I have been the lead programmer in the team, with Ondřej Pluskal and Roman Jergon as my colaborators and consultants. The graphical assets for the game have been created by Zuzana Šebestová, the fifth and final member of the Velvet Studios.

Sublockus is a 2D platformer game that draws heavily from many different games across genres. It has been mostly influenced by Terraria by Re-Logic and action RPGs, mostly the Diablo series developed by Blizzard. Originally, the game has been created as a single-player experience with emphasis on large procedurally-generated content, including huge randomized dungeons.

The main goal of this thesis is the implementation of a multiplayer game mode, which would use the Sublockus engine. The multiplayer mode is quite a different experience, focused on player-to-player combat in smaller arenas, using a narrower set of game features that are in line with this fast-paced action-oriented gameplay. With regard to our previous classification, Sublockus would be categorized as a fast-paced action game, as each player controls a single character and the outcome of their actions depends on split-second decision-making and execution.

From a technical viewpoint, Sublockus is written in C# using .NET and especially its XNA libraries. For graphical user interface we have used the Nuclex library, but apart from that the game engine has been built from the ground up. The main role of the XNA framework is loading 2D graphical assets and drawing them on the screen and space-based sound reproduction.

Along with the Sublockus game we have developed a simple level editor, which allows us to create levels in a user-friendly graphical interface. Levels are built from individual blocks of terrain and other interactive and non-interactive objects that can be added to the level. The player, represented by a male avatar, then moves around these levels, picks up and equips different items, learns and uses abilities and improves their characteristics, such as total hit points or movement speed. This is done through fighting

AI-controlled creatures and completing randomly generated quests. It is important to note, however, that only some of these features are present in the multiplayer mode.
Such, in short, is the foundation on which we have set out to build the multiplayer component of the game.

2.2 Rules of the multiplayer arena game mode

One of the players starts the game in the server mode, which will allow other players to join their game session using an IP address and optionally a port number as identification. Once inside the game session, players are pitted against each other in a small enclosed arena. Each player has a limited amount of health points and can attack his enemies using either a melee attack or a ranged attack. Furthermore, the players may be able to use other abilities, which they can use to attack their opponents or to gain other bonuses. Once a player lands a killing blow on another player, he is rewarded a so-called frag point, which counts towards his overall score. The dead player is resurrected after a short period of time at one of pre-defined spawn-points within the level with full health. The game interface shows the number of frags and death each player has acquired in a given session.
This rather simplistic rule-set defines for us a set of entities that enter the multiplayer mode and need to be considered when designing the synchronization mechanisms for the game. Entities that enter the multiplayer arena simulation are:

- players,
- terrain and objects in the level,
- activated abilities, such as projectiles,
- items in players' inventories and equipped on them,
- frag and death statistics.

We have already established that certain actions need to be managed by the authoritative server. From the rules described above, we can distill a list of critical actions that may be initiated by one of the clients, but need to be confirmed by the server. These are:

- a player entering the game,
- an ability is being cast .

Furthermore, other events are detected by the server, which then notifies the clients. These are:

- an ability cast hitting a player,
- a player changing their statistics (e.g. current health),
- a player dying,
- a player being respawned.

This list gives us a basis for the necessary message types that need to be implemented. We will discuss messages in detail in chapters 2.7, 2.8 and 2.9.

2.3 Share only what you need to share

As we have demonstrated in the first part of this work, the single game state shared among multiplayer peers is but an illusion, as only some information is regularly synchronized while the rest is simulated locally. The key to maintaining this illusion is that the local simulations behave identically or at least similarly enough so that the player does not notice any significant discrepancies. This is a principle that we make good use of in Sublockus as well. For example, entities that are immutable, such as the arena in which players play, require no synchronization at all. All we need to assure is that every peer has the same version of the arena when joining the game, which is done by having the level packaged and distributed with the program itself. If we wanted players to create their own levels, we might have to add the level to the initial synchronization that the server sends to each connecting client. As long as no changes to the environment are possible during the game, it will stay synchronized across all peers throughout the game session.

Another example of entities and events that need not be synchronized is things that are merely visual and have no impact on the gameplay itself and cannot affect other parts of the game state, such as animations. When a player avatar dies in Sublockus a spray of blood particles will erupt from its body. These particles are treated by the same way as other projectiles, such as fireballs, but as long as their effect is merely visual, the server does not have to worry if each client is rendering these particles in exactly the same manner. We can thus save a considerable amount of network bandwidth by keeping the simulation of these objects separate on each peer and only synchronizing entities that are relevant to the game-state itself.

2.4 Building on the right foundations

Before we started out with the implementation one of the first questions that had to be answered is which libraries, platforms or services we are going to use. In our case we have identified the following options. Their respective strengths and weaknesses are listed below.

2.4.1 XNA networking

We could utilize the built-in XNA networking system, which is the simplest solution as it provides an extremely intuitive interface. For internet connections it relies on a net of Microsoft servers that connect and maintain individual game sessions. It allows for simple management of game sessions and provides a high-level abstraction (18). While LAN connections are available without special requirements, Internet connections require a LIVE Gold account and XNA Creators Club Membership – both of which cost in the range of $100. These monetary restrictions severely limit the appeal of games using this system and therefore we opted against using this system.

2.4.2 Third-party networking libraries

Second option is to use third-party networking libraries. An example of such a library is the Lidgren library (19). For other programming languages and platforms there are other libraries available, however, for C# Lidgren seems to be the most suitable option, especially when we are looking for solutions available for the general public for free. Its features and the consequences of using it are covered in this article (20) by Christopher Park. It provides a level of abstraction above the standard Windows socket interface and provides classes for simple connection management. Both LAN and Internet connections possible as long as the server has a public IP address.

As with any open-source library we have to be careful, as there are no explicit guarantees it will work or that there will be any continued support. One of the main benefits of Lidgren is that it is built using the UDP protocol, while implementing the reliability and orderliness features we mentioned in chapter 1.2 and which we are looking for.

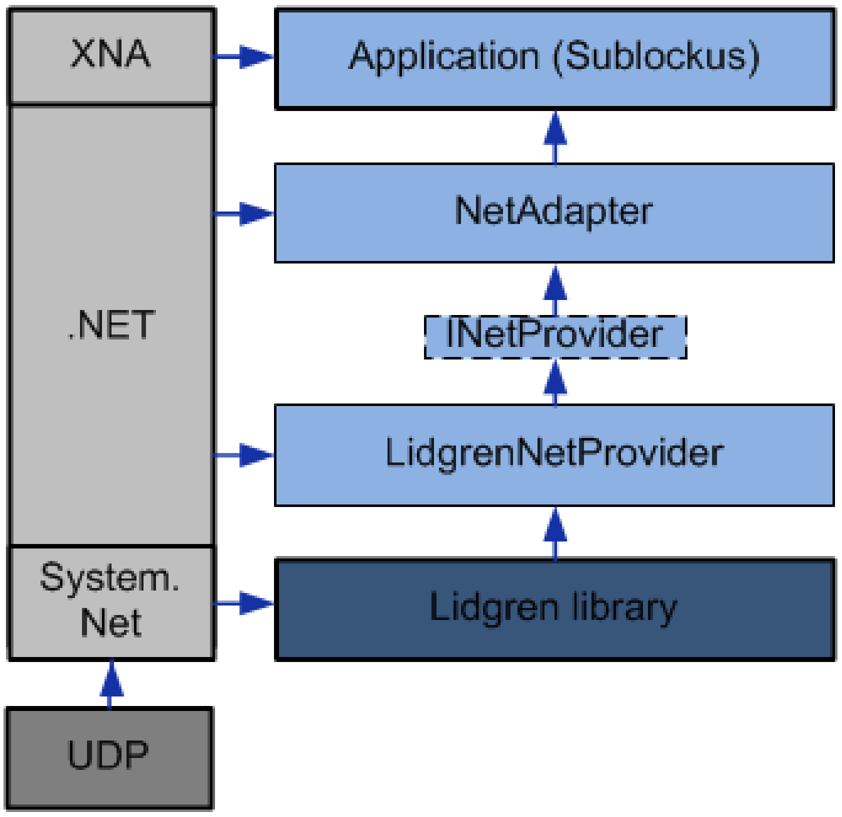
2.4.3 Custom-made framework on top of basic .NET networking

Last but not least, we could create the whole networking framework from the ground up on just what .NET provides. This is still a decent level of abstraction as we can establish TCP/IP connectios with a single command, however, there are no classes for servers, clients, game sessions, no fault handling etc. Especially if we chose to use UDP over TCP, we would have to implement a lot of general non-specific features that might steer us away from more interesting problems. Moreover, this approach would result in us writing code that has already been most probably written in better quality in a third-party library, which is generally a poor architectural choice.
Possible upside of this approach is the greater control and independence on any specific platforms, which might allow creating a more portable solution and obtaining more generally applicable know-how.

2.5 Architecture – the grand scheme of things

In this chapter we shall discuss the architecture of our implementation. We will show how we employed the Lidgren library and created a robust, flexible and reusable networking module. The key guiding concept behind this architecture was to make networking as simple as possible for the user, that is, to hide as much of the technical details as possible behind a high-level API. With this goal in mind we have created a separate module (in the form of a Visual Studio project that compiles into a standalone .dll file) called simply Networking, which exposes a small set of classes that provide the minimal networking features we are looking for. The central class of this library is the `NetAdapter` class. It provides a single point of access to all of the module´s functionalities, such as creating a server and client instances, sending messages, receiving and handling messages and provides a set of events that users can subscribe to. We shall describe this class in detail in chapter 2.6, for now suffice it to say that the `NetAdapter` class is merely an empty shell that has very little implementation and delegates most of the legwork to an instance of the `INetProvider` interface. In our case we have a single implementation of this interface in the form of the `LidgrenNetProvider`, which contains the core of the logic of our networking module. By separating the interface and the implementation in this manner we adhere to the Inversion of Control (21) principle. If we wanted a different implementation of the underlying networking communication, say, use TCP/IP instead of Lidgren's UDP, we would replace the `LidgrenNetProvider` with our custom provider and plug it in while the application using the Networking module need not change a single line of code since the `NetAdapter` class stays intact.

Figure 3: The Networking module architecture



Source: own creation

Figure 3 depicts the dependencies between the technologies, frameworks and classes used in our implementation. Both Lidgren and Sublockus use the .NET framework. Lidgren accesses the UDP protocol implementation in the `System.Net` namespace. The `NetAdapter` and `NetProvider` classes then provide loose coupling of Sublockus with Lidgren.

2.6 Networking module interface

In this chapter we shall take a slightly closer look at the interface of the Networking module. The centerpiece of the API is the `NetAdapter` class. All networking communication is handled by this class. The general idea of the `NetAdapter` is to provide the most simple and basic interface for sending and receiving messages. We shall demonstrate how this is done in a simple example that shows how to use the `NetAdapter` in a server application.

Figure 4: Example of using the NetAdapter as a server

```
/* Specify the client/server mode on construction */
netAdapter = new NetAdapter(NetPeerMode.server);

/* Signal the NetAdapter to listen as a server on a specified
port. This will enable the server to process incoming requests.
*/
netAdapter.StartServer(portnumber);

/* Optionally add event handlers to specify actions that should
be done in various situations, such as when a client connection
is approved. */
netAdapter.ClientConnectionApprovalSentHandler +=
OnClientConnectionApproved;

/* Update the adapter in a loop as long as you want it to be
running. During an update the NetAdapter will handle all incoming
messages that have arrived in between two updates. How these
messages are handled is defined separately in each message type.
The object passed to the NetAdapter is the context, usually the
ActiveGameState object */
while(serverIsRunning)
{
   NetAdapter.Update(this);
}

/* This will sever the connection between the server and its
clients. Once the session is over, dispose the NetAdapter and
thus free all of the system resources it has been using. */
NetAdapter.Dispose();
```

A second example will show how to use the `NetAdapter` in a client application. The following code sets up a `NetAdapter` in a client mode and connects to a server.

Figure 5: Example of using the NetAdapter as a client

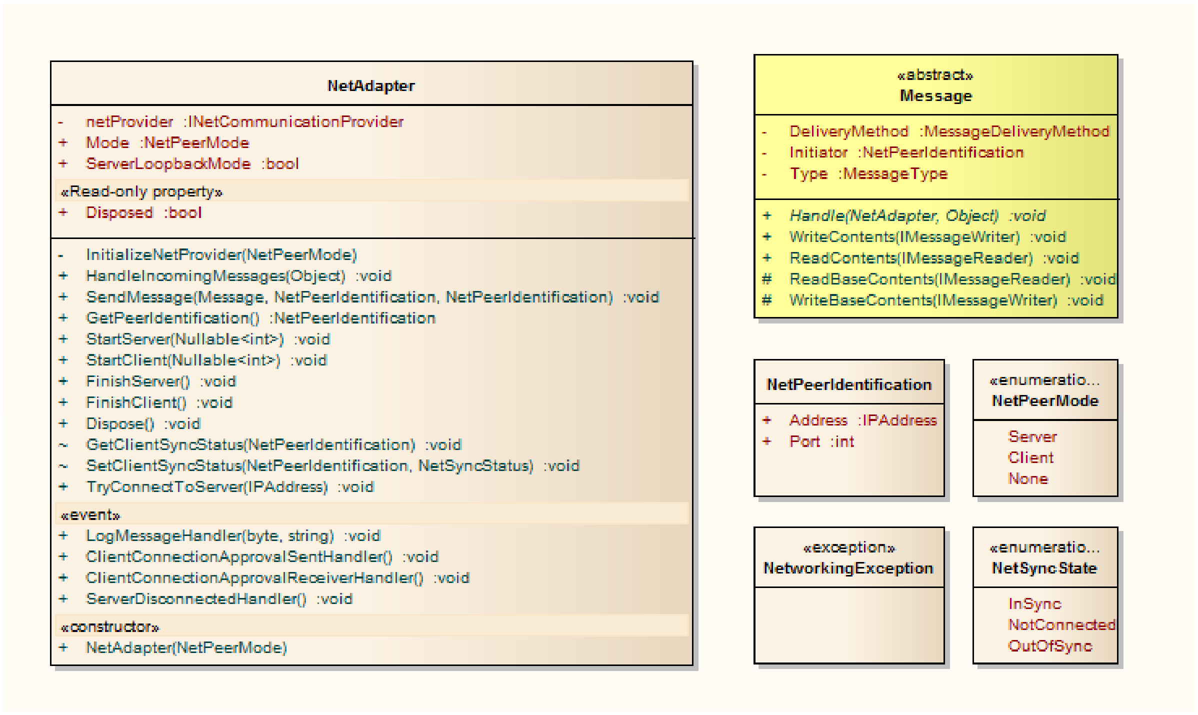
```
/* Create the NetAdapter in client mode */
netAdapter = new NetAdapter(NetPeerMode.client);

/* Start a client on a specified port number. */
netAdapter.StartClient(portNumber);

/* Attempt to connect to a server on the specified address. The
server will either accept the connection or deny it. Use the
event handlers to respond to each of those. If there is no server
listening on this address and port the NetAdapter will keep
resending the connection request indefinitely until another call
```

```
to TryConnectToServer with a different address or the NetAdapter
is shut down. */
netAdapter.TryConnectToServer(address);

/* Specify what to do once the connection to the server is
approved. Typically you will want to enter the game session. For
that you will probably need to obtain the server-side game-state.
We are using a concrete implementation of the InitSyncMessage to
tell the server to send us its game-state. */
netAdapter.ClientConnectionApprovalReceivedHandler += (() =>
{
    InitSyncMessage syncMessage = new SubInitSyncMessage();
   netAdapter.SendMessage(syncMessage);
});

/* Check for incoming messages in a loop and react to them. The
object passed to the NetAdapter is the context, usually the
ActiveGameState object */
While(clientIsRunning)
{
    netAdapter.Update(this);
}
/* To send a message to the server and through to other clients
instantiate a concrete implementation of the Message class and
send it. The framework takes care of delivering it to the right
peers. */
NetAdapter.SendMessage(new ConcreteMessage(/* your data */));


/* This will disconnect the server from the server and dispose
the NetAdapter and free all of the system resources it has been
using. */
NetAdapter.Dispose();
```

The only configuration the user of our API has to worry about in this scenario is the IP address of the server and setting both the client and the server to the same port. If no ports are specified for either the server or the client, default port number 14242 will be used.

As we can see in Figure 6, apart from the `NetAdapter` class, the framework also exposes several small classes and enumerations, which we will describe.

Figure 6: The API of the Networking module


NetAdapter
- netProvider :INetCommunicationProvider
+ Mode :NetPeerMode
+ ServerLoopbackMode :bool
«Read-only property»
+ Disposed :bool
- InitializeNetProvider(NetPeerMode)
+ HandleIncomingMessages(Object) :void
+ SendMessage(Message, NetPeerIdentification, NetPeerIdentification) :void
+ GetPeerIdentification() :NetPeerIdentification
+ StartServer(Nullable<int>) :void
+ StartClient(Nullable<int>) :void
+ FinishServer() :void
+ FinishClient() :void
+ Dispose() :void
~ GetClientSyncStatus(NetPeerIdentification) :void
~ SetClientSyncStatus(NetPeerIdentification, NetSyncStatus) :void
+ TryConnectToServer(IPAddress) :void
«event»
+ LogMessageHandler(byte, string) :void
+ ClientConnectionApprovalSentHandler() :void
+ ClientConnectionApprovalReceiverHandler() :void
+ ServerDisconnectedHandler() :void
«constructor»
+ NetAdapter(NetPeerMode)
«abstract»
Message
- DeliveryMethod :MessageDeliveryMethod
- Initiator :NetPeerIdentification
- Type :MessageType
+ Handle(NetAdapter, Object) :void
+ WriteContents(IMessageWriter) :void
+ ReadContents(IMessageReader) :void
# ReadBaseContents(IMessageReader) :void
# WriteBaseContents(IMessageWriter) :void
NetPeerIdentification
+ Address :IPAddress
+ Port :int
«enumeratio...
NetPeerMode
Server
Client
None
«exception»
NetworkingException
«enumeratio...
NetSyncState
InSync
NotConnected
OutOfSync

Source: own creation

NetPeerIdentification

This class encapsulates the information needed to identify a single peer, be it a server or client. It is a combination of IP address and port number. You can obtain the identification of the machine on which your client or server is running through the `NetAdapter.GetPeerIdentificationMethod()`. You will use these objects, for example, to specify recipients of a message in case you want to override the default by specifying the optional attributes in `NetAdapter.SendMessage()`.

NetPeerMode

This enumeration is fairly straight-forward, it is used to distinguish between the server and the client modes of the `NetAdapter`. The `None` option should never be used to initialize a `NetAdapter`, it is intended to denote that the whole application is used in a single-player mode.

NetSyncState

This enumeration lists the states in which the connection between a client and a server can be. When a client is connected to the server, but has not yet entered a game session, the `OutOfSync` value is used. Once an `InitSyncMessage` has been sent and as long as the state is kept synchronized, the `InSync` value is used. We shall look more closely at how these states are used and how the connection and initialization process works in the chapter 2.10 which covers the communication life-cycle.

NetworkingException

Exceptions thrown from within the Networking module are of the type `NetworkingException`. This allows the user to handle these exceptions separately from other generic exceptions.
The next 3 chapters focus on the last piece of the API, the message classes.

2.7 Messages

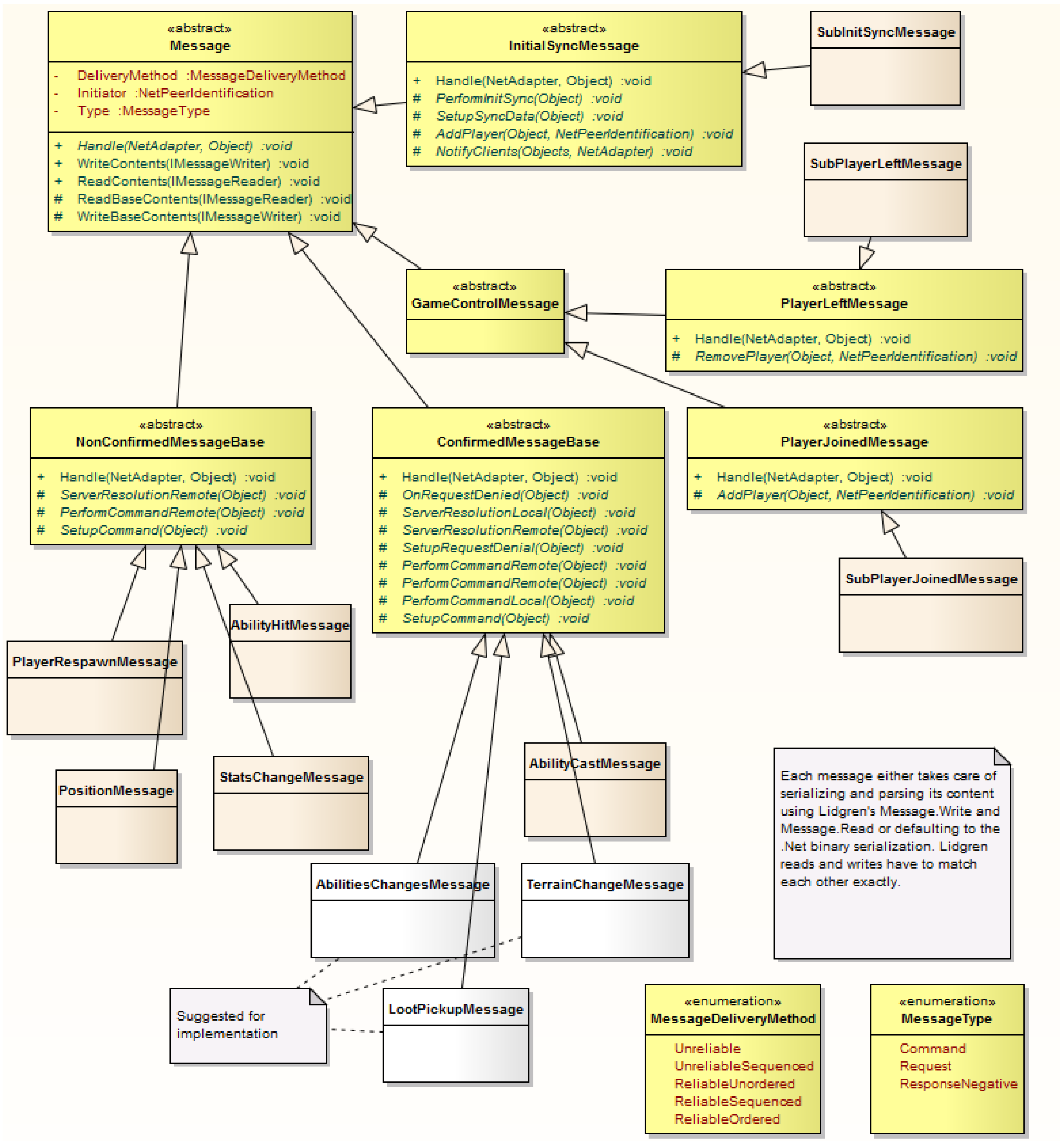
Messages encapsulate pieces of data that are sent back and forth between the server and its clients. Their key purpose is to keep the state of the communicating peers synchronized. Each message specifies its data structure as well as the functionality associated with it. Every message is either created on the server or the client, initialized with relevant data and then sent using the `NetAdapter` class. The message itself then specifies how it should be handled by the recipient through the `Handle(NetAdapter adapter,Object context)` method. The context object is whatever object you pass to the `NetAdapter` to operate on when receiving messages. In our case it is the whole game state. Received messages are thus able to modify the game state, which is what we usually want when we receive a message. Using this approach we can keep all of the logic pertaining to a single message in one place, which makes it easier to maintain.
To make the implementation of messages as streamlined as possible we provide several base classes for messages that can be used in different situations. Using the template method pattern, each of these base classes overrides the `Handle` method and implements a routing algorithm, which specifies which actions should be taken at which point and where the message should be sent. There are two main message types: the `ConfirmedMessageBase` and the `NonConfirmedMessageBase`.
The `ConfirmedMessageBase` class is used for messages that are initiated by the client and sent to the server for approval. The server then decides whether the requested action is legal or not and either changes its state accordingly and notifies all clients to update their state as well or sends back a denial message to the initiating

client. The whole process is illustrated further by the diagram in Figure 9. Messages inherited from the `ConfirmedMessageBase` do not have to worry about communication sequence and its orchestration – they merely implement the individual parts of the template method. These methods are as follows:

- **OnRequestDenied** – This method is called on the client when the request returns denied by the server.
- **ServerResolutionLocal** – This method implements the server-side resolution of the request for the primary client. This allows the user to use different implementation for the primary client and remote clients. This method returns a boolean value which indicates whether the request is approved or denied.
- **ServerResolutionRemote –** This method implements server-side resolution of the request for a remote client. This method returns a boolean value which indicates whether the request is approved or denied.
- **SetupRequestDenial** - This method is called to prepare the message with data sent to the client upon request denial, such as inserting a message that explains why the request was denied.
- **PerformCommandRemote –** Execute the command on a remote client (that did not initiate the request). This typically means changing the state of remote players or resolving actions initiated by them.
- **PerformCommandLocal** – Execute the command on the client that initiated the request. This is called when the server has approved the request.
- **SetupCommand** - This method is called to prepare the message with data sent to the client upon successful request resolution. The same setup is used to send a command to the initiating and remote clients.

The `NonConfirmedMessageBase` is used for messages that need not be confirmed by the server. This type of messages is used to increase the responsiveness of certain typical player actions that can be resolved on the client. An example of this is the `PositionMessage` which relays information about a player’s position and movement. Since player movement is affected by the environment that is always static and identical on the server and the client, we can trust the client to calculate the movement correctly[3]. This is a direct implementation of the concepts discussed in chapters 1.6 and 1.7.

Figure 7: Message inheritance tree


«abstract» Message
- DeliveryMethod :MessageDeliveryMethod
- Initiator :NetPeerIdentification
- Type :MessageType
+ Handle(NetAdapter, Object) :void
+ WriteContents(IMessageWriter) :void
+ ReadContents(IMessageReader) :void
# ReadBaseContents(IMessageReader) :void
# WriteBaseContents(IMessageWriter) :void
«abstract» InitialSyncMessage
+ Handle(NetAdapter, Object) :void
# PerformInitSync(Object) :void
# SetupSyncData(Object) :void
# AddPlayer(Object, NetPeerIdentification) :void
# NotifyClients(Objects, NetAdapter) :void
SubInitSyncMessage
SubPlayerLeftMessage
«abstract» GameControlMessage
«abstract» PlayerLeftMessage
+ Handle(NetAdapter, Object) :void
# RemovePlayer(Object, NetPeerIdentification) :void
«abstract» NonConfirmedMessageBase
+ Handle(NetAdapter, Object) :void
# ServerResolutionRemote(Object) :void
# PerformCommandRemote(Object) :void
# SetupCommand(Object) :void
«abstract» ConfirmedMessageBase
+ Handle(NetAdapter, Object) :void
# OnRequestDenied(Object) :void
# ServerResolutionLocal(Object) :void
# ServerResolutionRemote(Object) :void
# SetupRequestDenial(Object) :void
# PerformCommandRemote(Object) :void
# PerformCommandRemote(Object) :void
# PerformCommandLocal(Object) :void
# SetupCommand(Object) :void
«abstract» PlayerJoinedMessage
+ Handle(NetAdapter, Object) :void
# AddPlayer(Object, NetPeerIdentification) :void
SubPlayerJoinedMessage
AbilityHitMessage
PlayerRespawnMessage
PositionMessage
StatsChangeMessage
AbilityCastMessage
AbilitiesChangesMessage
TerrainChangeMessage
Each message either takes care of serializing and parsing its content using Lidgren's Message.Write and Message.Read or defaulting to the .Net binary serialization. Lidgren reads and writes have to match each other exactly.
Suggested for implementation
LootPickupMessage
«enumeration» MessageDeliveryMethod
Unreliable
UnreliableSequenced
ReliableUnordered
ReliableSequenced
ReliableOrdered
«enumeration» MessageType
Command
Request
ResponseNegative

Source: own creation

On top of these two base message classes we have implemented an array of messages that handle all of Sublockus multiplayer features. Figure 7 shows the whole inheritance

[3] Note that every time we trust the client to supply the server with any calculated information, we open up possibilities to cheaters to modify the program and circumvent the game‘s rules. If we are worried about cheating, but do not want to check every request with the server before performing it, we can perform so-called sanity checks, which can detect abnormal behaviour and disconnect a cheating player.

tree for messages used in the Networking module (colored yellow) and their concrete implementations used in Sublockus (colored light pink). They are the following message types:

PlayerRespawnMessage

This message is sent by the server to a client whose player has been killed and after a specified period of time has been revived by the server. This ensures that the respawn action is initated by the server, wchich removes the possibility of one of the clients respawning a player while others do not and de-synchronizing the session.

PositionMessage

This message is sent the most often. It contains position, speed and acceleration information of the player as well as the direction thay are facing and the directions they are moving in. Remote players are updated based on this information. In between position updates the movement is driven by the built-in physics engine. The `PositionMessage` is currently sent once every 100ms, which results in fairly smooth movement. Discrepancies are noticeable only in very high velocities, such as when a player jumps.

AbilityHitMessage

This message is sent by the server once it detects an `AbilityCast` (i.e. a melee attack or a fireball projectile) hitting one of the players. Each client then resolves the hit in the same way, typically reducing the health of the hit player and removing the `AbilityCast` object.

StatsChangeMessage

This message updates the stats, such as health or mana, of a remote player. Currently it is sent by the server only once a player's health drops below zero, resulting in death. Since the clock on each client is not perfectly synchronized, there can be slight discrepancies in the speed at which health is gradually regenerated over time. To prevent this from affecting the dead/alive status of a player we are relying on the server to signal when a player is dying, since it is a critical change in the game-state that needs to be synchronized at all times.

AbilityCastMessage

This message is sent by a client when its player wants to cast an ability. Since casting an ability is conditional upon the player having enough resources (e.g. mana), it needs to be confirmed by the server, which holds the only true authoritative state of the game.
Furthermore, there are two other base message classes that handle the general tasks of adding and removing players from the game and initializing new players with the current server state.

InitialSyncMessage

The `InitialSyncMessage` allows a client to request and receive the whole state of the server. Since our module is based on a join-in-progress mechanism (instead of a game lobby (as presented in chapter 1.5), it is necessary to set the game-state of a joining client to the same state as the server has. It is essential then for the server to send all parts of its game-state that are mutable. The immutable parts can be loaded and set up by the client, but they have to be populated by the dynamic elements, such as the individual players and objects within the level they can interact with. Only then can we rely on sending incremental changes. This means the initial synchronization needs to send a relatively large amount of data. In our case it contains the `RemotePlayer` objects for each player in the game, but it could easily contain the whole level with its objects. Using a similar approach as with the confirmed and non-confirmed messages, the Networking module provides an abstract base class that handles the communication logic and requires the client merely to specify which data are to be sent to the client and how the client game-state should be initialized.
In addition to that the `InitialSyncMessage` sends a `PlayerJoinedMessage` to all other clients when a client joins the game.

PlayerJoinedMessage and PlayerLeftMessage

`PlayerJoinedMessage` allows clients to add the remote player to their collection of remote players and start receiving updates for that player. It is sent by the server when a client sends an `InitialSyncMessage`. On the other hand, once a server detects that a client has disconnected, it will send a `PlayerLeftMessage` to all remaining clients.

In this message the clients should remove the leaving player and make all adjustments to the game state as necessary.

2.8 Message serialization – the two approaches

In this chapter we shall take a look at how messages are serialized. Serialization is the process of converting an object into a stream of bytes that can then be sent over the network inside IP packets. De-serialization is then the reverse process of creating the object from a stream of bytes. The Lidgren library provides a simple interface similar to `StreamWriter` and `StreamReader` with methods like `WriteString(string)`, `WriteByte(byte)`, `ReadString()` or `ReadByte()` for most of the built-in types. When serializing an object we call the appropriate `Write` method for each member of the given object. When de-serializing, we call the appropriate read methods in exactly the same order. Each `Message` class can override the `WriteContents` and `ReadContents` method where we can specify which members of the message should be serialized and how. For illustration we have added an example from the `PositionMessage` in Figure 8.

Figure 8: Example of message serialization

```
public override void WriteContents(IMessageWriter writer)
{
   WriteBaseContents(writer);

  writer.Write(this.entityId);
  writer.Write((byte)this.entityType);
  writer.Write(position.X);
  writer.Write(position.Y);
  writer.Write(speed.X);
  writer.Write(speed.Y);
  writer.Write(acceleration.X);
  writer.Write(acceleration.Y);
  //More of the same
}


public override Message ReadContents(IMessageReader reader)
{
   ReadBaseContents(reader);

   entityId = reader.ReadString();
   entityType = (PositionEntityType)reader.ReadByte();
   position = new Vector2(reader.ReadFloat(), reader.ReadFloat());
   speed = new Vector2(reader.ReadFloat(), reader.ReadFloat());
```

```
        acceleration = new Vector2(reader.ReadFloat(),
        reader.ReadFloat());
        //More of the same
        return this;
    }
```

Note the call to ReadBaseContents(IMessageReader), which takes care of serializing the type information of the message, so that it can be properly instantiated upon de-serializing. We are also using the IMessageReader and IMessageWrite interfaces, which provide a way to call the serialization methods of the underlying Lidgren message without relying on the specific Lidgren implementation.

This approach, while fairly effective, is somewhat cumbersome to code and difficult to maintain. The networking module provides an alternative that utilizes the built-in .NET Serialization. If a Message class does not override the ReadContents and WriteContents methods, the NetAdapter will default to this .NET Serialization process. We use a BinaryFormatter to serialize the whole Message class and then use the WriteBytes(byte[]) method to write it to the underlying Lidgren message. All a message then needs to do is use the [Serializable] attribute and mark all non-serialized fields with the [NonSerialized] attribute. Alas, the drawback to this approach is the added overhead of the general serialization. Since the .NET serialization uses Reflection, it takes a lot longer (22) to perform the serialization and de-serialization than simply to write the values into a stream. Even more importantly, though, there is a lot of overhead in the serialized data. From our experimental measurements, messages serialized using this approach were about 15 times larger than using the first method. This makes it unsuitable for messages that are sent often. In Sublockus, we use this second method for messages that are sent sparsely and do not represent the bulk of the traffic. Further optimization could be made to this automated serialization process by using some of the more efficient serialization libraries such as Protocol Buffers by Marc Gravel, which might bring it considerably closer to the basic Lidgren option.

2.9 Message routing

Having described the interface of the Networking module and the concepts behind the NetAdapter and Message classes, in this section we shall look into the dynamics of the system. We shall describe the communication sequence that happens between the

client and the server. We will take a closer look at the `ConfirmedMessageBase` and how it is sent between the client, the server and remote clients.
We can divide the communication into 3 steps:

1. **The client requests an action.**

   When the client wants to perform an action that it cannot resolve based on its client-side game-state, it sends a message to the server instead. The message is of the type `MessageType.Request`.

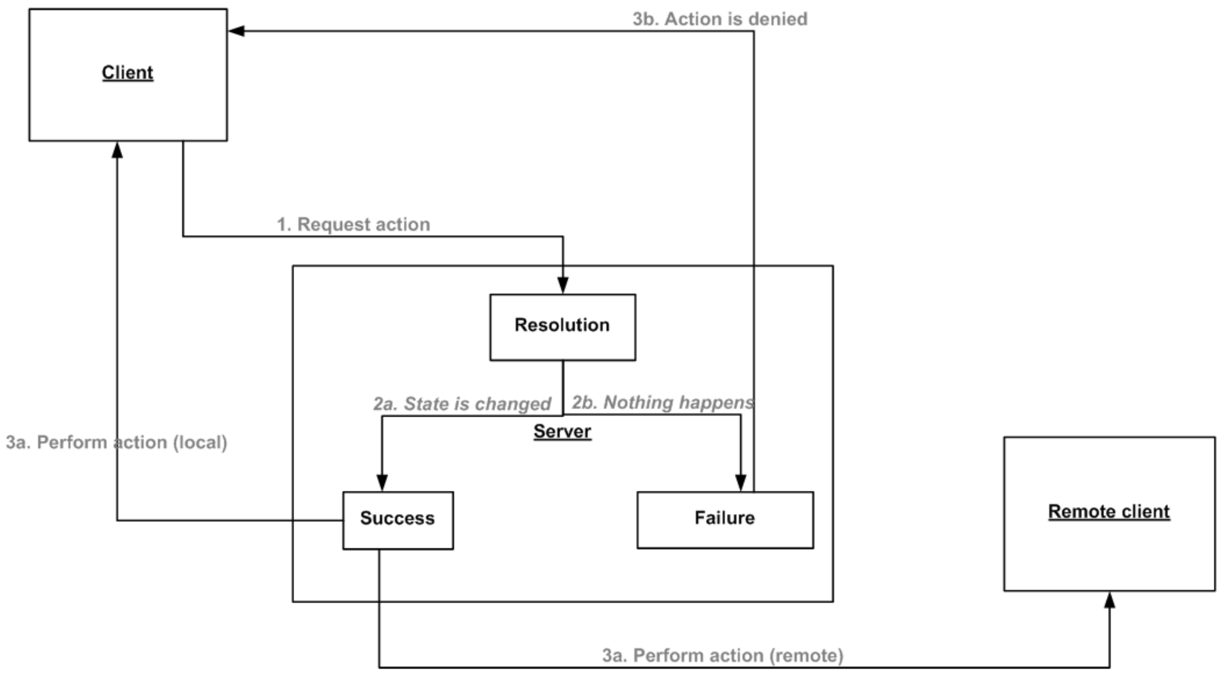
2. **Server resolves the message.**

   Once the server receives a request message, it runs the specified resolution implemented in the `ServerResolutionRemote(object)` which may alter the game-state. If the resolution succeeds, the server prepares the message to be sent as a `MessageType.Command`. Otherwise it sets the type to `MessageType.ResponseNegative`.

3. **Server sends response**

   In the third step the server sends its response. If it is a negative response, it is sent merely to the initiating client. In case the resolution succeeded and the game state has changed, all clients need to be notified. The message is loaded with the necessary information for the clients to perform the action and sent to all clients including the initiator.

Figure 9: Message routing



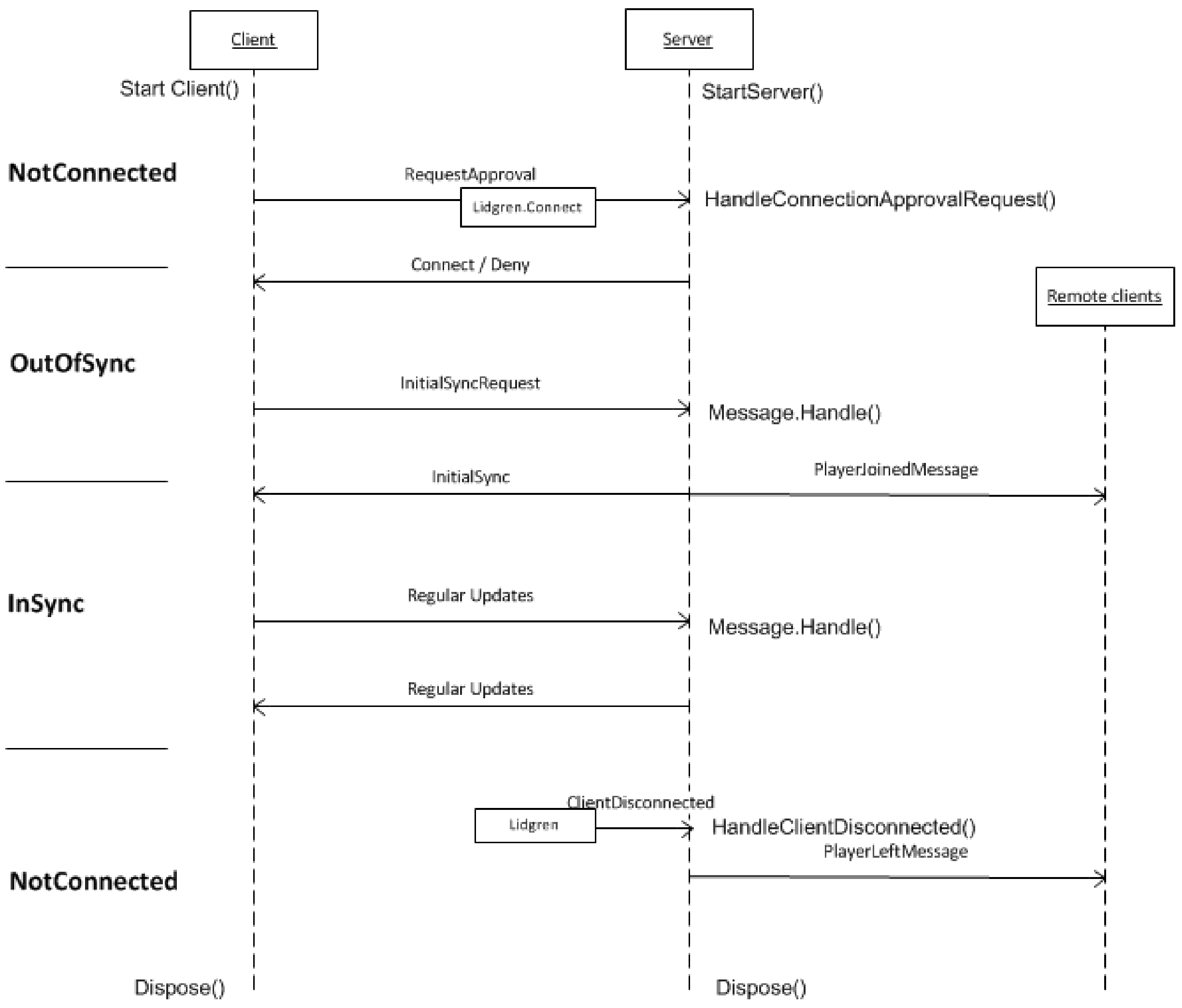
Source: own creation

2.10 Initialization and communication life-cycle

As described in chapter 2.6 and as is depicted in Figure 6 the client has 3 different states: `NotConnected`, `OutOfSync` and `InSync`. These states present a simple lifecycle and determine what kind of messages is sent between the client and the server.

In the diagram in Figure 10 we explore an example of a possible communication scenario.

Figure 10: Peer communication life-cycle



Source: own creation

Once started, a client is in the `NotConnected` state, which means it cannot send or receive any messages. The only thing it can do is to request connection to a server specified by an IP address through the `TryConnectToServer(IPAddress)` method. If the server returns a positive reply[4] and adds the client to its set of active connections, the client will switch to the `OutOfSync` state. This state allows the client to send and receive messages even though the game-state of the client is not synchronized with that of the server. This state enables the client and the server to possibly negotiate configuration and parameters of further communication. However, in our current implementation it is used to simply to send the `InitialSyncMessage` (described in chapter 2.7). Once the server fills the `InitialSyncMessage` with data

[4] Currently the server is set up to accept a connection based on a password sent by the client, which is by default empty. This could be generalized by adding an event to the `NetAdapter` that would accept a delegate that would perform an arbitrary check instead.

and sends it back to the client, it switches the state of the client to `InSync`. From that point on, this client is considered synchronized and its player is a part of the game state and the server will keep sending all update messages to it. Other remote clients are notified via the `PlayerJoinedMessage`.

It is important for all unordered messages to check whether the client-side game state is properly initialized, since it is possible for an unordered message to arrive at the client before the `InitialSyncMessage`. Messages sent via the ordered delivery method need not perform this check.

Also, the initialization process might create a considerable lag on the client, especially when sending a large amount of data. One might want to take extra measures to prevent this lag from affecting the gameplay experience.[5]

Once the server sends the `InitialSyncMessage`, the client is considered `InSync` and it can send and receive any messages. `InSync` clients send player messages with player actions to the server, which sends back changes in its state, which are prompted either by other remote clients or triggered on their own by the passing of time[6]. In this state, messages are sent back and forth until one of the peers quits the session, either by calling `finishClient()/finishServer()/Dispose()` or due to a failure in the underlying connection. A disconnection is detected by the Lidgren library which sends a virtual message to the server, which then notifies the remaining clients. If the server is disconnected, the game session ends and all clients are disposed.

2.11 Remaking a single-player game

One of the potential risks of the implementation of the multiplayer mode into Sublockus was the fact that the game was from the very beginning designed as a single-player-only game. Obviously, the architecture of a multiplayer application would be very different from a single-player version. In a single-player game, all the logic is executed on a single

[5]It might seem tempting to send the initialization data to the client, let it initialise and wait for an O.K. message and only then to proceed. Alas, in the meantime the server-side game state will have probably changed and the client would no longer be in sync if we started sending the incremental update messages now. A possible solution might be to buffer all incremental update messages while the client is initializing and send them once it is ready to receive them, however, that might create additional lag if the amount of buffered messages is high. A typical workaround to this situation is to give each new player a short grace period during which they are invulnerable to enemy attacks. This allows the client to perform all neccessary initialization while the player is already considered initialized by the server. This protection can be dropped once the player performs an action.

[6]Such as a projectile colliding with a player

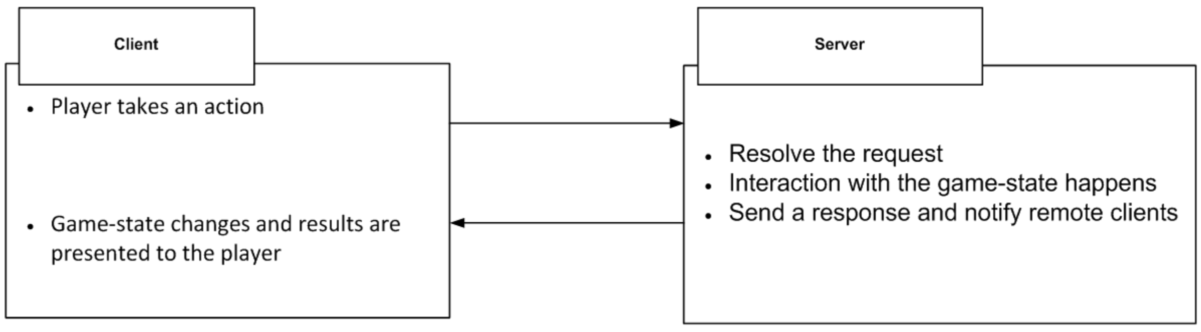
machine within a single application. This allows for a less structured architecture as it does not matter that much which pieces of code execute which functionality. However, when parts of the logic need to be executed on the server, the pressure on good organization of code is much higher. As a result, we were anticipating a major overhaul of the game that would entail changing the game logic in very many places. We mitigated this by including only the most vital functionalities into the multiplayer, leaving some of the complex interactions (such as inventory management and updating player models with different armor pieces) to the single-player portion. It is important to note that the challenge here is mainly in the amount of work needed in going through all the functionalities and creating specific messages for each of them rather than in designing the system and its logic.

In summary, the process of enabling multiplayer in previously single-player Sublockus boils down to the following: we look at the single-player game as a client that performs no communication with the server. It then consists of three stages:

- player taking an action,
- interaction with the game-state happening,
- game state changing and results being presented to the player.

For a client-server game, we need to take the middle part and relocate it to the authoritative server. The resulting structure is depicted in Figure 11.

Figure 11: Execution flow in a client-server architecture


Client
Player takes an action
Game-state changes and results are presented to the player
Server
Resolve the request
Interaction with the game-state happens
Send a response and notify remote clients

Source: own creation

This shift in logic execution from the client to the server (which is usually inside a message class) is quite easy to implement.

Additionally, we need to alter the reaction to events that are not triggered by player actions. These now have to be controlled and resolved by the server. In Figure 12 we can see an exemplary piece of code that handles the effects of a collision between a player and a projectile. The first case block in the switch represents what was previously done in the single-player version. For the multiplayer mode, we added a switch for each of the multiplayer modes (server, client, none). If a collision is detected on a client, we do nothing (we are waiting for a message to arrive from the server to tell us if the collision actually happened). On the server, if a collision is resolved and if it really happened, we send the `AbilityHitMessage` to all clients.

Figure 12: Example of collision resolution on the client

```
if (r1.Intersects(r2))
{
   switch (game.MultiplayerMode)
   {
   case NetPeerMode.none:
      collisionDetected = true;
      projectile.OnCollision(mob, player);
      break;
   case NetPeerMode.server:
      collisionDetected = true;
     bool result = projectile.OnCollision(mob, player);
     if (result)
      {
        AbilityHitMessage message = new
        AbilityHitMessage(projectile, mob);
        game.NetAdapter.SendMessage(message);
      }
      break;
   case NetPeerMode.client:
      //do nothing
        break;
   }
}
```

The execution is the finished on the side of the client in the `AbilityHitMessage` inside the `PerformCommandRemote(object)` method. The client finds the affected `Player` in its collection of players and the `AbilityCast` that caused the collision. It then performs the collision in the same way the server does in the previous example.

Figure 13: Example of collision resolution in AbilityHitMesage

```
protected override void PerformCommandRemote(object context)
{
   ActiveGameState game = SubNetUtils.GetGameState(context);
   PlayerBase player = game.getPlayerById(targetId);
   //Additional checks omitted from example
   Mob target = player.mob;
```

```
        AbilityCast abilityCast =
        game.activeAbilities.GetAbilityCastById(abilityCastId);
        Projectile projectile = abilityCast as Projectile;
        if (projectile != null)
        {
                projectile.OnCollision(target, game.player);
                projectile.collisionHandler.CollisionDetected = true;
        }
    }
```

Another important issue was the representation of remote players. There are two main approaches to the remote player object, it could either be a `Player` class – same, or very similar to the local `Player` instance, or it could be a totally separate class that wraps the `Mob`[7] class and adds just the components that are necessary to operate a remote player entity. Although the second approach seems like an enticing choice, since it leaves the logic for controlling the player to their `Player` class, as it is unique to the local player entity while the remote player entity uses a different control mechanism that takes inputs from the server, we opted for the former. We transferred part of the implementation of the `Player` class into `PlayerBase` and the extended it to create the `RemotePlayer` class. The only significant difference is that the `RemotePlayer` class takes its movement input from incoming `PositionMessages`, whereas the `Player` class takes input from the keyboard.

2.12 Listen-server or dedicated server

One of the chief concerns when building a client-server game is to decide on the type of the server to be used. There are two basic approaches, the listen-server and the dedicated server. The principal difference between the two is in the deployment of the server process. In dedicated server architecture, the server is confined to a wholly different program than the client. It is run in a separate process and typically on a separate machine. Conceptually, this is what one would expect, as in all of the previous chapters we have talked about the client and the server as being two separate entities, so it makes sense for them to be separate once deployed. Dedicated servers typically provide no graphical output, merely a text console, or some other configuration and management tool. A dedicated server merely receives messages from clients and sends

[7] A mob is any moving entity in the game capable of performing an attack. It is common to players and AI controlled characters. It is characterized by a body that is subject to game physics and a list of abilities. However, the `Mob` class does not implement certain game-specific features, such as items inventory or control mechanisms.

responses. This approach is most often used by publishers or other operators running servers on dedicated hardware with back-bone internet connectivity, so as to enable a high number of players joining a single game session (23) (24).

In many cases, however, it is not necessary to have more than a handful of players in a single game session and then we can utilize the listen-server architecture. A listen-server is a server that also takes in user input and provides graphical output, just like a client application would. This way there are no additional costs for dedicated hardware and bandwidth. In Sublockus we use the listen-server architecture, as it allows us to neatly package the server and client inside a single application. Another reason for choosing this approach is the single-player foundations of the game. Basically, we have added an optional server component to the client application.

Going down this route we had to resolve how the primary client (the client that runs inside the same process as the server) will communicate with the server. Since they are utilizing the same game state, it is not possible to make the distinction between a primary client and other clients totally transparent, as we shall demonstrate with the following thought experiment. Imagine an architecture, where the client would send a message to the server that resides within the same process (for now it does not matter how the message is sent and relayed). Let us use the good old ability cast as an example. When a client wants to cast an ability, this does not alter its state, but sends a request message to the server instead. The server then interprets the message and changes its game-state accordingly. So far we are fine even with the primary client that shares its state with the server. However, once the server has resolved the request, it will send a reply to the client and the client will then change its state as well. With a naive listen-server architecture where the primary client and server share the same state this would lead to a disaster, since the client logic would then make the second game-state change which would often lead to unintended results.

To prevent this from happening, we provide a different logic for resolving a message from the primary client and then prevent the server from sending the response to the primary client. In case of non-confirmed messages there is a similar problem, when the client first changes its game-state and then the server changes its state before notifying other remote clients. Again, there is a simple solution in that the server does not alter its game state when the non-confirmed message has arrived from the primary client. Thus we accomplish that the primary client and the server can share the same game state without compromising its consistency[8].

2.13 The space for improvement

Having covered most of the fundamental design decisions and implementation challenged in the multiplayer mode of Sublockus, it is time to step a little further away and comment on the results. We deem that the multiplayer mode implemented into Sublockus has reached the goals set forth by this thesis. At the same time, there are many features we originally intended to include in the multiplayer mode that have not made the cut, mainly due to time constraints. The resulting player-versus-player experience is very smooth and responsive, at least on LAN connections, which is where we performed most of our testing.

One of the greatest challenges (and aids, at the same time) has been the Lidgren library and learning to use it properly. We can say that the library is working mostly as advertised and delivers on all of its promises of being a lightweight solution for sending and receiving messages with very little CPU and bandwidth overhead while providing reliability and orderliness on top the UDP transport layer protocol. However, it is quite sparsely documented, which makes using it somewhat of a trial-and-error experience. This has proven to be especially challenging when dealing with some of the alternative scenarios, such as disconnects, multiple connection attempts or network overloading. As a result, the framework is a little rough around the edges and could be improved to handle some of these fringe scenarios more gracefully.

Generally speaking, there are multiple opportunities for further development of this thesis. One such interesting addition would be to implement a TCP communication provider that would provide similar reliability features as the Lidgren library using the TCP protocol. We could then easily compare the two implementations with our benchmarking test. Another direction of further development of the Networking framework is to implement some of the optimizations mentioned in chapters 1.8.3 to 1.8.5 such as time-delay or time-warp. Note, however, that these optimizations would probably require considerable modifications of the framework as it is not truly feasible

[8] Anyone overriding the base `Message` class directly has to account for this situation, otherwise this issue is made transparent by using the other base message classes.

to have the game state shared between the server and the primary-client for these optimizations to work.

A more easily implemented optimization would be to use spatial information to reduce the number of position updates that are sent between peers. Currently, each client sends 10 position updates every second no matter where its player is located in the level. Given the rules of the game, it is impossible for 2 players who are far away from one another (so far they are not rendered on each other's screen at least) to interact in any way. Therefore, we could forego these messages as they have no impact on the gameplay. The trick here is to recognize the moment when these two players approach so that they require these position updates again. A possible solution would be to dynamically scale the frequency at which the position messages are sent – the farther away from other players you are, the less frequently you send your position updates. The server can then decide which clients to update, not sending irrelevant position updates, which could further cut down on the bandwidth requirements.

3 Benchmarking

3.1 The design

In the third and last part of this work we shall take a step back and look at the final application we have created and put it through its paces. We have designed experimental measurement of selected properties of the network communication occurring in Sublockus. Since a multiplayer game is a complex system and there is a great number of measurable properties (principally, we can divide them into 2 categories: properties of the network communication and performance of the server or individual clients) determined by a huge number of variables, we limited ourselves and focus on measuring the effects of different delivery methods. One of the main architectural features of our Networking framework is the use of Lidgren networking library, which provides options for reliable and/or orderly delivery on top UDP. It seems fitting, then, to measure the effects of sending certain messages in an unreliable manner, which should generally be more efficient than the reliable and ordered delivery. We shall demonstrate that using the unreliable delivery method (e.g. UDP instead of TCP) is beneficial to the resulting latency. The difference should manifest especially in high packet-loss scenarios (20).

First and foremost, we explain how we calculated round trip time. Since we are measuring time across multiple computers we cannot simply obtain the time for the moment we send a message on one machine and the moment we process the message on the other machine as the clock on each machine may be shifted by an arbitrary delta. To synchronize the clocks on all our machines we use the Network Time Protocol, as implemented by David L. Mills (25), which sets the clock on each machine to a common time. The NTP protocol may typically generate an error in the range of a few milliseconds based on network properties (26).

In our experiment we measured how the client-server round trip time changes based on different delivery methods under different network conditions. We accomplished that by adding 2 timestamps to every message that specified the time of its creation and sending. Then upon receiving the message on the other peer we logged the event with the following information:

Message id, type of the message, time of creation, time of sending, time of receiving, time of processing, current network latency, current network jitter, current network packet drop-rate, message size and delivery method.

We logged the information into a text file using the CSV format, which allowed us to easily analyze the data in a spreadsheet application.

Note that by adding the timestamps to each message we slightly changed the messaging protocol, by adding extra overhead to each message, which changes its actual size and could theoretically affect the latencies. Compared to the overall size of an average message, however, is this change negligible and should not introduce a significant bias to our measurement.

3.2 The measurements

We performed each measurement in a distinct session, each configured to different Network settings and using a different delivery method. We simulated the various network conditions by configuration parameters in the Lidgren library. Each session consisted of starting a server, connecting a single client to it and sending just position updates for 1 minute. Thus we obtained, hopefully, a dataset where the only significant variables were the ones we were consciously changing – those being minimum latency simulated in the network and simulated packet-loss rate, whose effects on the overal latency we measured across three distinct delivery methods. We chose the unreliable delivery method, reliable-unordered and reliable-ordered. The values chosen for additional simulated latency were 0ms, 200ms and 1000ms. Packet-loss rate was tested at the 0, 0.01 and 0.1 levels. As 0.1 (i.e 10%) packet loss is an extremely high value for actual real-life scenarios, we changed the packet-loss rate to 0.05 in the last high-latency scenario, to allow us to establish a stable connection. Even though we have thus obtained a rather sparse data set, it has given us a clear indication of how each of the tested protocols behave and to make an informed conclusion.

In every test session we measured the time it took for each message to get from one peer to the other[9]. We then averaged these times to come up with a single number representing the latency for the given configuration. The resulting data are presented

[9] The latency we were measuring here is on the level of our Networking module. That is, we write down the time when a message is handed over to the Lidgren framework via the `Send()` method. Similarly, we log the time when a message is given over to our framework through Lidgren's `receiveMessage()` method.

below in tabular and graphical form. The complete measurement logs are available in the digital attachments section.

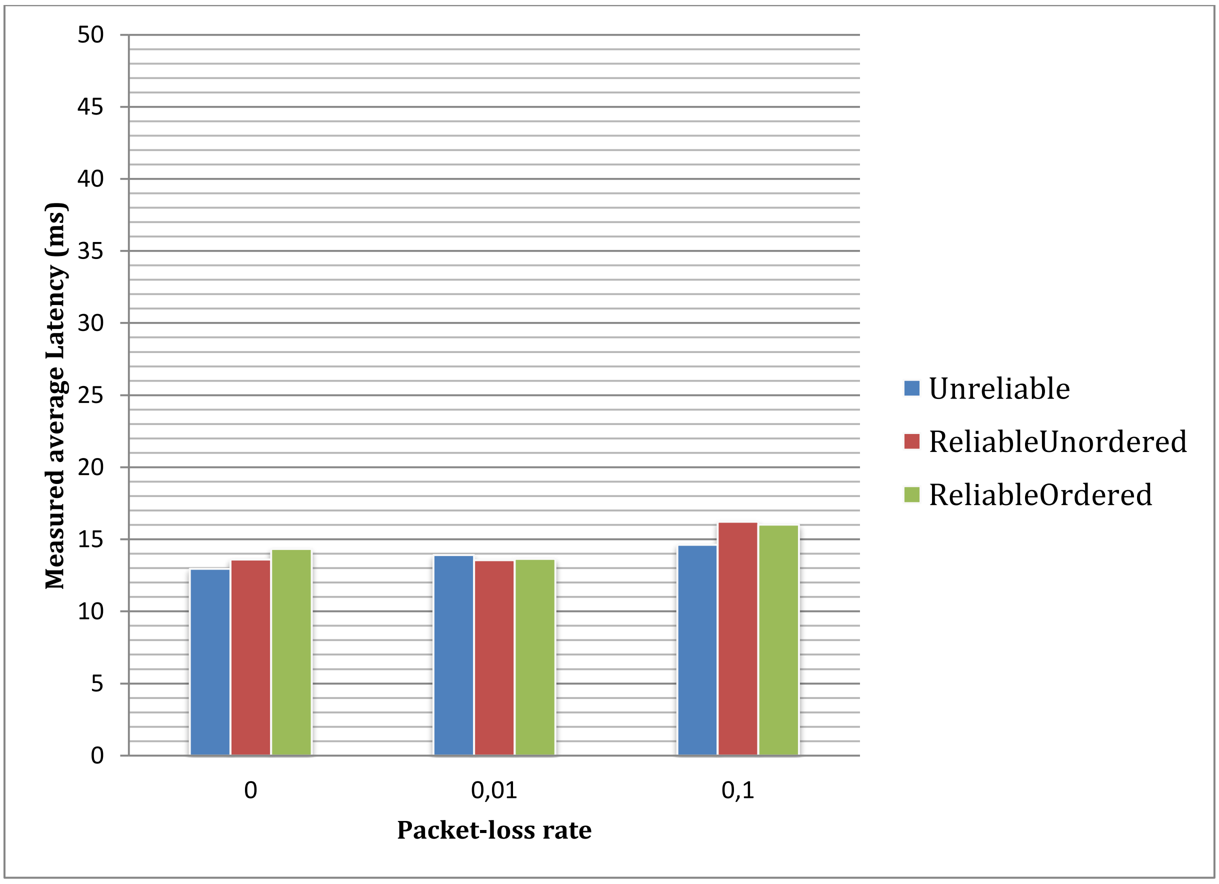
Low minimum latency network (0ms)

Table 1: Effects of packet loss in a low latency network

		Delivery method		
		Unreliable	**ReliableUnordered**	**ReliableOrdered**
packet-loss	**0**	12.9660	13.5980	14.3371
	0.01	13.9098	13.5660	13.6409
	0.1	14.6354	16.2290	16.0208

Source: own creation

Figure 14: Effects of packet loss in a low latency network



Source: own creation

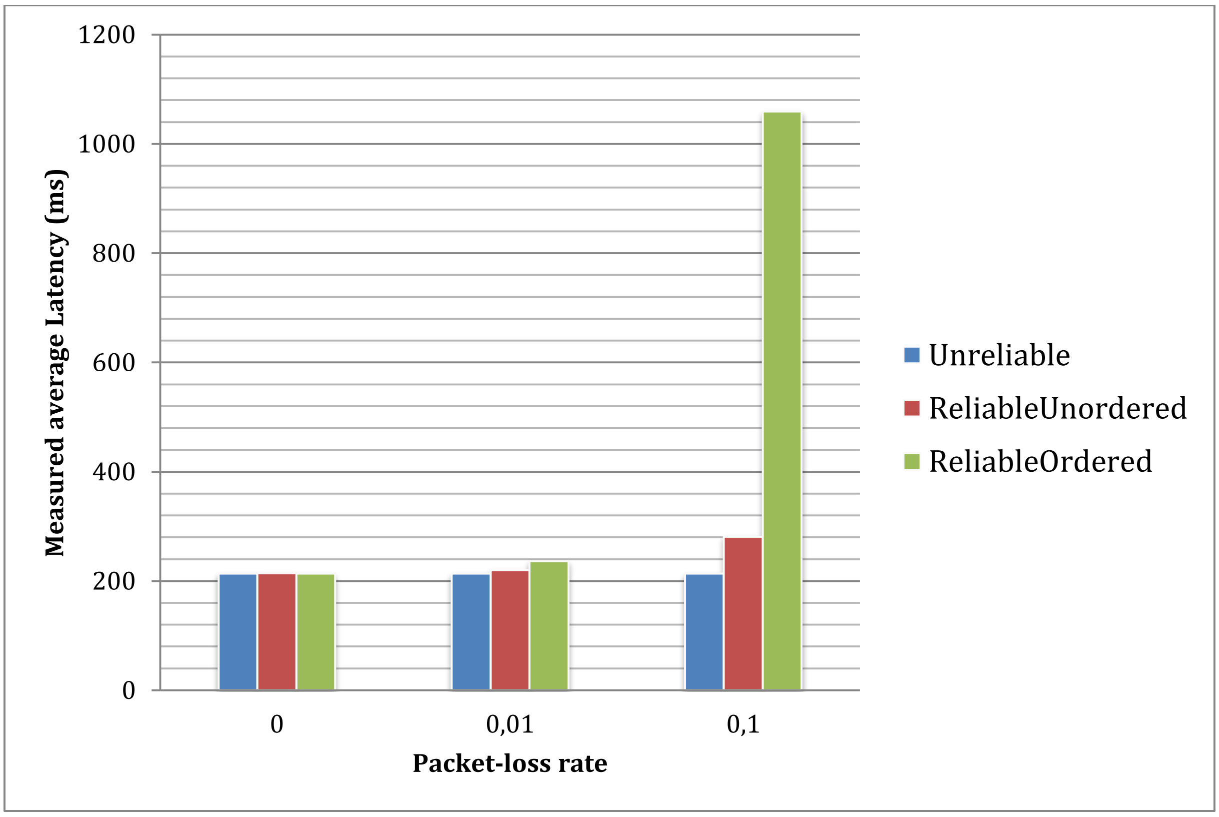
Medium minimum latency network (200ms)

Table 2: Effects of packet loss in a medium latency network

		Delivery method		
		Unreliable	ReliableUnordered	ReliableOrdered
packet-loss	0	213.5659	214.0323	213.1152
	0.01	213.4817	219.5952	236.1904
	0.1	213.0655	281.0277	1059.0600

Source: own creation

Figure 15: Effects of packet loss in a low latency network



Source: own creation

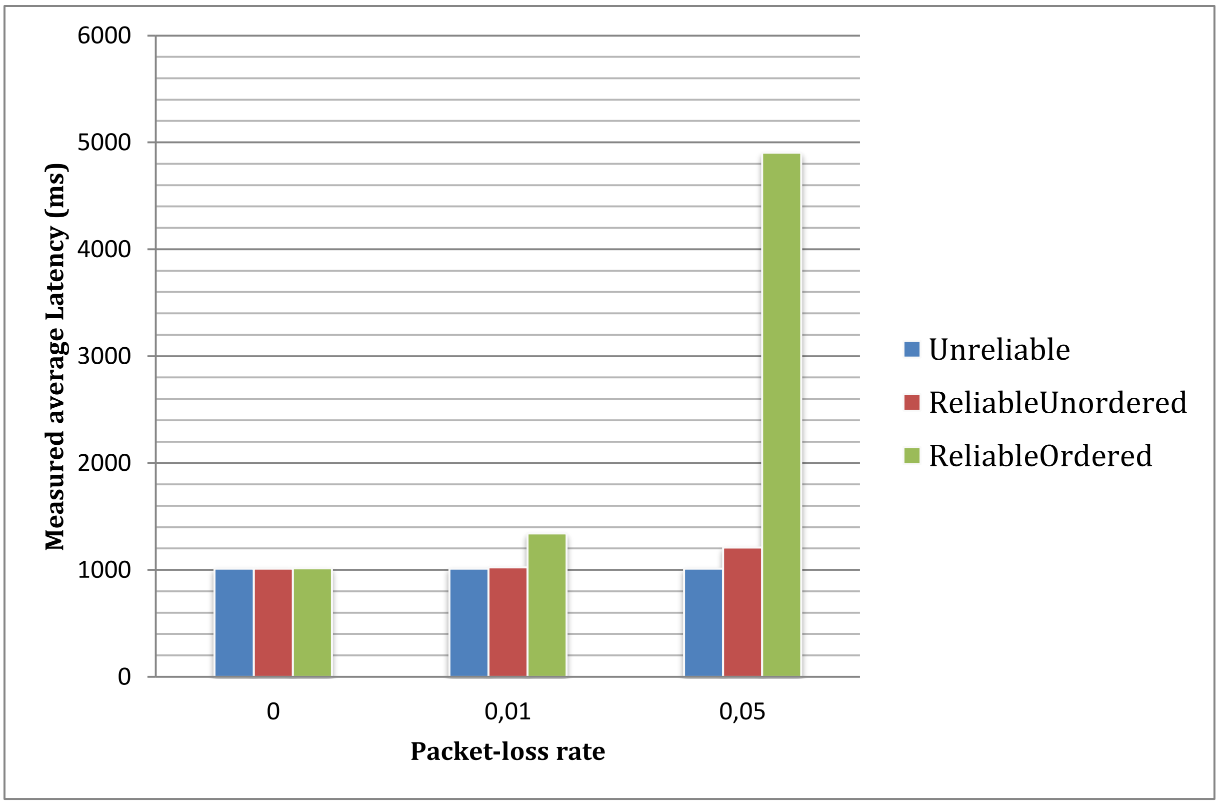
High minimum lantecy network (1000ms)

Table 3: Effects of packet loss in a high latency network

		Delivery method		
		Unreliable	**ReliableUnordered**	**ReliableOrdered**
packet-loss	**0**	1013.5867	1013.7413	1014.2981
	0.01	1013.4400	1024.1473	1342.1371
	0.1	1013.0925	1209.5096	4904.3536

Source: own creation

Figure 16: Effects of packet loss in a high latency network



Source: own creation

3.3 The analysis

From the collected data we can discern clearly that the reliable-ordered method creates a massive overhead on slow connections with high packet-loss rates. For a 1000ms connection with 5 % drop rate, the percentage increase in time it took on average for a message to arrive at its destination was nearly 500% compared to unreliable delivery. This confirms our hypothesis that unreliable connection will perform better in poor network conditions. It is interesting to note that the reliable-unordered delivery method, while being slower than the unreliable method, is incurring a much smaller penalty. If we take a closer look at the raw data gathered during the experiment we will see the reason for that. When a packet loss occurs in a reliable-ordered protocol, all messages sent after the lost one have to wait in line, until the lost message is successfully resent and only then they are processed one by one. Essentially, the next several messages after a packet loss incur the same latency penalty as the lost message. This penalty gradually decreases until base-level latency is reached or another pack-loss occurs. On the other hand, with the reliable-unordered delivery method, only the lost message incurs the extra latency, while others are unaffected. Therefore, this delivery method should be strongly preferred for messages that need to be delivered reliably, but we can deal with them arriving in random order.

Meanwhile, for low latencies, as shown in Figure 14, the differences between the delivery methods are almost negligible, approaching the measurement error of our experiment. A possible explanation for the perceived better performance (relative to the high-latency scenario) is that the queueing effect described above does not occur. Since a message is sent roughly once every 100ms, if a packet loss occurs, there is enough time to accommodate a resend before the next message is forced to wait upon arrival.

This demonstrates that the general approach of sending most messages via the unreliable or at least unordered channel, and using the reliable-ordered method for only a selected handful of messages, is correct.

Conclusion

In the first section of this thesis we have explored the multiplayer mode in different computer games in its many varieties. We have discussed the key design and implementation differences and have shown the options and tools we have at our disposal to create multiplayer computer games. We have catalogued a variety of optimization algorithms and explained how they pertain to different genres of online multiplayer games.

We have also discussed the general principles and described their advantages and drawbacks and made certain decisions that we applied in the second part – the implementation of multiplayer mode into an action-oriented platformer game. We have chosen the transport protocol to be UDP. We have described the client-server and peer-to-peer network topologies and designed our own implementation of the listen-server architecture. We have designed a custom conflict resolution strategy that divides all actions into non-critical actions that are not confirmed by the server and critical-level actions that are confirmed by the server.

In the second part we have designed and implemented multiplayer mode in the game *Sublockus*. We have built a reusable networking library that uses the Lidgren networking library and provides all of the networking features used in Sublockus' multiplayer mode. We have described the interface of this library and some of its key components.

By enhancing the Sublockus game with a multiplayer mode, we have greatly increased the playability and general appeal of the product and increased the odds of its success facing our competition. There is still a considerable amount of work to be done before we can release though. In terms of networking, this will mainly consist of extensive testing in different network setups and improving the stability of the game as well as reducing the size and amount of messages sent.

After implementing the designed networking library and multiplayer mode we conducted a series of benchmarks designed to test the effects of different message delivery methods under various network conditions. From these measurements we have concluded that the unreliable delivery method, inherent to UDP, is the right option for messages that are sent often and are not sensitive to packet loss.

Taken as a whole, we believe, this work may serve well as a guide to any programmer new to the area of online multiplayer games.

Bibliography

1. **Adams, Ernest.** *Fundamentals of Game Design.* San Francisco : New Riders, 2013. ISBN 0133435717.
2. **Armitage, Grenville , Claypool, Mark and Branch, Philip.** *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games.* New York : John Wiley & Sons, 2006. p. 232. ISBN: 0-470-01857-7.
3. **Smed, Jouni and Hakonen, Harri.** *Algorithms and Networking for Computer Games.* 1st. New York : Wiley, 2006. p. 264. ISBN 978-0470018125.
4. **Greer, Jim and Simpson, Zachary Booth.** Minimizing Latency in Real-Time Strategy Games. [book auth.] Dante Treglia. *Game Programming Gems 3.* Hingham : Charles River Media, 2002, pp. 488-495.
5. *An empirical evaluation of TCP performance in online games.* **Chen, Kuan-Ta, et al., et al.** New York : ACM, 2006. ACE '06 Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology . ISBN 1-59593-380-8.
6. **Fiedler, Glenn.** UDP vs. TCP. *Gaffer On Games.* [Online] [Cited: 3 21, 2015.] http://gafferongames.com/networking-for-game-programmers/udp-vs-tcp/.
7. **Svarovsky, Jan.** Real-Time Strategy Network Protocol. [book auth.] Dante Treglia. *Game Programming Gems 3.* Hingham : Charles River Media, 2002, pp. 496-505.
8. *Latency and player actions in online games.* **Claypool, Mark and Claypool, Kajal.** 11, New York : ACM, 11 1, 2006, Vol. 49, pp. 40-45. ISSN 0001-0782.
9. *On the impact of delay on real-time multiplayer games.* **Pantel, Lothar and Wolf, Lars.** New York : ACM, 2002 . NOSSDAV '02 Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video . pp. 23-29. ISBN 1-58113-512-2.
10. **Aarseth, Espen, Smedstad, Solveig Marie and Sunnanå, Lise.** A MULTI-DIMENSIONAL TYPOLOGY OF GAMES. [Online] 2003. [Cited: 1 2, 2015.] http://www.arts.rpi.edu/public_html/ruiz/egdfall09/readings/new%20topology%20of%20games%20A.%20Arsneth.pdf.
11. **Barron, Todd.** *Multiplayer Game Programming.* Roseville : Prima Tech, 2001. ISBN 0-7615-3298-6.
12. *Local-lag and timewarp: providing consistency for replicated continuous applications.* **Mauve, Martin, et al., et al.** 1, Piscataway : IEEE Press, 2004, IEEE Transactions on Multimedia, Vol. 6. ISSN 1520-9210.
13. **Sanchez-Crespo, Daniel.** *Core Techniques and Algorithms in Game Programming.* s.l. : New Riders Games, 2003. ISBN 978-0131020092.
14. **Terrano, Mark and Bettner, Paul.** 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond. *yale.edu.* [Online] 3 22, 2001. [Cited: December 6, 2014.] http://zoo.cs.yale.edu/classes/cs538/readings/papers/terrano_1500arch.pdf.
15. *Consistency in replicated continuous interactive media.* **Mauve, Martin.** New York : ACM, 2000. CSCW '00 Proceedings of the 2000 ACM conference on Computer supported cooperative work . pp. 181-190. ISBN 1-58113-222-0.
16. *How to Keep a Dead Man from Shooting.* **Mauve, Martin.** Enschede : Springer Berlin Heidelberg, 2000. Interactive Distributed Multimedia Systems and Telecommunication Services. Vol. 1905, pp. 199-204. 978-3-540-41130-7.
17. **Alexander, Thor.** A Flexible Simulation Architecture for Massively Multiplayer Games. [book auth.] Dante Treglia. *Game Programmin Gems 3.* Hingham : Charles River Media, 2002, pp. 506-519.

18. **Hargreaves, Shawn.** *Shawn Hargreaves Blog.* [Online] Microsoft. [Cited: 3 21, 2015.] http://www.shawnhargreaves.com/blogindex.html.
19. **Lidgren, Michael.** Lidgren networking library. *code.google.* [Online] [Cited: 1 2, 2015.] https://code.google.com/p/lidgren-network-gen3/.
20. **Park, Christopher M.** Choosing A Network Library in C# . *Games By Design.* [Online] Blogspot, 2 24, 2010. [Cited: 3 21, 2015.] http://christophermpark.blogspot.cz/2010/02/chosing-network-library-in-c.html.
21. **Fowler, Martin.** Inversion of Control. *Martin Fowler.* [Online] June 26, 2006. [Cited: 5 28, 2015.] http://martinfowler.com/bliki/InversionOfControl.html.
22. **Novak, Maxim.** Serialization Performance comparison (C#/.NET). *M@X on DEV.* [Online] March 25, 2014. . [Cited: 6 28, 2015.] http://www.maxondev.com/serialization-performance-comparison-c-net-formats-frameworks-xmldatacontractserializer-xmlserializer-binaryformatter-json-newtonsoft-servicestack-text/.
23. **Armstrong, Andrew.** The Mammoth Dedicated Server Guidebook. *http://www.mammoth.com.au.* [Online] Mammoth Medi, 2009. [Cited: 6 12, 2015.] http://www.mammoth.com.au/~/media/92E27A0F97F34A958D6D4B9F3CD2A64F.ashx.
24. **Abdelkhalek, Ahmed, Bilas, Angelos and Moshovos, Andreas.** "Behavior and performance of interactive multi-player game servers.". [book auth.] Rajkumar Buyya. *Cluster Computing 6.* 4. Toronto : Prentice Hall, 2003, pp. 355-366.
25. **Mills, David L.** *Computer Network Time Synchronization: The Network Time Protocol.* New York : CRC Press, 2006. ISBN-1420006150.
26. **Murta, Cristina D., Torres-Jr., Pedro R. and Mohapatra, Prasant.** Characterizing Quality of Time and Topology in a Time Synchronization Network. *NTP Survey.* [Online] Nov 27, 2006. [Cited: 6 29, 2015.] http://www.ntpsurvey.arauc.br/globecom-ntp-paper.pdf.

List of figures

List of tables

Attachments

Attached to this text is a CD with digital content including:

- the installation package,
- the installation guide,
- the source codes,
- the license agreement.