Design and Development of a Distributed Multiplayer Game

Masterarbeit

Alexis Davidson
Fachbereich Mathematik und Informatik
Institut für Informatik
Freie Universität Berlin, Deutschland

February 4, 2019

Erstgutachter: Prof. Dr. Katinka Wolter
Zweitgutachter: Prof. Dr.-Ing. Jochen Schiller

Selbstständigkeitserklärung

Hiermit bestätige ich, dass ich die vorliegende Ausarbeitung mit dem Titel *Design and Development of a Distributed Multiplayer Game* selbstständig und ohne unerlaubte Hilfe angefertigt habe.

Ich versichere, dass ich ausschließlich die angegebenen Quellen in Anspruch genommen habe.

Statement of Authorship

I declare that this thesis entitled *Design and Development of a Distributed Multiplayer Game* is the result of my own research except as cited in the references. The thesis has not been accepted for any degree and is not concurrently submitted in candidature of any other degree. Except where acknowledged in the customary manner, the material presented in this thesis is, to the best of my knowledge, original and has not been submitted in whole or part for a degree in any university.

Alexis Davidson
Berlin, February 4, 2019

Abstract

The choice of the network architecture or topology is crucial when developing an online software. The developer can choose between adopting distributed topologies or a client-server network architecture. In the video game development field, the client-server architecture is usually favored over the distributed one.

This work aims at investigating the specific pros and cons for these two network architectures by designing and developing a game inspired by notable existing multiplayer video games in both topologies. In this game project we run an experiment where AIs play together on each version. We record network and cheating data and analyze them. This data is then used to propose different compromises that can be made for game developers to adopt an optimal solution.

Our results show that the distributed method generated a surprisingly lighter network bandwidth than the client-server method. However, the code complexity and time spent on the implementation were a magnitude higher than for the client-server solution. As expected, the distributed solution and its voting system allow a high percentage of cheating which depends on the number of malicious players in the same game session. Furthermore, the distributed network architecture shows slower reaction times because of additional phases happening when exchanging messages to acquire consent for an action.

If a game company does not have the capital or the confidence in their success to invest in dedicated servers, it will choose the topology to adopt in accordance with the nature of the game. In most cases, our results show that games are playable in both topologies. However, there are specific characteristics of the game that do not support the one or the other network architecture. One of the possible cost-efficient solutions that would improve the quality and reduce the cheating in online games would be to rent a cheap web server. Such solution would apply in both network architectures.

Contents

List of Figures

Chapter 1.

Introduction

Before creating an online application, developers have to choose between the client-server and the peer-to-peer (hereinafter "***P2P***") network architectures. They both have different advantages and drawbacks that will influence programmers. Programmers will then make their choice based on their application's goals.

The client-server network architecture

The client-server model is a network architecture in which components follow a hierarchical structure. The server is the central component, controlling lower-level components, called clients. Only the server has authority and the clients have no autonomy. In order to do a task or obtain data, a client must send a request to the server and wait for an answer containing the permission for the task or the requested data.

The client-server architecture gives the developers and the server administrators control over the data flow. It is easy to implement and it allows to enforce updates, assure the connectivity as desired, collect different data for statistics, and more [1].

The P2P network architecture

The P2P architecture is a network architecture in which components communicate with each other directly and not through a server. This means that costs for running servers are dropped and privacy can be kept, which is welcomed in messaging systems, e.g. *FireChat*[1], *Telegram*[2] and *WhatsApp*[3] or payment systems, e.g. *Cringle*[4], following the idea of a P2P topology using end-to-end encryption.

The P2P model can only make sense when an application is based on the interaction between its users. Other examples of P2P applications are: digital currency, file sharing systems, task distribution applications, etc. or even a multiplayer video game where users confront each others online [2].

[1] Russians Are Organizing Against Putin Using FireChat Messaging App: `https://www.bloomberg.com/news/2014-12-30/russians-are-organizing-against-putin-using-firechat-messaging-app.html`, accessed: 04.10.2018

[2] End-to-End Encryption, Secret Chats: `https://core.telegram.org/api/end-to-end`, accessed: 04.10.2018

[3] WhatsApp FAQ - End-to-end encryption: `https://faq.whatsapp.com/en/android/28030015/`, accessed: 04.10.2018

[4] Will Cringle win the race?: `https://www.companisto.com/en/article/article-2212`, accessed: 04.10.2018

Different P2P topologies

However, the P2P network model can be implemented in different ways. A modern approach is to simulate a client-server hierarchy in which a user takes the role of the server, controlling a session, and still communicates with other users directly [3]. Another approach is to let all clients have the same amount of rights and to have no centralized authority. This is a decentralized distributed system [4] and it presents many challenges [5, 6].

How developers choose which network model to follow varies depending on the nature of the software they aim to develop. In some cases, the decision may not matter so much and in other cases there is not even a choice. When creating a website for example, the data a user requests have to come from a server, administrated by the website creators[5]. In video games, there is no obligated network model to implement. The developers are free to choose which one to use, depending on the nature of their game and the pros and cons ensuing.

1.1. Motivation

Similarly to real time systems like air traffic control systems, networked multimedia systems, space systems, etc. [7] video games present very specific networking challenges [8]. The example of the classic chess board game illustrates these challenges well.

The example of chess

To be able to play the same game together people need a common context they can react to. In chess, every player sees the same board hence the same game state. In this case, there is only one device on which the game is played, so only one sate of the game.

It gets more complex when a game is being played on different devices. In correspondence chess, there are different chess boards so different game states in different locations. When a player makes a move, it sends the information of the move (2 byte of data is enough) to the other player so it can synchronize its state of the game.

If a cat were to knock the game board over, the player would need to reestablish its state of the game. It could either replay all moves from the beginning to get to the current state or ask the other player to send a picture of its game state. If each player was to send a picture for each move, there would not be any desynchronization risk but it would not be optimized at all (a chess state board is 32 byte of data).

Synchronizing modern online games

For chess it might not seem to be an issue, but in modern online games a lot can happen on the screen (see Fig. 1.1). There are player positions, ongoing animations, environmental changes, particle effects, physics, etc. composing a much larger state space. This is such a challenge to synchronize in real-time that games adapt or limit their gameplay based on this technological barrier, and lots of different tricks are used to overcome it.

[5] Mozilla - What is a web server?: `https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server`, accessed: 19.10.2018

Figure 1.1.: Battlerite (2016): a real-time online game with a large state space

The example of the synchronization of animations illustrates one of these tricks well. Instead of sending an animation for it to be played on another device, get it preloaded on every device and only send the information about which animation is to be played. Some states are preshared and in this case create the illusion that the animation was sent.

Also keeping in mind which information is crucial for the gameplay, hence need to be synchronized, and which information is not crucial for the gameplay, like the weather in the background, and hence need not to be synchronized, is a very good and straightforward way to optimize the network traffic.

All these challenges illustrate the necessity of picking the network model that is the most adapted to the gameplay and the game's needs, when developing a video game.

Known advantages and drawbacks of the client-server and distributed P2P network architectures

The advantages of the client-server network architecture include [9]: (i) simplicity in implementation; (ii) cheating prevention; (iii) control and administration.

The drawbacks include: (i) single point of failure; (ii) bad scalability; (iii) costs in dedicated servers or additional network and computation load on the player host.

The advantages of the distributed P2P network architecture include [4]: (i) no single point of failure; (ii) good scalability; (iii) decentralization and no server costs.

The drawbacks include: (i) complexity in implementation; (ii) weak cheating prevention.

Given these advantages, what are the reasons distributed P2P is usually never used for online video games and why are client-server models and topologies favored? Are there ways to overcome the possible drawbacks of using distributed P2P and what would be their requirements?

1.2. Related works

While existing works on the subject mainly focus on the cheating aspect in online games, in this work, we aim to directly compare different aspects of the client-server and the distributed topologies in the design and development of a real-time online game.

Works relating to cheating in online games

With the use of a server and checksum algorithms, Mönch et al. present in [10] an approach that prevents an attacker from modifying a game-client and from accessing sensitive information in the game-client's memory.

Baughman et al. [11] explore different exploits possible for cheating in real-time, multiplayer games for both client-server and serverless architectures. The presented methods for anti-cheating are heavy in complexity.

In [12], Kabus and Buchmann design an online P2P game with the goal of making it cheat resistant. The method implemented is not suited for fast-paced action games.

Difference with related previous works

In this work, we directly compare client-server and distributed topologies in the context of real-time online video game development. In order to do that, we create two demo versions of a simple online video game. The first version uses the client-server topology and the second version implements the distributed topology.

We record network and cheating data for both versions and compare the results. We analyze them and interpret the advantages and drawbacks for each network architecture. Finally we propose different compromises that can be made for game developers to adopt an optimal solution.

1.3. Plan

This work is organized in the following manner:

- In chapter 2, we introduce basic knowledge in telematics that is relevant to this work.
- In chapter 3, we briefly present modern games in general, along with their different types and a few of many possibilities and impacts of cheating in games.
- In chapter 4, we introduce different existing techniques for cheating in (online) video games and typical ways to counter them. We also look at the additional challenges coming with a distributed architecture.
- In chapter 5, we present an overview of typical game network architectures.
- In chapter 6, we determine the principle, rules and characteristics of the game we aim to develop for this work.

- In chapter 7, we clearly define the fundamental characteristics of our game and we describe the tools we use. For each network architecture, we describe the concept and choices that have been made and their implementation.
- In chapter 8, we illustrate how and in which environment and setting the experiment runs.
- In chapter 9, we look at the results generated by the experiment, compare those from different architectures and settings, analyze, and interpret them.
- In chapter 10, we summarize our experience with this work and weigh the pros and cons of the different network architectures that we interpreted from the experiment. Finally, we propose different solutions for developers for choosing an appropriate network architecture for their game.
- The Appendix A is a manual to run the software on a local machine.
- The Appendix B documents the source code of the project.

Chapter 2.

Introduction to system models and communication

Before diving into the core of the subject, there is some basic knowledge in telematics that needs to be mentioned. In this chapter, we briefly introduce systems models and some basic communication foundations that we need to consider.

2.1. Distributed systems

A distributed system is a collection of independent components that appears to its users as a single, coherent system [4]. The components are located on different computers within a wired or wireless network. To achieve a common goal, the components communicate and coordinate their actions by passing messages to one another [13].

Examples of distributed systems include the following [4, 14]:

- Telephone and cellular networks
- Computer networks such as the internet
- World Wide Web
- Multiplayer online games
- Information systems, Flickr, YouTube, Google Scholar
- Network file systems
- Environmental management (sensor networks, disaster warning systems)

2.1.1. Challenges and upsides

No common physical clock

A physical clock is an electronic device that processes to record the passage of time [15]. Every computer contains its own physical clock. In some distributed systems, it is required to know at what time particular events happened. In these cases, synchronizing physical clocks is necessary and can prove to be a challenge.

Heterogeneity, variety and difference

Distributed systems may contain many different kinds of hardware and software working together in order to solve problems [16, 17]. Heterogeneity defines the diversity in a distributed system in the following aspects:

- Hardware: computers, phones, embedded devices, etc.
- Operating System: Windows, Linux, Mac, etc.
- Network: local network, Internet, etc.
- Programming languages: C++, Java, Python, etc.
- Developer roles: programmers, system administrators, etc.

Despite this diversity, the distributed system must stay coherent and the data must be able to travel through different systems without losing significance.

In video games, heterogeneity often means that a game can be playable on different operating systems. The most efficient way to achieve this is to use a cross-platform game engine.

Openness

The openness of a computer system determines at which degree this system can be extended and reimplemented in various ways. The published interfaces must be well-defined so that it is easier for developers to add new features in the future and thus extend the whole system.

Security

Since each node has the same level of hierarchy, security in distributed systems is a challenge. Any node could be malicious and without a central authority, specific algorithms must be used in order to detect and banish attackers.

Scalability

As the number of users increases, distributed systems must be scalable. The scalability of a system is defined by its ability to handle the addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity [18].

Failure handling

Computer systems sometimes fail. There can be many reasons for that: bad electrical connection, power supply failure, operating system failure, network failure, etc. Even when a computer has no total failure, it can still run and produce incorrect results due to hardware or software faults. Handling failures in a distributed system can be difficult.

Concurrency, consistency under concurrent requests

In a distributed system, resources can be shared by clients. Several clients may attempt to access a shared resource at the same time. In such a concurrent environment, data need to remain consistent.

Transparency

Transparency is an important property that hides the fact that processes and resources are physically distributed [19]. Distributed systems designers must hide the complexity of the systems as much as they can so that the system is perceived as a whole by the user. The quality of a distributed system can be assessed through its transparency.

2.1.2. Centralized systems

In a centralized system, a central component has authority over the system and controls the lower-level components. In such a hierarchy, it can instruct a middle level component to instruct a lower-level component. The centralized architecture can be seen as equivalent to client-server systems which will we see in the next section.

2.1.3. Decentralized systems

In a decentralized system, the lower-level components operate on local information, not on instructions from a controlling component. They work together in a complex manner and are equally responsible for contributing to the common cause of the system.

2.2. Network architectures

There are different architectures to adopt for components in a network.

2.2.1. Client-server

The client-server architecture reflects a centralized architecture. The server is the central component, controlling lower-level components, called clients (see Fig. 2.1). Only the server has authority and the clients have no autonomy. In order to do a task, a client must send a request to the server and wait to receive the permission for it. There can be multiple layers or tiers of servers, meaning that a server can be a client to another server. In general, the client-server architecture implies a lot of waiting and does not scale well.

Client-server for software

The client-server model is used on most online software. A common example is when accessing a website, the client sends a request to a centralized server in the hope to receive the content of a web page back (see Fig. 2.2).

Nowadays, with the increasing availability of the Internet, offline software also exist online. The advantage: save your work to a centralized cloud (server) and you can access it from

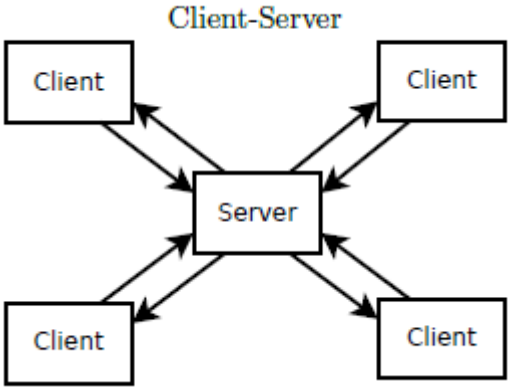

Figure 2.1.: Topology of a client-server network with four clients

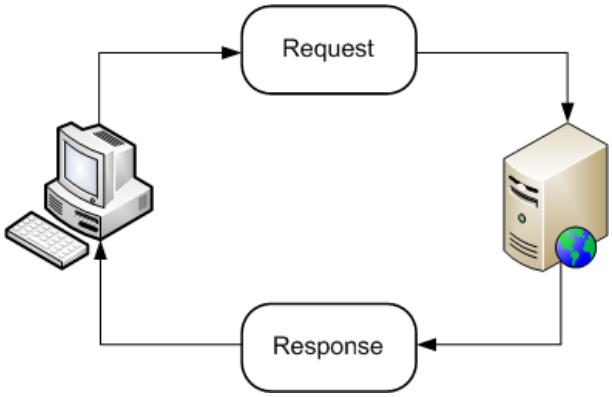

Figure 2.2.: Requesting a web page content from the server

any machine in the world. One other advantage is that there is less risk to lose your work in case your machine or drive fails. The offline software examples that we cited before: a standard text editor, a file manager or a classic offline music player, all now exist in online, synchronized versions: *Google Docs*[1], *Microsoft OneDrive*[2], *Spotify*[3], to cite some.

2.2.2. P2P

The peer-to-peer (P2P) network architecture reflects a decentralized architecture. Components are called peers, nodes or clients and are equally privileged. There are different types of P2P architectures: structured and unstructured.

Structured P2P

In a structured P2P architecture, the network overlay is organized into a specific topology. When a node joins the network, the affected nodes must adapt and update their connections to different neighbours in order to keep the structure intact. Having a structure topology enables the use of efficient search and failure recovery algorithms. There are many different structured P2P topologies (see Fig. 2.3), including:

- Line: The nodes form a line. The nodes towards the middle tend to be more loaded.
- Ring: Data travel through nodes in a circular fashion.
- Fully connected: Each node has a connection to every other node in the network.

Unstructured P2P

In an unstructured P2P topology, the connections between nodes are unorganized and basically random. An unstructured network is easy to build because there is no structure imposed to it. However, it comes with performance limitations. Depending on how the actual graph looks like, some nodes may be flooded by messages coming from several nodes. A node could also end up being a single point of failure, if it is the only way to communicate between two clusters of nodes.

2.3. Communication methods

When communicating within a network, nodes send messages. In order to do that, there are different communication protocols that can be used.

[1]Google Drive - Cloud Storage: `https://www.google.com/drive/`, accessed: 25.08.2018
[2]Microsoft OneDrive: `https://onedrive.live.com/`, accessed: 25.08.2018
[3]Spotify: `https://www.spotify.com/`, accessed: 25.08.2018

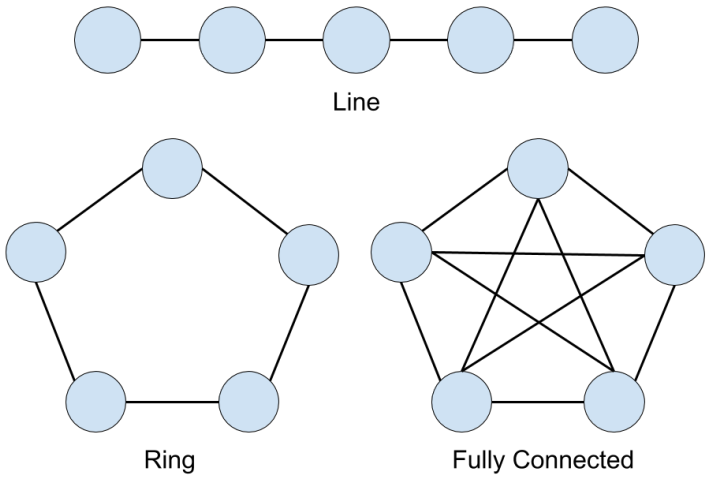

Figure 2.3.: Different network topologies

2.3.1. TCP/IP

The Transmission Control Protocol (TCP), also known as TCP/IP, enables two components to establish a connection with one another. Data can be packetized, addressed, transmitted and received [20]. TCP ensures that data will be delivered safely and in the same order in which they were sent.

To guarantee all these aspects, the TCP/IP model, influenced by the OSI model, is composed of different layers with different purposes.

2.3.2. UDP

The User Datagram Protocol (UDP) is primarily used for establishing low-latency and loss-tolerating connections between applications. It has no handshaking dialogues and there is no guarantee of delivery, ordering, or duplicate protection [21]. Since UDP packets do not transport information to guarantee these aspects, they are lighter.

UDP is ideal for real-time network applications where latency is critical. In gaming and voice and video communications, suffering data loss does not affect the perceived quality as much as in other types of software.

2.4. Network metrics

2.4.1. Delay

The delay of a network specifies how long it takes for a packet to travel from one node to another. Usually, delay is measured in milliseconds (ms). Depending on the geographical spread, the quality of the network, the amount of data sent, the density of the traffic within

the network and more, delay can vary a lot [22, 23]. It is an important aspect to take into account when designing a network application.

2.4.2. Bandwidth

The amount of data a node can receive and send in a range of time is called bandwidth. Due to hardware limitations, the maximal bandwidth a node can handle is limited. Because of this limitation, developers of a network application must reduce the size of travelling packets as much as possible [24].

Bandwidth is typically measured in bits per second. There are many measurement tools that help visualize the bandwidth of a running network application, like WireShark[4] or the Windows Resource Monitor.

[4]Wireshark - Go Deep.: `https://www.wireshark.org/`, accessed: 10.01.2019

Chapter 3.

Introduction to games

Playing is older than culture [25] and it is part of the origins of creativity and its development. By playing, children and young animals grasp a better understanding of their surroundings and their relationships with the outside world [26]. Playing a *game* differentiates itself with merely playing around in that there is a set of rules to respect. Through time and progress in technology, games evolve.

In this chapter, we briefly introduce modern games in general, along with their different types. In the last section, we present a few of many possibilities and impacts of cheating in games.

3.1. Games and video games

3.1.1. The meaning of games

Since games have such a broad definition, they come in different forms and have different purposes.

- Lucrative: Games are most often thought of as pure entertainment. Though the purpose of some games is the fun, they tend to have educating aspects [27, 28], like puzzle-solving, coordination, familiarizing with technology, etc.
- Competitive: By nature, a game must challenge a player's skill [29]. The player needs to train to improve his performance or dominate competitors. A player can show its superiority in specific domains by winning a game against others. Depending on how a game is viewed by the culture, it can be a proof of discipline and hard work.
- Educational: Powerful learning tools can be created by designing a game with the right challenges [30].
- Gamification: Having a playful design improves user engagement [31]. Gamification can be applied to almost every aspect of life: marketing, inspiration, health, work, education, etc.

The meaning of a same game can vary, depending on the context in which it is played. The same game can be played for fun, for money or fame. The stake of the outcome of a game defines how seriously it is to be taken.

3.1.2. Game mediums

Games exist through different mediums supporting the needs of their rules.

No medium

Some games don't need hardware and only exist through their rules and players. An example is the *tag game*, involving two or more players chasing other players in an attempt to simply touch them, usually with their hands. Once a player gets touched, it is his or her turn to chase others.

Physical games

Board games and most sports games require physical objects to interact with. Nowadays, though all physical games get digitized and are playable online, they still keep unique advantages. Board games serve as a good opportunity to meet friends or family and they support having a good time with them [32, 33]. Sports games can be favored over digital games to stay in shape and social.

Digital games

Digitization unlocks tremendous potential in games. Digitized games can:

- replace all existing physical games with the cost of the aspects we stated in the previous paragraph.
- improve existing physical games:
  - Improve the respect of existing rules: In football, video assistant referees start making an appearance in world tournaments[1] like the 2018 FIFA World Cup[2].
  - Simulate other players with artificial intelligence: Chess is playable alone.
  - Playing online across the world
  - Storing scores, publishing scores and competing with others around the world
- enable new ideas and rules that were impossible before.

Though some digital games can exist as an audio form, games as in video games are the most popular form. Depending on the targeted audience, games are playable on personal computers (PC), consoles or phones.

[1] Video Assistant Referees (VARs) Experiment - Protocol (Summary): `https://www.knvb.nl/downloads/bestand/9844/var-handbook-summary`, accessed: 13.01.2019

[2] IFAB comes to landmark decision about VAR: `https://www.fifa.com/about-fifa/news/y=2018/m=3/news=coming-soon-press-conference-following-the-ifab-annual-general-meeting.html`, accessed: 13.01.2019

3.2. Single and multiplayer video games

Whether a game is single or a multiplayer has crucial effects on its nature and impact.

- *Singleplayer* games are games that can be played alone. Input from only one player is expected throughout the course of the gaming session.
- *Multiplayer* games are games playable as more than one person, either locally or over the internet.

3.2.1. Singleplayer genre

The singleplayer game genre gained in popularity with the first arcade games and is now still widely produced for console and PC. A few notable singleplayer titles include the arcade game *Pac-Man* (1980), the world renowned *Super Mario* series (1985-today) by *Nintendo* and its sibling *The Legend of Zelda* series (1986-today), etc.

This genre can focus on a compelling story and character development [34], which can sometimes almost feel like a movie. This means that the focus of singleplayer games is often on the quality of its graphics and story telling.

Because of the absence of other players, the main player plays with the computer. This often means that some sort of artificial intelligence (AI) appears [35, 36]. The quality of the AIs in games is a challenge with a big influence on the player's experience [37].

Though most singleplayer games are offline, there are modern ones that have online features as auxiliary roles. The *Dark Souls* series (2011-today) is very well known for their extremely challenging games which focus on the singleplayer experience. Because of the haunting atmosphere and the punishing nature of the games, the player is bound to feel hopeless [38]. To provide remedy, an innovative and modern idea based on online communication was implemented [39]. Upon death, players leave a ghost behind so others can see how they died and be cautioned not to repeat the same mistake.. You can also leave a mark on the ground before a challenging area and be summoned by a player who needs help. You then enter their singleplayer world which becomes multiplayer and fight monsters together. All these online features are optional and the game can also be played offline.

3.2.2. Multiplayer genre

Since the multiplayer genre bases itself on the interaction between players, it focuses more on gameplay and replayability than on graphics and story telling. Depending on developers' choices and limits, multiplayer games can be played locally or online.

Local multiplayer

The first video games in history were local multiplayer games [40, 41]. The famous *Pong* (1972) was meant to be played by two players in an offline environment. Because of the increasing availability of the Internet in homes[3], this style lost in popularity but is still

[3]Internet Growth Statistics 1995 to 2018 - the Global Village Online: `https://www.internetworldstats.com/emarketing.htm`, accessed: 25.08.2018

viable for home consoles. A widely known game of this genre is *Mario-Kart* (1992-today), though it now also offers online multiplayer modes.

A notable characteristic of this type of game is that the screen is split so that every player sees the same screen and has a part of it attributed to them [42].

Online multiplayer

In online multiplayer games, players can play with and against each other across the world. Because each player uses their own personal computer or console, each player has their own screen for themselves. This means that the screen does not need to be divided like in local multiplayer games.

In order to support both local and online multiplayer or because it makes sense for the gameplay, some online games still split the screen.

The online aspect makes it easy to find new opponents and play unlimited, so players become more and more competent. Elite skills and competitivity in online games are enjoyable to watch for many, which led to the rise of the popularity of *eSports*[4] [43]. Similarly to competitive sports, there can be a huge amount of money in play, based on the outcome of a competitively played video game.

Challenges for online game developers include the following:

- Building up a large enough player base through marketing
- Additional code complexity and implementation costs
- Guarantee the quality of the connectivity between players
- Vulnerability to cheating and cheat prevention

3.3. Cheating in games

Cheating in games can take numerous forms and have different impacts [44]. Here are a few known examples:

- Doping[5] in physical sports to increase performance like running speed, endurance, etc.
- Injuring a player on the opposite team when the referee is not looking, e.g. in football.
- Using different gambling tricks to count cards in poker[6].
- Furtively using cheating software during eSports matches[7].

[4] ESports or electronic sports: competitive, professional video gaming

[5] 10 Biggest Doping Scandals in Olympics History: `https://www.livescience.com/55779-biggest-doping-scandals-in-olympics-history.html`, accessed: 14.01.2019

[6] Phil Ivey Ordered To Pay Back 14 Million Dollars For Cheating: `https://www.huffingtonpost.com.au/2016/12/19/phil-ivey-ordered-to-pay-back-14-million-for-cheating_a_21631160/`, accessed: 14.01.2019

[7] 5 eSports pros who were caught cheating: `https://www.foxsportsasia.com/esports/958816/5-esports-pros-who-were-caught-cheating/`, accessed: 14.01.2019

In video games, cheating mainly means exploiting the software contributing to the game [45] or using external tools. In the next chapter, we present typical cheating techniques in video games and ways to prevent them.

Chapter 4.

Attacker models in games

When there are rules, there are ways to break them. This also applies to software and games. Depending on the nature of the application, having the rules broken by a malicious user can have a serious impact. For games where real money is wagered, like gambling or competitive games, cheating must be prevented. Furthermore, when an online video game leaves too many cheaters in peace, the community will lose interest in that game because of recurring unfairness.

No matter the security put in a software, there is always a way to breach it [46, 47]. It all depends on how motivated the attacker is and how much worth it is to prevent the attack. The core idea in fighting against cheating and hacks is to demotivate the attacker by adding obstacles, even if they are ultimately avoidable.

In this chapter we introduce different existing techniques for cheating in (online) video games and typical ways to counter them. In the last section we look at the additional challenges coming with a distributed architecture.

Cheating in video games is a very vast subject from which we only present an overview here. For more in-depth works on this matter, see [10, 48, 44, 49].

4.1. Editing game files

You can modify the rules or simply delete them by editing the games files on a disk [50]. For example, you can make your character invincible by deleting the code responsible for damage taken from enemies. You can give your character more speed, multiply your score, or even better: win the game instantaneously.

This is doable through reverse-engineering. With the right tools, the executable can be decompiled and source files are generated [51]. One can simply edit the source code as wished and recompile it to create a modified version of the game. One way to prevent this is to use a secured external server to verify the integrity of the game files when starting the game.

4.2. Altering packets

In online games, packets containing data about a player's actions are transmitted. By editing them, one can modify the real version of what happened and possibly break the rules set by the client application.

To prevent this, online games store all their rules on a central server. Clients send request to make actions. Actions are only applied if they are judged legitimate by the server. This solution causes delay and can degrade the player's experience.

4.3. Modifying value in memory

Another way to break rules is to edit the values in the memory at run-time. For example, edit the value of a player's health, its position, the conditions to win, etc.

Again, storing the gameplay sensitive data on an external server solves this.

4.4. Using external tools to improve performance

Even with the right server setting, there are still ways to cheat. If you cannot alter the client software or the packets, you can enhance your performance in a game by using non-allowed tools.

4.4.1. Wallhack and maphack

In some video-games, you are not always supposed to see where the enemy player is. Knowing where the player is or what it is doing has a high value and can often decide the outcome of the game.

In First Person Shooters (FPS), wallhacks are tools that inject code between your process and the DirectX/OpenGL rendering libraries. The goal is to apply transparency to specific materials so that the user can see through walls [52]. This helps the cheater know where an enemy will appear and be ready to shoot even before it is supposed to know its presence.

In Real-Time Strategy (RTS) games, players can adopt different strategies in order to beat their opponent. In top-down views, there is a fog of war that hides areas in the map where you have no units (or paws). Scouting these hidden areas to find out what the enemy is doing is part of the gameplay and strategy. Maphacks simply get rid of that fog of war [53], making the enemy's strategies and whereabouts an open book. The cheater can then adapt to and counter every action the opponent is doing. Maphacking is known to be used by cheaters in the popular games *Warcraft III* (2002) and *Starcraft* (1998-today).

One way to prevent this hack is to not send information to a client that it is not supposed to know of [54]. Do not send the position of a unit if it is in the fog of war. In a FPS game, the pace of the gameplay is so fast that this solution is not viable.

4.4.2. Aimbot

In FPS games, how you are aiming defines your skill. Aimbots are external tools that improve your aiming. When an enemy appears on the screen, the aimbot will make the cheater immediately aim at the enemy's head and shoot. Aimboting is known to be used by many cheaters in the popular games *Counter-Strike* (2000-today) and *Call of Duty* (2003-today).

4.4.3. Farming bots

In some games, the in-game currency has a real-life money value. By using bots, a character will be playing on its own following an artificial intelligence. It will kill enemies, loot in-game money and sell found gear. Farming bots are known to be used in the popular games *Diablo* (1996-today) and *World of Warcraft* (2004-today).

4.4.4. Punishing on top of preventing

There are statistical algorithms that detect if players get too lucky by looking at their previous matches' statistics [55]. The problem with this method is that the player only needs to adjust its tools to cheat a little less, in order to go undetected by the algorithm.

Some games have replay features. After each game, a replay is saved in a compact format. It can be watched in the perspective of different players and with real-time statistics. This way one can review a match from a suspicious player's perspective and observe if its behaviour is suspicious [56]: is he looking at a player through the walls? is its aiming non-human?

Counter-Strike (2000-today) has, on top of a very solid replay feature, a system where players can report suspicious players[1]. If a player has received enough reports, this suspicious player will be reviewed and judged by other players picked randomly in the community. These judges will look at replays from the suspicious player's perspective and then submit their verdict. If the judges submit enough correct verdicts, they receive rewards. That way they are motivated to judge in a fair way. Of course, if a player has been found out to be a cheater, the corresponding account is banned.

4.5. Attacking players

Instead of improving your own performance, you can alter other players' performances. By altering other players' packets, a cheater is performing a man-in-the-middle attack and can neutralize them [57].

Another very well known attack is the distributed denial-of-service attack (DDos attack). It involves sending a lot of traffic to the targeted IP address [50]. The player or server is neutralized because they can only handle a limited amount of traffic.

4.6. Sending a modified high score

Some games base competition solely on the final score that a player reached at the end of a round. This high score is sent from the player's client application to a central server. Of course, a cheater can modify the message containing the high score.

Using algorithms to encrypt the message before sending it and decrypt it from the server-side seems like the most straightforward approach. The problem is that keeping algorithms secret doesn't work [58]: It only takes a small amount of reverse engineering to find out what the code is doing.

[1] Counter-strike: Global Offensive - Overwatch: `https://blog.counter-strike.net/de/index.php/faq-zu-csgo-overwatch/`, accessed: 11.01.2019

A solution would be to not send the high score but all the moves made in the round. Then the server can recompute the round and calculate the final score.

4.7. Cheating prevention in distributed systems

Because of the absence of a central authority in distributed systems, cheating prevention works by making peers work together and counting on the legitimacy of the majority. There are different more or less complex algorithms for this, with different advantages and drawbacks.

4.7.1. Mutual checking

The idea behind mutual checking is that *"you may not trust a single client, but you trust the consensus of multiple unaffiliated clients"* [59]. To ensure the legitimacy of every action, a player will ask other clients before performing any of them. Clients consult each other, make a decision whether this action is legitimate or not, and apply it. We can separate this process in following distinguishable steps:

- Asking for consent: a client wants to act and sends requests
- Voting algorithm: all the clients consult each other and decide whether the requested action is legitimate or not
- Applying consent: the result is spread and the action applied (or not)

The trick in this process is to keep the system consistent and also limit the number of messages sent and computation needed. Also, using a voting algorithm for each gameplay sensitive action increases delay.

Here are a few simple consensus algorithms that require little computing and message exchanges.

The requester is always right

The most trivial consensus algorithm is no consensus algorithm: The node wanting to perform an action just performs it and all other nodes go along with it. This alone does not prevent cheating.

Voting, majority

When asking for an action, the requester asks all other nodes what their decision would be and proceeds to apply the decision that occurs the most. This is democratic. If the majority of players are not cheaters, cheating will be fully prevented. If the majority of players are cheaters, the population is corrupted and cheating may never be prevented and even legitimate requests may get denied every single time.

Random node gives the decision

The decision is made by a random node, alternating at every request. This allows cheating randomly and legitimate requests will be denied randomly if a cheater is in the session.

Arithmetic mean

The decision is the arithmetic mean of the answers of all nodes. This might work in some situations, but often, requests do not make sense in that way. For example, if the goal of a request is to get a single integer between 0 and 10, if we take the arithmetic mean of several nodes' answers, the final decision will always tend towards 5.

4.7.2. Lookahead cheat and lockstep

By purposefully dropping update messages, a malicious player can gain an advantage by receiving information about other players in advance [11]. This technique is called *lookahead cheat*.

We can understand this technique better by looking at the following example: A cheater drops update messages at a time t and waits. It receives incoming messages from another player at time $t+1$ and sees that it would receive an unpredictable, fatal blow. The cheater, still at time t, can react accordingly and send the update messages again. In this example, the cheater defends against an unpredictable blow, using the lookahead cheat.

Countering the lookahead cheat with lockstep

To counter the lookahead cheat, lockstep divides game time into rounds. Every player has to submit their move as an encrypted message before each round. Once all moves have been sent, they are revealed in plain text. That way, malicious players will not receive future update messages before having sent theirs.

The main problem of lockstep is that the game will run at the speed of the slowest player's connection for everyone. For fast paced games, this solution is unacceptably slow.

Chapter 5.

Overview of game network architectures

In this chapter, we present an overview of typical game network architectures. We analyze their characteristics and also mention some notable games that are using them to help us understand the network models in a video-game environment. Furthermore, we will use some of the characteristics in these games to later define meaningful rules for our game.

5.1. The singleplayer network architecture

The most trivial model we want to start with is the singleplayer model. We cannot really talk about a network model here since everything stays local. The goal of this model is to implement and test game features in an easier and faster way than we would in a network model where we have to build and launch multiple instances of the game each time. Also, we can develop the different game features in a modular way, and rework them later for the other network models that we will implement.

If we were talking about non-game softwares, one implementing a "singleplayer" model would be any software that is not using network protocol to fulfill its duty. For example, a standard text editor, a file manager or a classic offline music player.

Taking this general principle and applying it to a video game, it cannot offer the player anything network- or online-related.

AIs in offline mode in online multiplayer games

Counter-Strike (2000-today) is a video game focusing on multiplayer competitive gameplay. When no Internet connection is available, *Counter-Strike* and other online multiplayer games offer the player the possibility to play offline with AIs, also called bots [60].

This means something very interesting for us. This game not only implements a multiplayer mode but also an offline mode, with exactly the same gameplay features, which is what we want to do. To achieve this elegantly, they programmed in such a modular way that gameplay features are independent of the network model used in a session [61].

5.2. The client-server network architecture

The reason such a model is considered the easiest to implement is that the rules of authority are pretty straightforward [62]: the server has the authority and decides for actions that require consensus (see Fig. 2.1). However, for the movement of a player, we can attribute

the authority to the client, who will have control over its own character's movements since, its position synchronization requires no consensus.

This model is less handy to test than the single-player model. We have to start multiple instances of the same game, create a game as a host with one instance and join the game as client with the others.

To test out latency issues, these game instances should be started on different machines. One could also modify the network capacity using an external tool, e.g. *clumsy*[1].

5.2.1. Centralizing sessions

An underestimated advantage gained from using a centralized server is the ability to have a centralized place where all players can come and find other players to play with [63].

The method to find other players can vary. In the early days of multiplayer gaming, games made use of server lists. Anyone with the right settings can create a server for others to join. The created server is then to be found on a centralized server list that can be browsed by anyone possessing the game.

Nowadays, players get lazy to search for a session themselves, so the matchmaking solution is preferred: The player picks the game mode he or she wants and is placed into a queue waiting for other players who are looking to play the same mode. This is simpler to use but can lead to long waiting times if the player base is too small.

5.2.2. Preventing cheating with a server

Ideally every action completed by a client should be controlled by the server. This would, however, pose some latency issues for real-time and speed sensitive actions. There are different techniques that allow a smooth synchronization for such actions.

Generally, there is only one action in a real-time online game that requires to be so smooth that it cannot wait for a consensus (server confirmation) before being applied on the screen, but still needs to be synchronized: a player's movement. When a player moves its character, the most important aspect is fast responsiveness of the inputs for the controls to feel handy. Also, the values of the position vector are precisely modified to a point where with each change, mostly separated by a few milliseconds, there is a high chance of no impact on the gameplay for other players.

Now there is a problem with not asking for consensus. The player could cheat by modifying the client software and go wherever it wants, even into forbidden areas in the game. To prevent that, the server verifies at regular intervals if the player's position matches where it should be. If not, the server corrects the position and the player jumps back to a previous correct position.

Another technique is to not synchronize the players' positions but the inputs. For example, when another player presses the Forward key, we start making him move forward, until we receive the message that he released the Forward key. This requires less synchronization messages but can lead to imprecision and desynchronization: If a single input is lost during transmission, different players will not see the same game state on the screen anymore.

[1] clumsy, an utility for simulating broken network for Windows Vista / Windows 7 and above: `https://jagt.github.io/clumsy/`, accessed: 19.12.2018

5.2.3. Dedicated servers

Dedicated servers are powerful servers that can be rented from providers. They offer the benefits of high performance, security and control.

They are relativel expensive. For gaming purposes, the prices range from around 60 to 200 dollars per month, depending on the bundle chosen[2]. Therefore, games using dedicated servers need a return on investment that is large enough to cover these costs.

As a solution for the single point of failure problem, certain game companies add back up servers in case the main one fails. However, with this solution, additional server costs ensue.

5.2.4. Simulated client-server system via P2P

To avoid paying servers for a game because the player base is too small or irregular, or simply to cut costs, a compromise can be made: Still implement a central server for the matchmaking but once players enter one game, promote one player to play the role of the server and force the other players to become clients and to connect to it in P2P.

There are two major drawbacks with this technique. Firstly, most of the disadvantages of the P2P network model are inherited. The player promoted to server may be highly unreliable and, as a single point of failure, end up fatal.

Secondly, a way to bypass the firewall of the promoted server when connecting to it is necessary. This can be done by making a so-called NAT Punch-through[3]. This method was used in the strategy game *Heroes of Delum* (2018) and also the popular game *Starcraft: Remastered* (1998-2017) [61].

Having a central server, even outside game sessions, has its advantages. As we already established, it can be used for matchmaking so that players have a centralized place to find each other. That server can also be used to prevent cheating. This can be done by verifying the legitimacy of the game files content each time the executable of the game is launched by the player.

5.3. The distributed P2P network architecture

The most important characteristic to keep in mind in a distributed system is that there is no central server. There are only clients (or nodes), all having the same rights.

Age of Empires

Age of Empires (1997) is a popular real-time strategy game that uses a P2P architecture. Peers exchange data directly with each other. When an invalid command is received, it simply gets dropped. To prevent inconsistency, peers exchange their game states periodically. Since the game state of a cheater will differ from the one of all other players, the cheater can be detected and removed [49].

[2]Game Servers - Dedicated Servers: `https://www.ovh.com/world/dedicated-servers/game/`, accessed: 17.01.2019

[3]NAT Punch-through: `http://www.raknet.net/raknet/manual/natpunchthrough.html`, accessed: 17.01.2019

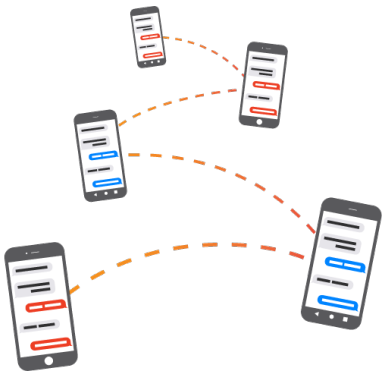

Figure 5.1.: FireChat, a P2P messaging application

The first problem with this solution is the heavy network load coming from the entire game states exchanges. The second problem arises when there are more malicious players than honest players, in which cases honest players will get removed from the game.

As *Age of Empires* was one of the first commercial games to use a P2P architecture, several deficiencies got exposed. To counter them, the game had to be updated multiple times[4].

FireChat

FireChat is a messaging application that allows users to contact each other without needing an internet connection[5]. Using Bluetooth, you can directly send a message to someone within a range of 70 meters. If that person is further away, the message is passed via a chain of phones (that must have *FireChat* installed) until the target is reached (see Fig. 5.1).

Because *FireChat* does not depend on an internet connection, it can be used during sports games, rallies, music festivals, and in emergency situations.

With the example of *FireChat*, we can see how the technical choice of a network architecture can reveal the hidden potential of a known type of software.

Multiplayer games via Bluetooth

Following the idea of *FireChat*, we know that it is possible to play the same game on different devices, without an internet connection and using Bluetooth [64]. In most cases though, a player will simulate a server on its device and create a client-server topology and not a distributed P2P one. This is favored because of the simplicity of the implementation and because it does not require a cheat-proof software in such a close-ranged network.

[4]How to Hurt the Hackers: The Scoop on Internet Cheating and How You Can Combat It: `https://www.gamasutra.com/view/feature/131557/how_to_hurt_the_hackers_the_scoop_.php`, accessed: 21.01.2019

[5]How To - Open Garden: `https://web.archive.org/web/20161119044307/https://www.opengarden.com/how-to.html`, accessed: 18.01.2019

Chapter 6.

Defining our game

For our scope we aim to define a game that is enjoyable but simple enough to avoid unnecessary complexity.

6.1. Principle

Our game is called *Rocket Lanes*. It can be played as two, three or four players. There are four lanes on the screen, one attributed to each player (see Fig. 6.1).

One player controls the movement of a single spaceship, trapped in its lane and has to avoid incoming rockets.

Every player can send rockets to another player's lanes to aim to hit their spaceship.

A player can also cast a shield on its own spaceship to be immune to the next collision with a rocket.

The goal of the game is to keep your own spaceship intact and eliminate every other players' spaceships, using rockets.

To reduce the complexity of the game, we left out the ammunition feature that is presented in the mockup. The idea was that the player could buy ammunition and make its spaceship shoot on incoming rockets.

6.2. Rules

6.2.1. Spaceship

A player can move its spaceship with the WASD keys. It cannot move outside its lane. When the spaceship hits a rocket, the rocket explodes and the spaceship takes one damage. The spaceship will start with five health. At zero health, it explodes and the player loses.

6.2.2. Gold

Each player receives three gold every second. Gold can be used to send rockets or cast a shield.

6.2.3. Rockets

Sending a rocket costs five gold.

A player can send as many rockets as fast as wanted.

Figure 6.1.: Mockup of Rocket Lanes

When a player sends a rocket, it will be spawned on the first non-empty lane on the player's right. If the player is on the far right of the arena, the search begins from the far left.

The rocket will be spawned on the top part of the targeted lane, at a random X coordinate.

6.2.4. Shield

Using a shield costs 20 gold. A player can only use a shield every 15 seconds. A shield lasts two seconds and absorbs the damage of colliding rockets.

6.3. Different actions and states

There are three types of actions that will be implemented and synchronized in a different way between the players.

6.3.1. Actions that require consensus

These are the most important and cheating-sensitive actions. In our case, joining a game, sending a rocket and casting a shield are the actions that require the consensus of the other players (or the server). In other words, when a player wishes to complete one of these actions, he first "asks" the other players (or the server) if he is allowed to do so.

6.3.2. Actions that require synchronization without consensus

These are less gameplay impacting states. In our case, the spaceships' positions won't require consensus.

The first reason is that the position of a spaceship is refreshed too often and will lead to latency.

The second reason is that it is not that important if there is an inconsistency in one player's position, meaning if it is not synchronized perfectly.

6.3.3. Actions that require no synchronization

Actions that do not require synchronization should not have an impact on the gameplay. In our case, these are special effects, sound effects, UI content, etc.

6.4. Justifying the game rules choices

These game features and rules were chosen to fit our experiment as good as possible to.

- Similar to a local multiplayer game, all players are visible from the same screen. In our experiment, this is helpful to observe latency and players' behaviors.
- Our game features compose all types of actions: actions that require consensus, actions that require synchronization without consensus and actions that require no synchronization.
- The gameplay is very simple to reduce complexity and the graphics are minimalistic to enhance clarity.
- Also, the game does not exist yet, so we keep a level of creativity and uniqueness that is encouraged in the game development field.

Chapter 7.

Implementation

In this chapter, we describe the tools used, the conception and implementation of our game using three different network models. Details for each network architecture can be found in the appropriate sections.

7.1. Concept

Rocket Lanes, the game we develop, has some fundamental characteristics to keep in mind during the implementation. *Rocket Lanes* is:

- a video game
- a lucrative game
- a multiplayer game
- an online game

This game is unique since it uses both the client-server and the distributed network architectures. Usually, an online game just sticks to one online network architecture.

Finally, since this game only exists for the sake of the experiment, it is not necessary to make it enjoyable. This reduces complexity a bit.

7.2. Tools

The most notable tools we used for this project are the Unity Engine, its UNet features for client-server networking and its Transport Layer API for P2P networking.

7.2.1. Unity

Unity is a multi-platform game engine developed by Unity Technologies. The first version came out in 2005 and since then five major versions have been released. Unity gained in popularity, mostly because of its interesting cross-platform development tools and because it is free. It is regularly updated and the community is big. Unity targets 27 platforms and is already working with virtual and augmented reality[1].

Unity allows scripting in C# in a component-based environment.

[1] Industry-leading multi-platform support: `https://unity3d.com/unity/multiplatform`, accessed: 25.08.2018

7.2.2. UNet

Unity provides very handy tools supporting client-server multiplayer which we are going to use for our client-server implementation.

The `NetworkManager`[2] component simplifies the work, so that the developer doesn't have to manipulate TCP sockets. It manages server hosting, clients joining and spawning, entities synchronization, etc. One can easily customize network behaviour and adapt it to the game's needs by overriding the `NetworkManager` component.

This method is the most popular and is recommended by Unity Technologies until this day, though they announced in August 2018 that they are working on a new networking layer[3].

Network Identity

To synchronize the state of objects in a scene between clients, Unity offers the `Network Identity` component. The developer can adjust parameters like the network sending rate which represents the rate at which the server will send this object's information to clients (aka. tickrate).

Simply by adding this component to an object, its position and rotation will be synchronized.

Remote Procedure Calls

Unity supports Remote Procedure Calls (RPC)[4]: The server can remotely call a function on the client's application.

It is crucial that it is the server that imposes a function call instead of the client doing it on its own locally. That way the server can verify the legitimacy of an action before authorizing it and thus prevent cheating.

7.2.3. Transport Layer API

The Transport Layer API provides a thin networking layer on top of UDP sockets. It can:

- simulate a connection with other peers and detect when one connects or disconnects
- provide reliable and unreliable channels so that no additional implementation for acknowledgement (ACK) or packet failure of important messages is needed
- send and receive messages asynchronously
- make use of serialized messages

Learning to use the Transport Layer API takes time, but given the advantages provided, this tool makes working with P2P networking significantly less painstaking.

[2] Using the Network Manager: `https://docs.unity3d.com/Manual/UNetManager.html`, accessed: 17.09.2018

[3] UNet Deprecation FAQ: `https://support.unity3d.com/hc/en-us/articles/360001252086-UNet-Deprecation-FAQ`, accessed: 13.12.2018

[4] Remote Actions: `https://docs.unity3d.com/Manual/UNetActions.html`, accessed: 13.12.2018

7.3. Setting up the environment

7.3.1. Installing Unity and creating GitHub repository

First, we downloaded and installed the latest version of Unity on Windows[5]. Then we created a repository on GitHub[6].

In order to synchronize a Unity project correctly on a Git repository and avoid uploading huge sized engine files, an appropriate .gitignore file is necessary. It contains the following:

```
.vs
/Build
/Library
/Temp
/*.userprefs
/*.csproj
/*.unityproj
/*.sln
/obj
/.vscode
```

7.3.2. Creating a mockup

To help us visualize the game and its features, we created a mockup early on (see Fig. 6.1). We also took the opportunity to export the different sprites in order to have resources that are ready to be used and implemented in the game.

7.4. Software architecture

7.4.1. Assets

When developing a video game, it is not only important to structure the code but also all the used resources, also called assets. There are all types of assets: textures, sound effects, musics, scripts, particle effects, scenes, materials, etc. A good standard structure is to arrange them according to their type (see Fig. 7.1). More information can be found in Appendix B.

7.4.2. Network

With its component-based design, Unity forces the developer to adopt an object-oriented programming behaviour which is meaningful when developing a game. In our case, we needed to think very thoroughly about how we wanted to structure the network part. A common mistake would have been to start developing the game without considering the network and only adding it in the end. This would have required a refactoring process so laborious that it may have made more sense to recode the whole software.

[5]Download - Unity: `https://unity3d.com/get-unity/download`, accessed: 13.08.2018
[6]Rocket-Lanes GitHub repository: `https://github.com/alexisdavidson/Rocket-Lanes`: 25.01.2019

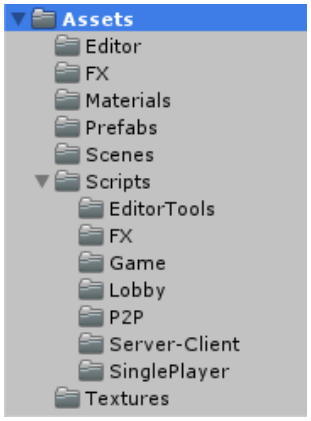

Figure 7.1.: Tree view of assets folders in Unity

Therefore, from the beginning we needed to consider how we want to implement three different network models that act on the very same game.

The gameplay features should not depend on which network model is being used. This means that if we want a rocket to be spawned from a gameplay logic, only one function will be called and not one for each network model. This forced us to use an interface and each implementation of that interface represents a different network model.

This is the interface we ended up with:

```
//INetworkController.cs
public enum ConsentAction {SpawnRocket, CastShield, JoinGame};

public interface INetworkController
{
    bool Initialize();
    void Quit();
    void AskForConsent(ConsentMessage consentMessage);
    void ApplyConsent(ConsentMessage consentMessage);
    bool HandleCollisions(Lane lane);
}
```

Ultimately, each network model differentiates itself based on one of these methods: the initialization and shutdown, asking for consent, applying a consent and handling collisions. The `ConsentMessage` class contains an id parameter, a `consentAction`, a result and a list of `int` parameters. It also has methods for serialization and deserialization that are used when sending and receiving a message through the network.

The `ConsentAction` enumeration lists the different actions that require consent: spawning a rocket, casting a shield and joining a game.

7.5. Singleplayer

We started by implementing the singleplayer model. The goal was to define all game components so that we could later focus on the other network models solely. Since we had a

well-thought interface, the game logic would not be lost or deprecated when implementing them.

We would also be able to have very fast results in terms of gameplay so we could test it out.

7.5.1. Player and player controller

An instance of the `Player` class represents a player in the scene, whether it is the current user, or others connected to the same game sessions. The important attributes of this class are health and position. We attached a sprite of a spaceship to the `Player` game object, so each player is represented by a spaceship in the scene.

The current user needs a way to control the `Player` instance attributed to him. For that, we attached a `PlayerController.cs` script to it, which handles the keyboard inputs of the user and allows him to move his spaceship.

7.5.2. Lane

A Lane contains a spawning position for a player, boundaries (walls), and spawning positions for the rockets. It also has a unique id and a unique color.

Important for the gameplay is to know whether a lane is occupied by a player or not. An unoccupied lane means that a player died there or disconnected, or it was never occupied in the first place. Either way, no rocket should be spawned there and the lane should be ignored in terms of gameplay.

7.5.3. Rocket

When a rocket is spawned, it simply moves forward until it collides with a wall or a player. Handling the collision of a rocket depends on whether we have the authority or not. In a client-server network environment, this would be the task of the server. This is why we check the authority before making any collision operation.

If the rocket collides with a player, it loses health and the rocket is destroyed. If it reaches the bottom wall, the rocket is destroyed without causing damage to any player.

7.5.4. Shield

When a shield is cast, a spaceship is invulnerable for two seconds. This means it won't lose health when colliding with a rocket. To be able to cast a shield again, 15 seconds must pass.

7.5.5. Gold

We chose not to implement a currency system in the game, because it would have added a complex feature that is unnecessary for the purpose of this master thesis. Therefore, there are no limits to the number of rockets sent and shields cast.

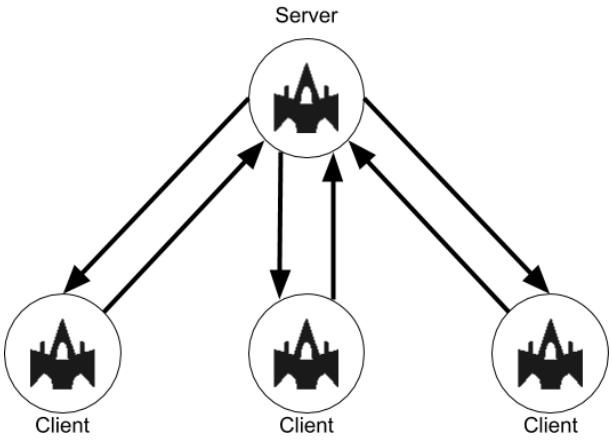

Figure 7.2.: Client-server topology: The server is also a player

7.5.6. Game controller

The `GameController` class acts as a singleton and stores references of the user's `Player` instance, the lanes and the network controller. The `GameController` class has handy methods, e.g. finding the next occupied lane and gameplay methods, e.g. sending a rocket. Sending a rocket is done by calling the `AskForConsent(...)` method of the current network architecture and passing through the right parameters.

7.5.7. Artificial intelligence

To be able to simulate a game session, we created a simple AI for the computer players in single-player mode. At random intervals of time, it picks a random direction to move along. Also, at random intervals of time, it sends a rocket to the next occupied lane.

7.5.8. Result

As expected, implementing the single-player network architecture proved to be pretty straightforward. All games rules were done and testable in a single-player mode, which is an important aspect for gameplay testing.

We hoped that for the next network models, we would only need to adapt the implementation of the interface and modify the other scripts as little as possible. That way we could solely focus on the network part.

7.6. Client-Server architecture

In this architecture, a player has the role of the server and the other players are the clients (see Fig. 7.2). This is because we are not using dedicated servers, otherwise players would only come as clients.

Unity provides different components to facilitate the implementation of a client-server network architecture. We will now introduce them and explain how we use them for different

purposes.

7.6.1. Network Manager

The `NetworkManager`[7] is a handy class that provides simple ways to start and stop client and servers, to manage scenes, and has virtual functions that user code can use to implement handlers for network events. We created a class that inherits the `NetworkManager` class and also implements our network interface.

7.6.2. States

We need to think about what states there are, what actions can be made and what information exchanges are needed in order to keep everything consistent between players.

Since *Rocket Lanes* is not a turned-based game but a real-time one, the number of possible states tends towards infinity. Also, the spaceships' and rockets' positions have a high precision hence a very vast domain. The only way to approximate consistency is to synchronize all gameplay-sensitive objects like rockets and spaceships and gameplay-sensitive data like a player's health, across clients and as fast as possible. Assuming the client-server architecture, Unity provides tools for this problem.

7.6.3. Network Identity

Unity's Network Identity component gives a game object a unique identity and controls and synchronizes its attributes on the network. We can choose which attributes to synchronize and at which rate by adding the `Network Transform` component (see Fig. 7.3). To achieve a smooth result, we chose to synchronize position nine times per second, or every 111 ms.

We gave a `Network Identity` component to the Player and the Rocket prefabs and we synchronize only the `Rigidbody 2D` component which represents the position and rotation of a game object. In other words, the Unity network system will take care of the position synchronization of the players and rockets on its own. Also, Unity uses linear interpolation in order to smooth out the movements of objects.

7.6.4. Local player authority

As we established earlier, it is meaningful to give a client authority on its player character. This way the motions feel more fluid. We did it by setting the `LocalPlayerAuthority` attribute to `true` in the Network Identity component (see Fig. 7.3).

By marking the `Health` attribute as `[SyncVar]`, the health of a player is automatically synchronized among all.

7.6.5. Connection and spawning

When a client joins the server, the server first checks if the game is not already full. If it is, the clients is removed and receives an error message. Otherwise, the server finds the next

[7] NetworkManager: `https://docs.unity3d.com/ScriptReference/Networking.NetworkManager.html`, accessed: 24.09.2018

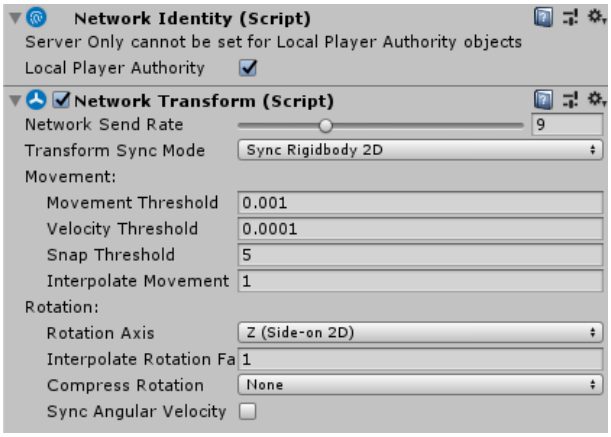

Figure 7.3.: Network Identity and Network Transform components

unoccupied lane which will be where the player spawns and we instantiate a player prefab in the position defined by this lane. In simpler words, when a client joins the server, the server spawns, on the right lane, a player object upon which the client will have authority.

This pseudo-code shows the simple handling of incoming connections on the server-side:

```
OnClientConnect(client):
    if(GameIsFull()):
        Kick(client)
    else:
        SpawnPlayer(FindNextFreeLane())
```

If a player tries to connect to a non-existing server, a timeout error is received after a few seconds and the connection is canceled.

7.6.6. Asking for consent

Asking for consent happens by sending a message to the server. This message contains a serialized `ConsentMessage`, describing which consent action we want to ask for and containing a varying numbers of parameters.

7.6.7. Receiving a consent request

When the server receives a message, it analyzes which consent action was requested and the parameters. Then, depending on the result of its decision, the consent is applied or not.

```
OnRequestReceived(request):
    if(request.IsValid()):
        ApplyRequest(request)
```

No message has to be sent back since because it will automatically be synchronized among clients once the server applies an action.

7.6.8. Applying consent

```
ApplyRequest(consentMessage):
    if(consentMessage.consentAction == ConsentAction.SpawnRocket):
        lane = gameController.lanes[consentMessage.parameters[1]]
        NetworkServer.Spawn(lane.Spawn(consentMessage.result))
```

In this example the client requested to spawn a rocket. To apply this request, we verify if the current application is the server and we spawn it to the corresponding lane. This occurs by calling the method `Spawn`[8] which handles the spawning of network objects adequately.

7.6.9. Collisions

Only the server handles gameplay-sensitive collision detection and reactions. These are deactivated client-side. Server-side, when a collision happens, the destruction of the rocket happens and the touched player loses health. This is automatically synchronized among clients.

7.6.10. Messages and phases count

Connection

When a client connects, a handshake between the client and the server happens. The client then receives information about the game session and the game starts. We do not know how many messages or phases the handshake takes. However, if we count the client connecting and the server giving information back, we have *2* messages and *2* phases.

Position synchronization

A client sends its position to the server (*1* message), the position is sent to all other clients (*n-2*). This is a total of *2* phases, with *n-1* messages.

Consent request

The client sends a request to the server (*1* message), the server answers to all (*n-1*) clients. This is a total of *2* phases and n messages.

7.6.11. Run examples

Here is an example of a session scenario, server-side:

```
- Server starts
- Game starts
- while(...):
    - Play alone: move spaceship, spawn rockets, cast shields
- Client connects, allow him and spawn a spaceship for him
- while(...):
```

[8] Spawn: `https://docs.unity3d.com/ScriptReference/Networking.NetworkServer.Spawn.html`, accessed: 24.09.2018

```
    - Play: move spaceship, spawn rockets, cast shields
    - Receive requests to spawn rockets and cast shields, allow them (or not)
    - Receive positions synchronization from client
- Client disconnects
- Server stops
```

We create a server and play alone until a client joins. We play with the client. The client leaves and we leave.

Here is an example of a session scenario, client-side:

```
- Try to connect to server...
- Timeout
- Try to connect to server...
- Game is full
- Try to connect to server...
- Authorized
- Game starts
- while(...):
    - Receive other players' positions
    - Receive rocket spawning and shield casting information
    - Send own spaceship's position synchronization
    - Send rocket spawning and shield casting requests
- Disconnects
```

We try to join an non-existing server and get a timeout. We try to join a full server and get an error. We finally join a server authorizing us to play. We play, then leave.

7.6.12. Result

The implementation of the client-server architecture was straightforward. It only took a few days to get a conclusive result and no real problems were encountered. The simplicity of this implementation was astonishing because most of the needed synchronization problems were covered up by Unity's features, which were perfectly documented.

The game runs very smoothly and there do not seem to be any networking problems whatsoever. Since all gameplay-sensitive actions are verified and performed by the server, cheating is completely ruled out for clients.

7.7. Distributed architecture

This implementation was much more complex than the client-server's one. The states and the actions that can be executed are similar to the client-server case but implemented in a whole different way. This is why we had some more planning to do before attacking the coding part.

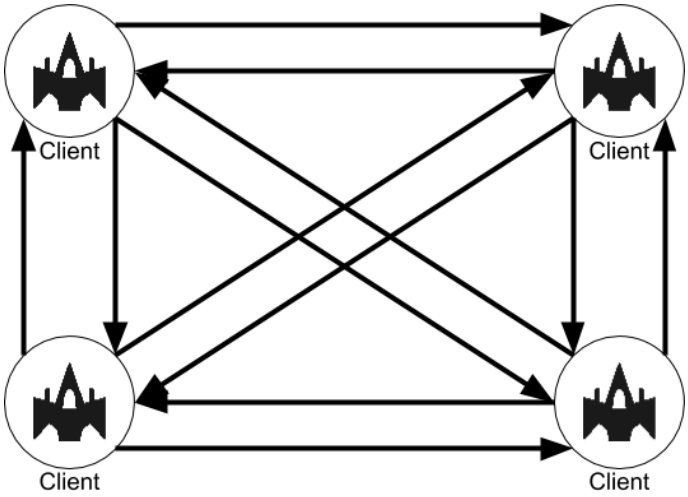

Figure 7.4.: Fully connected topology

7.7.1. Topology

In real-time applications, it is important that messages are exchanged as fast as possible. There is no better way to do that than sending data directly to the node it is addressed to. This is why we chose the *fully connected topology* (see Fig. 7.4) where all nodes are connected to each other. To achieve that, when a node joins the network, it needs to receive information about all other nodes in order to be connected to them.

7.7.2. Mutual checking and voting

To reduce the cheating rate, we chose to use mutual checking. This means that the legitimacy of every move is verified through a vote. By doing that we trust in the majority of clients.

The reasons we chose this method are: (i) guaranteed consistency; (ii) moderate complexity; (iii) applicability on real-time games.

7.7.3. Implementing the consensus algorithm

At any time, a node can ask other nodes for consent, specifying which consent action is to be taken and some parameters. Upon receiving such a request, depending on which consent action is to be taken, a node computes his answer and sends it back:

```
OnAskForConsent(consentMessage):
    if(consentMessage.consentAction == ConsentAction.PlayerJoin):
        /* ... */
    else if(consentMessage.consentAction == ConsentAction.SpawnRocket):
        /* ... */
    AnswerConsent(consentMessage, parameters)
```

When receiving an answer, the initial node that made the request stores an answer until it receives enough or until timeout. Then, it can apply the decision, or not. All nodes must

be informed so they can apply the decision as well.

```
OnAnswerConsent(consentMessage):
    Addvote(consentMessage)
    if(ReceivedEnoughVotes(consentMessage):
        SendToAll(MessageType.ApplyDecision, consentMessage)
        ApplyDecision(consentMessage)

ApplyDecision(consentMessage):
    if(consentMessage.consentAction == ConsentAction.PlayerJoin):
        /* ... */
    else if(consentMessage.consentAction == ConsentAction.SpawnRocket):
        /* ... */
```

In a game with n players, upon each request, *n-1* messages are sent out to make the request. *n-1* messages are received back containing the other nodes' decision. *n-1* messages are sent out with the final decision to be applied. This makes a total of *3n - 3* messages exchanged during *3* phases.

A few cases still need to be clarified:

- In case a node is failing, we added a timeout to that request: After x ms, even if all nodes did not answer, a decision is taken.
- If the amount of votes is even for different decisions, we apply the one of the requester.
- If a player is alone in a game session, it has no one to ask consent to hence applies directly its own decision

7.7.4. Connection manager

Keeping track of the nodes the current instance is connected with is a must. However, there are other complications that must be addressed. What if a node connects, but the game is full? What if a node connects but fails to connect to one of the other nodes in the game? The `Connection Manager` class we implemented handles all that.

Connection object

The `Connection Manager` class contains a list of `Connection` objects, representing a connection to another node. It contains its host id and a connection id, specific to the Transport Layer API, its IP and port, its lane and a flag indicating whether this node has fully connected to the network or not.

Joining the game

When a new node tries to join the game by connecting to a node in the network, it must first receive a confirmation that the game is not full. To achieve that, the node in the network which received the connection request will send a consent request, asking other nodes if this new player can come in or not. In other words, if the game is full or not.

```
/* The node receiving the connection requests asks other nodes for consent */
OnConnectReceived(connection):
    connectionManager.Add(connection)
    AskForConsent(ConsentAction.PlayerJoin, gameController.PlayersCount)

/* The asked nodes answer with their opinion, in this case, the number of
players already in the game */
OnAskForConsent(consentMessage):
    if(consentMessage.consentAction == ConsentAction.PlayerJoin):
        AnswerConsent(consentMessage, gameController.PlayersCount)
```

The result is then sent back to the new node, along with the information of the other nodes so that it can connect to them.

```
OnAnswerConsent(consentMessage):
    if(ReceivedEnoughVotes(consentMessage):
        ApplyDecision(consentMessage)

ApplyDecision(consentMessage):
    if(consentMessage.consentAction == ConsentAction.PlayerJoin):
        finalResult = TakeMajorityResult(consentMessage)
        if(finalResult < gameController.MaxPlayersCount):
            AllowNodeToJoin(connectionManager.connectionsInformation)
```

Only once the new node has established a connection to all the nodes in the network, it announces its success. It is then flagged by other nodes as *successfully connected* and finally takes part in the game.

```
OnAllowedToJoin(connectionsInformation):
    foreach(connection in connectionsInformation.connections):
        Connect(connection)

OnConnectedToAll():
    foreach(connection in connectionManager.connections):
        Send(MessageType.SuccessfullyConnected)
    gameController.StartGame()
```

Similarly to a timeout, if all of this does not happen after a couple of seconds, the new node is removed and the connections with it are broken.

7.7.5. Synchronizing positions

Since position synchronization does not require asking for consent, it is easier to implement. Each node simply sends the position information of its spaceship to all other nodes, and they apply it. We use the same sending rate as in the client-server implementation: nine times per second, or every 111 ms.

7.7.6. Messages and phases count

Position synchronization

A node sends its position directly to all other nodes. This happens in *1* phase and *n-1* messages.

Consent request

A node sends a request to all other nodes (*n-1* messages). They all respond (*n-1* messages). Then the decision, if there is any, is spread (*n-1* messages). This happens in *3* phases and *3n-3* messages.

Connection

When a client requests to connect, a message is sent (*1* message). The node asks consent to the other nodes (*2n-2* messages). When the result is here, it sends information about the game to the client (*1* message). The client then tries to connect to every other node (*n-2* messages). Once this is done, the game starts. All of this happens in *5* phases and *3n-2* messages.

7.7.7. Using the Transport Layer API

At first, using the Transport Layer API was a challenge. The documentation is confusing and the community using this API is almost non-existent, making it very repelling. It took days to get a single result confirming that this API would be working for our goals.

Other than that, the advantages listed previously were substantial and saved us a lot of headaches.

7.7.8. Run examples

Here is an example of a session scenario:

```
- Create new game
- Game starts
- while(...):
    - Play alone: move spaceship, spawn rockets, cast shields
- Node(s) connect(s), allow it(them)
- while(...):
    - Play: move spaceship, ask for consent about spawning rockets
    - Receive requests to spawn rockets and cast shields, give answer
    - Receive positions synchronization from other node(s)
- Quit
```

We create a new game and nodes join and play. In the end we quit.

Here is a second example of a session scenario:

```
- Join existing game
- Connect to all nodes
```

```
- Game starts
- Play
- All other nodes disconnect
- Game continues
- Node connects
- Quit
```

We join an existing game, when suddenly all other players quit the game. The game still continues even though we didn't create it and other nodes can still join us.

7.7.9. Result

Here is a list of the noteworthy cons from the distributed network architecture for *Rocket Lanes*:

- Planning, implementing and finalizing took months
- The API's documentation is bad because this technique is practically not used
- The code base is extremely complex and big
- The movement synchronization is not perfectly smooth
- The actions have a small delay
- Because of its complexity, the implementation may contain bugs
- It is not cheat proof

And here are the pros:

- No single point of failure
- No server cost
- Any player can leave and the game goes on

7.8. Conclusion

Overall the implementation went without too many problems and results were seen relatively fast. The single-player's implementation went flawlessly, the client-server one almost so. Despite experiencing a clearly slower progress for the implementation of the distributed architecture, overcoming challenges one at a time proved satisfying and motivating.

Using Unity and its features helped save time and headaches tremendously. Sometimes, however, bugs did appear and locating the roots of the problem was a hard task since they came from the Unity engine itself. Many times we had to upgrade Unity or rollback to a previous version where that specific bug did not exist. In other cases, we had to make workaround fixes in order to solve the engine's problems. A particularity was that we implemented both P2P and client-server technologies and that caused a fatal network

problem on the project. Since almost nobody implements both these network architectures on the same project, finding help in the documentation or by the community was impossible. Hence, we had to investigate the fix ourselves.

In the end, even if the implementation took longer than expected, all the goals were met and everything planned was successfully implemented.

Chapter 8.

The experiment

8.1. Characteristics to measure

Now that we implemented both client-server and distributed network architectures, we need to measure some data in order to compare them with each other. The characteristics we are interested in are the following:

- Number of messages sent
- Number of messages received
- Outgoing bandwidth
- Incoming bandwidth
- How much time has passed until a request is applied
- Allowed cheating percentage for consent requests

Number of messages sent and received

Each time a client or node **receives** a message through the network, the number of received messages is iterated.

Each time it **sends** a message through the network, the number of sent messages is iterated.

Outgoing and incoming bandwidth

Because we used tools within Unity to implement the client-server architecture, we have less low-level control over its code. Therefore, the measuring of certain metrics is not trivial. This is why we use an external tool to measure the outgoing and incoming bandwidth, the *Resource Monitor* on Windows.

How much time has passed until a request is applied

When a client or a node sends a request, we store the current timestamp along with the request id. Upon receiving confirmation of the specific request, we can save the amount of time that passed between the moment the request was sent and the moment the request was applied.

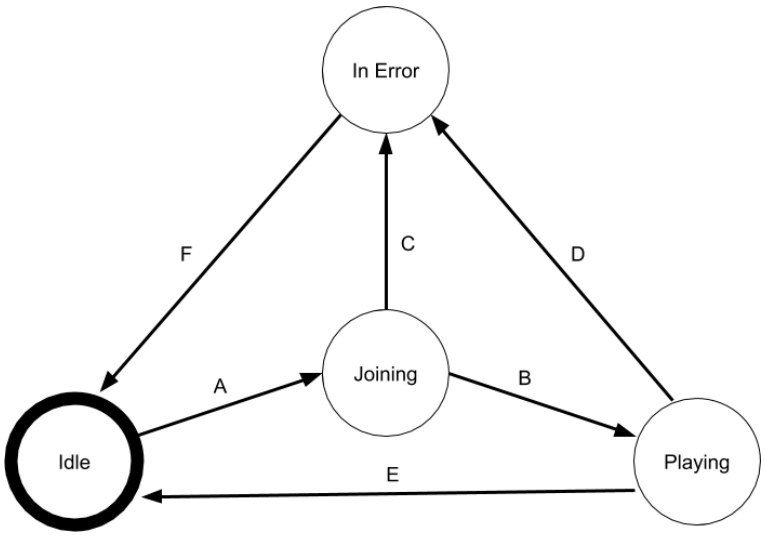

Figure 8.1.: States graph for an automated run

Allowed cheating percentage for consent requests

Each time a node sends a request to perform an action, we added the possibility that it might be a non-legitimate request or in other words, an attempt to cheat. The parameters of that action contain values that are not allowed by the rules. For example:

- A node will try to send a rocket in a non-existing lane
- A node will try to cast a shield too soon after the previous cast

In the client-server architecture, these requests are verified by the server and immediately rejected if they are not legitimate. In the distributed architecture, nodes take votes and only if the majority of votes is against a request it gets rejected.

We flag a cheating request in order to track it and when the request gets approved we increment the amount of cheating passed. If the request gets rejected and the cheat prevented, we increment the number of unsuccessful cheating attempts.

8.2. Automated runs

If we want to generate enough data, we need to run the simulations several times. The best way to do that is to write an algorithm that automates runs while recording data.

We wrote a state-based algorithm that does just that. Before starting an automated run, we need to set up whether the current instance will use the client-server architecture or the distributed one. In the client-server case, we also need to choose whether the current instance will play as a client or a server.

A run starts in the `Idle` state (see Fig. 8.1). Then it tries to join (or create) a game. This can fail, in which case the state `In Error` is entered, or succeed, in which case the

Transition	A	B	C	D	E	F
Delay (s)	2 - 4	1	1	20 - 40	20 - 40	1 - 3
Action	Join or create a game	Add AI component to player	-	-	Reload scene	Reload scene

Table 8.1.: State transitions

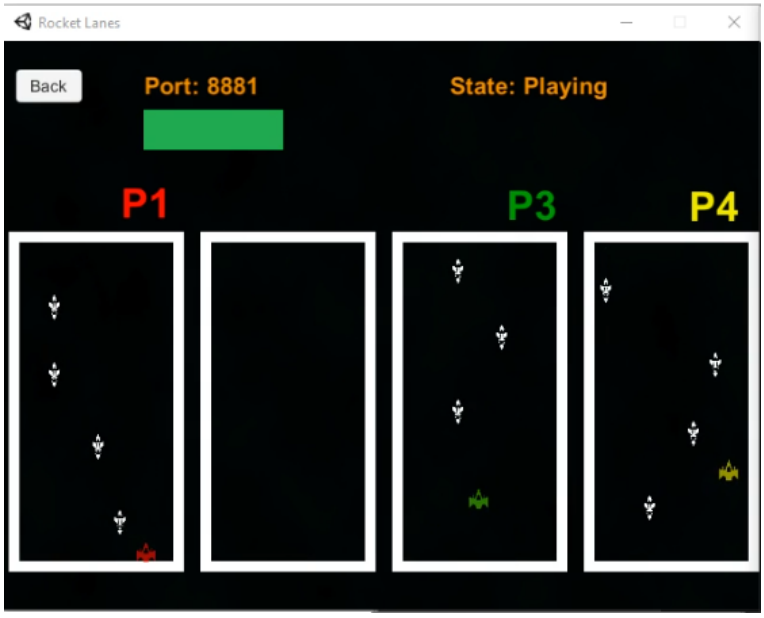

Figure 8.2.: A single instance running an automated distributed simulation

state `Playing` is entered. Both these states are followed up by the `Idle` state again, except the `Playing` state that can also be followed up by the `In Error` state.

Each state transition is characterized by the execution of an action and occurs after a specific range of time. See Table 8.1 for the specific actions and delays.

On a single instance is displayed which port is used and which state it is in. Each game has a unique color generated at creation in order to visualize who is playing with whom. This color is shown below the port (see Fig. 8.2).

8.3. Data recording

The data to be recorded are the parameters that we listed at the beginning of this chapter, except for the outgoing and incoming bandwidth (see next section). They are measured with counters manually coded in the software. When *data recording* is enabled in the main menu of the software, data will start being recorded immediately once a game is started and will stop when the game is finished. Once stopped, a *json* file is generated and placed in the `/NetworkData` folder beside the executable. At the end of the experiment, we will have a

Instances (players) count	Running time	Latency	Cheat attempt rate
4	1h	No latency (+0ms), Normal latency (+50ms), High latency (+500ms)	33%

Table 8.2.: Parameters for the experiment runs

lot of generated files (one for each game session) that we will need to put together and from which we will compute average values.

8.4. Hardware specifications

These are the hardware specifications of the desktop computer the experiment will run on.

- Operating system: Windows 10 Education
- Processor: Intel(R) Core(TM) i5-7600K CPU @ 3.80GHz 3.79GHz
- RAM: 16,0 GB
- System type: 64-bit, x64-based processor
- Graphic card: NVIDIA GeForce GTX 1060 6GB

We do not expect computation time to get in the way of network measurements.

8.5. Run settings

The different parameters we chose for the experiment are shown on the Table 8.2.

8.5.1. Instances count

We need to choose how many players (or instances) will take part in the experiment. The more players, the busier the network.

We chose four instances.

8.5.2. Running time

This parameter only affects the amount of data we have at the end. The more the better. It is essential to use the same time for each runs if we wish to compare them afterwards.

We ran each for one hour, which corresponds to approximately 400 game sessions of 30 seconds among all instances that are constantly joining and leaving.

8.5.3. Latency settings

Depending on the nature of the game and the implementation of the network code, the effect of latency can vary [65]. Since we are running the experiment on the same machine, locally, we experience almost no latency. To put our implementation in more realistic conditions, we can modify our network capacity using an external tool, *clumsy*[1]. This way, we can visualize what happens when packets drop, packets take a long time to arrive, packets are duplicated or erroneous.

We will make runs with: no latency, a normal latency (50 ms delay), a high latency (500 ms delay).

8.5.4. Cheating

To test out if cheats pass through in distributed (we assume that they don't pass in client-server), we programmed the AI to send consent requests of non-legitimate actions. For example, a node may wish to send a rocket outside a lane, or cast a shield before it is ready to be used again. The numbers of tried cheats and the number of successful cheats are counted and stored.

8.5.5. Special rules

To ensure that rockets are sent and shields are cast as consistently as possible, we choose to resurrect dead players after 10 seconds.

For the distributed experiment, an instance will start a game and never leave it. Other instances will join it and leave it. This way, we get a very similar experiment to the client-server one where the server creates a game and never leaves it. We will be able to compare data in a meaningful way by doing this.

8.6. Starting the experiment

8.6.1. Client-server run

We start the executable the right number of times to have the number of instances wished. For each one of them, we set the role as *Server* and for all the other ones as *Client*. Soon enough, a game is created and clients joined. Different instances are playing with each other, joining and leaving the game regularly.

8.6.2. Distributed run

We start the executable the right number of times in order to have the number of instances wished. For each one of them, we set which is the listening port, what range of ports are to be targeted and finally press start. Different games are created, players join each others and leave regularly. In Fig. 8.3, we can see four instances running and playing together.

[1] clumsy, an utility for simulating broken network for Windows Vista / Windows 7 and above: `https://jagt.github.io/clumsy/`, accessed: 19.12.2018

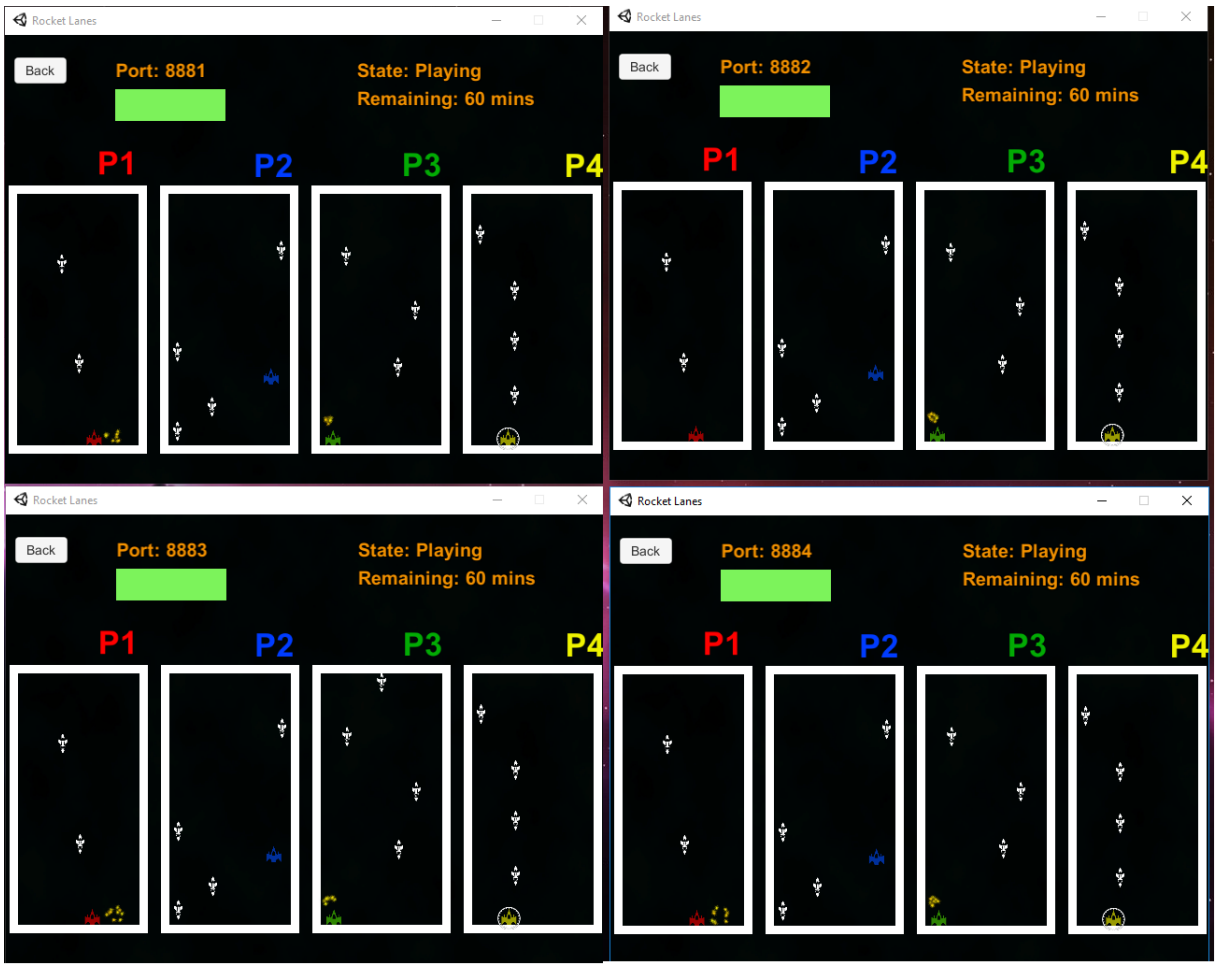

Figure 8.3.: Four instances running an automated distributed simulation

As there is no way to know which instances are playing, when an instance wants to join a game, it must randomly choose in a pool of ports, which one to join. This is why, during the simulation, we often see that an instance tried to join a non-existing game and therefore is in the *In Error* state.

Once the time we choose to run the simulation for is elapsed, we simply close everything.

Chapter 9.

Results analysis

In this chapter, we visualize the data gathered from the experiment, specifically the attributes mentioned in the previous chapter. After we interpret them, we discuss other pros and cons that are not readable from the collected data.

All collected data represent an average of multiple game sessions with a duration of approximately 30 seconds.

We study the client, the server and the node from the distributed architecture separately.

9.1. Number of messages

The average amount of messages, important messages, consent requests sent and received during a game session of 30 seconds, with four players, are shown in Fig. 9.1.

Messages sent and received

The client sends 303 and receives 303 messages. The server sends 1102 and receives 889 messages which is approximately three times more than the client's values. We notice that the server sends more than it receives. This is due to the fact that the server is itself a player and sends information of itself to other clients as well as the information received from other clients.

In the distributed architecture, a node sends 943 and receives 943 messages. Those numbers are the same, which underlines the fact that each node has the same functions - no node sends or receives more than another node. We notice that a node sends approximately 150 fewer messages than a server but 640 more than a client. It receives more messages than a server and more than three times as much as a client.

Important messages sent and received

A client sends 29 and receives 29 important messages. These numbers are the same. These important messages contain consent requests and connection requests.

A server sends 315 and receives 101 important messages. These important messages contain consent decisions and connection messages. For each important message received, the server needs to send it back to all other clients. This is why the server sends more important messages than it receives.

A node sends 245 and receives 247 important messages. These important messages contain consent requests, consent decisions, consent opinions, connection requests and connection

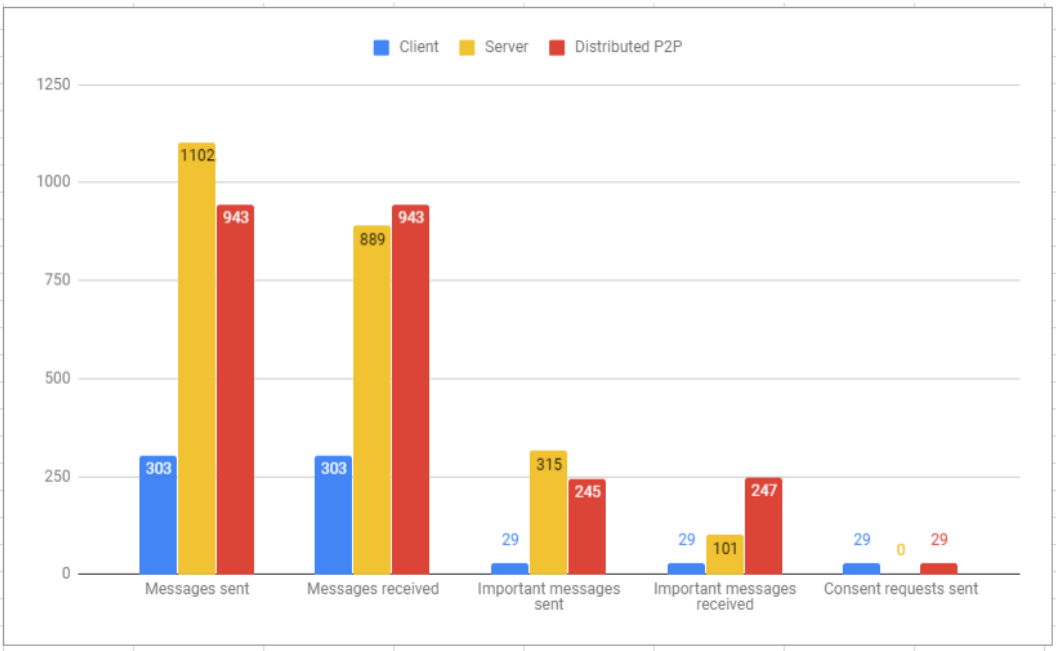

Figure 9.1.: Messages, important messages and consent requests sent and received depending on the network architecture of an instance

answers. These numbers are higher than the ones in the client-server architecture because of the voting system of the distributed architecture that requires a lot of message exchanges.

Consent requests sent

A node and a client sends the same amount of consent requests. It makes sense since these are the product of a gameplay component: sending rockets and casting a shield. Since both clients and nodes have no total authority over these actions, they need to request consent all the same.

The server though does not send any consent request. When the server wishes to send a rocket or cast a shield, it just applies it.

9.2. Bandwidth

For the client-server architecture, we can see on the graph of the Resource Monitor (see Fig. 9.2) that the average network activity stagnates around 100 Kbps (kilobits per second). We notice that the instance running the server has a higher send rate, with an average of 4378 B/s (see Fig. 9.3). The clients' instances have a send rate between 468 and 506 B/s. The server's instance has an average receive rate of 1443 B/s and the clients' instances have an average receive rate between 1212 and 1308 B/s.

We notice small peaks when a player requests to join the game, and negative peaks when there are less than four players in the game.

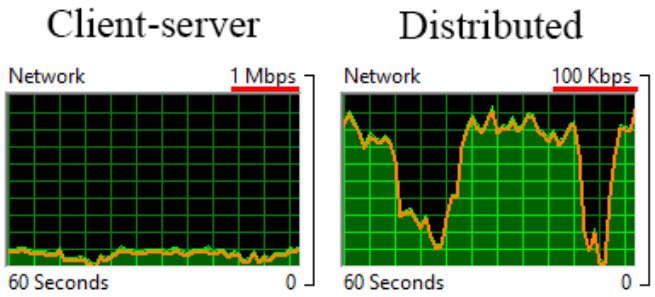

Figure 9.2.: Bandwidth graphs visualized by the Resource Monitor. Note that the two graphs have been scaled differently by the recording software, but have the same magnitude.

Client-server

Processes with Network Activity

Image	PID	Send (B/sec)	Receive (B/sec)	Total (B/sec)
Rocket Lanes.exe	5000	4.378	1.443	5.821
Rocket Lanes.exe	6608	506	1.308	1.814
Rocket Lanes.exe	11204	468	1.241	1.709
Rocket Lanes.exe	556	468	1.212	1.681

Distributed P2P

Processes with Network Activity

Image	PID	Send (B/sec)	Receive (B/sec)	Total (B/sec)
Rocket Lanes.exe	5000	1.102	1.095	2.196
Rocket Lanes.exe	11204	1.011	1.017	2.028
Rocket Lanes.exe	6608	992	991	1.984
Rocket Lanes.exe	556	990	990	1.980

Figure 9.3.: Sending and receiving rates visualized by the Resource Monitor

Overall, the server's network bandwidth is much more loaded than the client's one.

For the distributed architecture, we can see on the graph of the Resource Monitor (see Fig. 9.2) that the average network activity stays between 10 and 100 Kbps. It is important to note that the graphs look very different for the two architectures only because the Resource Monitor from Windows automatically scales it. In reality, both graphs have the same magnitude.

We can clearly make out where players leave and join. At the 18 seconds mark, we can see that two players left and joined again quickly after that. At the 50 seconds mark, we can see that in a very short period, no packets are exchanged, meaning that no player is currently in the same game as another one.

According to the Resource Monitor, the sending rate varies between 990 and 1102 B/s and the receiving rate varies between 990 and 1095 B/s (see Fig. 9.3). These values are in total slightly lower than the ones of the client-server and that is a surprise. We expected the

distributed bandwidth to be more loaded because of the additional messages coming from the voting system, but this is not what we see.

We need to detect what type of messages the bandwidth consists of to understand why. Position synchronizations' messages are sent nine times per seconds per player and consent messages only when spawning a rocket, casting a shield or when a player joins. This means that in reality, position synchronization messages take a much more important part of the bandwidth than the consent messages. We can visualize how much by looking at the graph: The small variations come from the consent requests, and the big peaks and negative peaks from the position synchronization messages.

Now in the distributed architecture, the position synchronization messages are sent directly to another, and in the client-server architecture, they first need to pass through the server, and then to other clients. This consists of more phases and it is why we see a higher total bandwidth for the client-server implementation than for the distributed one.

9.3. Time until answer

This attribute determines how much time passes between the moment a gameplay-sensitive action is requested by a player and the moment it is applied. For example, when a player wishes to send a rocket, how long until it appears on the screen, due to networking delay?

In Fig. 9.4 we can see the average time passed until a message is received, depending on the latency settings, for a client and a node in the distributed architecture. Since the server always applies an action without requesting it, this attribute is always 0 and is not shown on the graph.

On low latency (0 ms delay), the client waits 46.9 ms and the P2P node 64.6 ms. On normal latency (50 ms delay), the client waits 163.9 ms and the P2P node 180.4 ms. On high latency (500 ms delay), the client waits 1063.7 ms and the P2P node waits 1189.6 ms. We notice that the P2P node always waits slightly longer than the client. Also, when the latency gets higher, the difference gets bigger. These numbers stay relatively close though, since requests happen in two phases in both architectures. In P2P the request is sent to multiple instances, so the marge for error and latency is bigger. Hence, the slightly higher average time until an answer is received.

9.4. Cheating

The server has a 100% cheating success rate because since it has complete authority. A client has a 0% cheating success rate, since all requests pass through the server that has complete authority.

For a node in the distributed architecture, for 24.4 cheats tried, 5.69 passed. This represents a 23% cheating success rate.

9.5. Implementation costs

We estimate the costs of the implementation by measuring the code complexity of a network architecture and by the time spent on it. We stored the number of tickets attributed to a

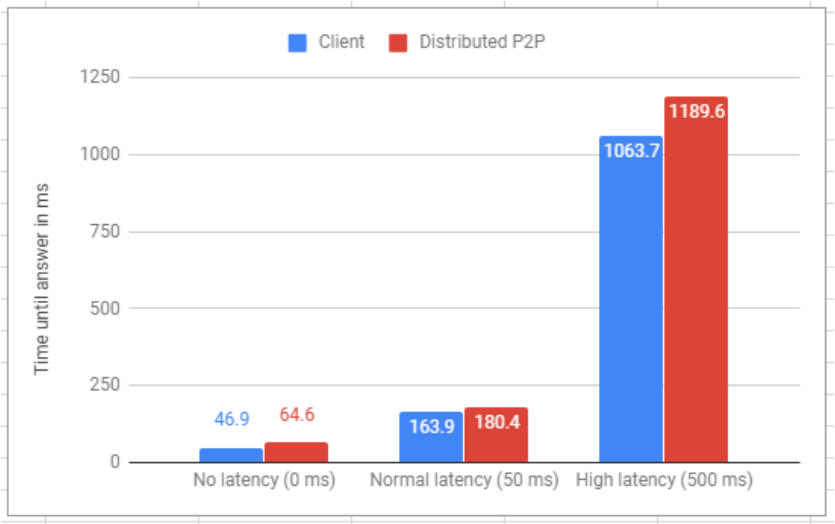

Figure 9.4.: Average time passed until a message is answered

task, the number of commits to the repository, the file count, the line count and the time spent on the implementation of a network architecture (see Fig. 9.5).

- 15 tickets were created for client-server specific tasks while 80 were created for the distributed architecture.
- 35 commits were pushed for the client-server implementation and 90 for the distributed implementation.
- There are 4 script files and 300 lines of code for the client-server implementation and 20 files and 1200 lines of code for the distributed implementation.
- two weeks were spent on the implementation of the client-server architecture and two months were spent on the implementation of the distributed architecture.

These are approximate values and should never speak for themselves. In this case, they only support the case that implementing a distributed architecture cost more than a client-server one, as the difference is clearly visible and for every single aspect. The implementation distributed has values three to four times greater than those of the client-server implementation.

9.6. Other aspects

9.6.1. Server costs

Obviously there are no server costs when using the distributed architecture. However, this applies for the client-server one as well since we are not using any dedicated server.

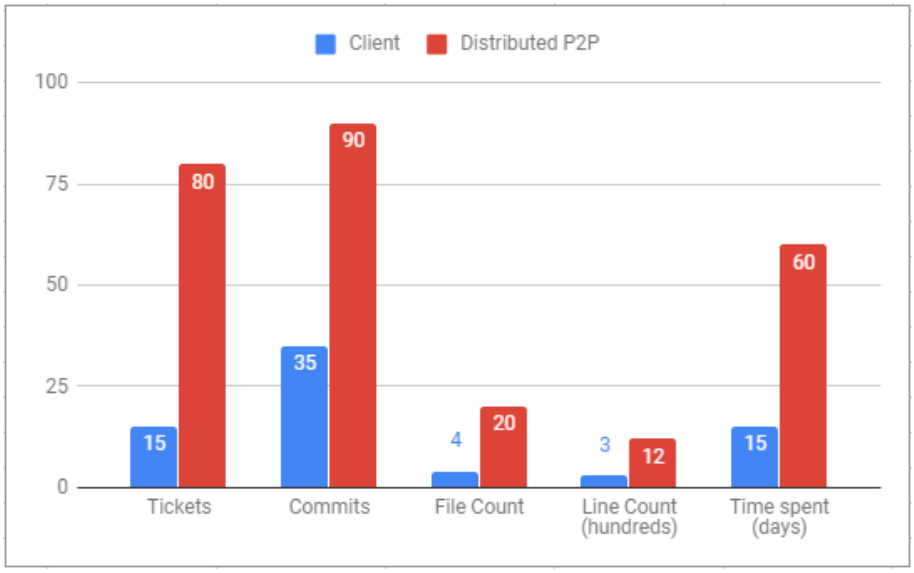

Figure 9.5.: Code complexity and time spent on the implementation

9.6.2. Single point of failure

With the client-server topology, when the server fails or simply leaves, the game is immediately over. This behaviour is common in multiplayer video games.

With a distributed topology, if a node fails, it leaves, but the game goes on among the other nodes. The failing node could theoretically join again and continue playing. This behaviour is not something we see in multiplayer video games at all. In theory, there could be a batch A of players that get replaced by a batch B, totally different but continuing the same initial game. That would mean that a game state could be prolonged in a distributed way, only maintained by peers that relay each others. Eventually, when all peers leave, that specific game session is lost forever.

9.6.3. A single player with high latency

In the client-server topology, when a client has a high latency, he will receive information with a delay and experience difficulties to play smoothly. If the server has a high latency, every single player in the game session is affected.

In the distributed topology, if a node has high latency, every node will experience difficulties in playing smoothly. This is mainly because we use a voting system where we vote for every node to answer (until a timeout). This could be fixed by kicking a node from the network if it has a too high latency.

9.6.4. Miscellaneous features

Some notable network features are given when using the client-server tool from Unity. These would be interesting to implement in our distributed version as well but represent a significant amount of work, meaning significantly bigger implementation costs.

	Drops	Duplication	Delay	Out of order	Tampering
Client-server	no effect	no effect	no effect or timeout	no effect	disconnect
Distr. P2P	no effect	no effect	no effect or timeout	no effect	failure

Table 9.1.: The effects of packet manipulation

Linear interpolation

When synchronizing entities' positions, Unity applies linear interpolation. This makes movements seem smooth even if some frames are faked. Not having this in our distributed implementation makes spaceships stutter and the gaming experience less pleasant.

State recovery via snapshot

When joining a game, the server sends a snapshot of the game to the client. In their screen, all rockets and players are spawned instantly.

9.6.5. Packet manipulation

Using external tools, we tried the effects of packet manipulation on different network architectures in our game (see Table 9.1). It is no surprise that the Unity tools for the client-server model handles them without problem.

Thanks to the Transport Layer API, the packet drops are handled by using a reliable channel when sending an important message.

The duplication has no effect since we handled it in our consent manager.

The delay has no other effect than delaying an action. If the delay is too high, timeouts happen.

Packets arriving out of order also have no effect.

Altering the content of a packet can have fatal consequences though. This might be easily fixed, as we verify the content of each packet before using it.

Chapter 10.

Conclusion

Implementing the same video game in both client-server and distributed architectures proved to be an interesting and unique experience. There is no other reason to do this other than an academic one. Only because of that, the Unity Engine showed some difficulties: It doesn't expect both network technologies in the same project, and since nobody does that, this specific problem had not risen yet.

The singleplayer implementation went flawlessly and the client-server one, because of astonishing networking tools, almost so. However, the distributed part was a series of challenges.

The first one arose before writing a single line of code: The Transport Layer API, the preferred tool for UDP communication in Unity, has a bad, unclear documentation and an almost non-existent community. Most issues were dealt with by finding isolated projects using the same API. Even in the README section of such a project it is stated that *"The UNet Transport API is poorly documented and badly explained in many places"*[1].

But once we started to use the API correctly, it proved to be a mandatory tool, taking care of amazing features like simulating connections with peers, providing reliable channels, sending and receiving messages asynchronously and making use of serialized messages.

While this tool helped us a great deal in the end, the implementation costs were still of another magnitude than for the client-server one. We estimate the time spent and the complexity to have been three to four times greater for the distributed implementation than for the client-server one.

Now this means that choosing the distributed architecture over the client-server one could only be worth it if the resulting advantages were significantly outweighing the cons for the video game we aim to develop.

10.1. Weighing the pros and cons from the experiment

Since each video game is different, there is an artistic potential for uniqueness and the goals developers aim to achieve vary. However, there are scientific and economic aspects that are universally crucial to any application with networking: consistency, a small network load, a fast responsiveness, stability, low costs and no cheating.

[1] A full example of using the low level UNet transport API in unity3d: `https://github.com/shadowmint/unity-example-unet-transport`, accessed: 03.01.2019

Consistency and stability

In our experiment, consistency and stability were achieved for both network architectures.

Network load

Unexpectedly, the distributed version showed a slightly lighter network bandwidth than the client-server version. This was a surprise. We expected the distributed system to be heavier because of all the consent request messages. The results of the experiment showed that the network load mostly consisted of the position synchronization messages that are sent nine times per second. In the distributed topology, these messages go directly to other peers while in the client-server topology, they first have to go to the server then to the other clients, all of which consist in having to go through one more phase.

On top of that, the player running the server is heavily burdened with additional computation and networking load in comparison to clients. This could make it uneven between players, depending on the hardware they are running on.

Responsiveness speed

Because distributed has more phases for requesting and applying an action, the responsiveness is not as fast as in the client-server case.

Also, if a single node has high latency, the whole voting process takes longer and actions are delayed for every player. This could be solved by taking votes in order to kick nodes that are having high latency.

Cheating prevention

Our experiment showed that a cheating node in the distributed system has a 23% cheating success rate. For the client-server architecture, the client has a 0% cheating success rate and the server 100% because it has complete authority. Dedicated servers set up by the developers will of course not cheat but will cost.

It is difficult to say if another voting algorithm could have reduced the cheating success rate in the distributed system. It is common sense that if in a game session there are more cheating players than legitimate players, the game has slim chances to make any sense or be fun. This also applies to board gaming: if the majority of players does not agree with the rules you think are true, there is no way of imposing them. In fact, it could even be that your rules are outdated or wrong.

Matchmaking

Matchmaking is a process that finds and brings players together so that they can enter the same game session. If you play with friends at home, you obviously do not need to find players, but if you want to play on the internet with random players, this is mandatory. Matchmaking is simply not possible with a distributed topology. A centralized instance is needed to store players that are looking for a game and bring them together. It could be a website, an API or even a piece of paper with a list of IPs and ports. We could think of

some sort of Bluetooth scanning process finding players in the area, but this is an extremely restricting condition.

Single point of failure

In a distributed topology, there is no single point of failure. There is no server that can fail and interrupt the game. If any node fails, it leaves the network and the game goes on for the other players. This is a behaviour that is not known in the multiplayer video-game world when no dedicated servers are in play. But what we can make of it depends heavily on the nature of the video-game to be developed. Is it really such a huge problem to have a single point of failure, when failure in reality so rarely happens? And are the consequences of a single failure that bad in the context of lucrative video-gaming?

10.2. Possible solutions

If a game company has the confidence that their new online game will have a solid player base and a large enough return on investment, the right move is to rent quality dedicated servers with backups. They have high performance, security and prevent cheating. On top of that, they have the ability to provide matchmaking features and do not put additional network loads or computation on players' personal computers.

Complications arise when a game company does not have the capital or the confidence in their success to invest in dedicated servers. Compromises must be made in order to find the most adapted solution to satisfy the game's needs.

First of all, in order to enable a matchmaking feature, one could rent a very cheap or free web server. This would also allow the possibility to verify the legitimacy of the games' files when a player launches it and thus reduce cheating by a bit.

Now for the in-game, real-time part comes the question whether a client-server or a distributed topology is to be adopted. The answer to this question heavily relies on the nature of the game. Here are a few examples:

- In a First Person Shooter, the game is fast paced. Delay must be reduced as much as possible: the distributed topology would be too slow because of the additional delay ensuing from the voting algorithms.

- In poorly optimized games, players' computers experience heavy computation. With a client-server topology, the player running the server would be overwhelmed by the additional computation.

- In online games with a lot of simultaneous players, a player's server would be overwhelmed by the network load and would not scale well.

- For an ordinary game with few simultaneous players like *Rocket Lanes*, our experiment showed that the game is playable in both topologies. The most significant difference appears to be the additional implementation costs for the distributed architecture, which makes it not the best suited option.

The key is to find a game concept that can put up with the limitations of a specific network architecture. In order do to that, the process of deciding which network model to adopt must happen together with the defining of the game rules.

Cheating prevention without dedicated servers

When not using dedicated servers, it is difficult to say whether it is best to have a host player that can cheat at a 100% rate (client-server topology), or the fact that players can cheat if not the majority of them are honest (distributed topology).

One way to help against this dilemma would be to make an honest player leave game sessions that its client application does not trust: If a server or other peers refuse the legitimate requests of the honest player too many times, or create moves that are not legitimate, the honest player marks the server or the other peers as untrustworthy and leaves the game.

10.3. Further research

To continue this work, it would be meaningful to implement, in the distributed system, the functionalities that came with the Unity's client-server tools as given: linear interpolation, dropping a player with high latency, snapshots and state recovery, packet merging and verifying the content of a packet in case it is altered.

With a cheap or free web server, one could add a launcher that enables matchmaking and that verifies the legitimacy of the game's files to limit cheating. One could also build a centralized reputation system, storing honest and malicious users based on players' votes.

Finally, one could try to find an innovative idea that actually benefits from the distributed system and that would be impossible with the client-server topology.

Bibliography

[1] Hai Zhang. Architecture of network and client-server model. *CoRR*, abs/1307.6665, 2013.

[2] Amir Yahyavi and Bettina Kemme. Peer-to-peer architectures for massively multiplayer online games: A survey. *ACM Comput. Surv.*, 46(1):9:1–9:51, July 2013. ISSN 0360-0300. doi: 10.1145/2522968.2522977.

[3] Mark B. Gagner. *Peer-to-peer distributed gaming application network.* 2003.

[4] Andrew S. Tanenbaum. *Maarten van Steen. Distributed Systems, Principles and Paradigms.* Prentice Hall, 2007.

[5] Lars Braubach and Alexander Pokahr. *Addressing challenges of distributed systems using active components.* Springer, 2011.

[6] Shirin Nilizadeh, Sonia Jahid, Prateek Mittal, Nikita Borisov, and Apu Kapadia. Cachet: a decentralized architecture for privacy preserving social networking with caching. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 337–348. ACM, 2012.

[7] Kanaka Juvva. *Real-Time Systems.* 1998. 18-849b Dependable Embedded Systems.

[8] William Sims Bainbridge. The scientific research potential of virtual worlds. *Science*, 317 5837:472–6, 2007.

[9] Robert Orfali, Dan Harkey, and Jeri Edwards. *Client/server survival guide.* John Wiley & Sons, 2007.

[10] Christian Mönch, Gisle Grimen, and Roger Midtstraum. *Protecting online games against cheating.* 2006.

[11] Nathaniel E Baughman, Marc Liberatore, and Brian Neil Levine. Cheat-proof playout for centralized and peer-to-peer gaming. *IEEE/ACM Transactions on Networking (ToN)*, 15(1):1–13, 2007.

[12] Patric Kabus and Alejandro P Buchmann. *Design of a cheat-resistant P2P online gaming system.* 2007.

[13] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design.* Addison-Wesley Publishing Company, USA, 5th edition, 2011. ISBN 0132143011, 9780132143011.

[14] Greg R Andrews. *Foundations of Parallel and Distributed Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999. ISBN 0201357526.

[15] Sandeep Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. *Logical Physical Clocks.* 12 2014. ISBN 978-3-319-14471-9. doi: 10.1007/978-3-319-14472-6_2.

[16] C Zaschke, Barbara Essendorfer, and C Kerth. *Interoperability of heterogeneous distributed systems.* 05 2016. doi: 10.1117/12.2223895.

[17] J. D. Thomas and K. Sycara. *Heterogeneity, stability, and efficiency in distributed systems.* July 1998. doi: 10.1109/ICMAS.1998.699069.

[18] B. Clifford Neuman. *Scale in Distributed Systems.* IEEE Computer Society Press, 1994.

[19] Robert Stroud. Transparency and reflection in distributed systems. *Operating Systems Review*, 27:99–103, 04 1993. doi: 10.1145/506378.506426.

[20] Behrouz A. Forouzan. *TCP/IP Protocol Suite.* McGraw-Hill Higher Education, 2nd edition, 2002. ISBN 0072460601.

[21] Jon Postel. User datagram protocol. Technical report, 1980.

[22] Atef Abdelkefi and Yuming Jiang. *A Structural Analysis of Network Delay.* 05 2011. doi: 10.1109/CNSR.2011.15.

[23] Li Zhang, Zhen Liu, and Cathy Xia. *Clock synchronization algorithms for network measurements*, volume 1. 02 2002. ISBN 0-7803-7476-2. doi: 10.1109/INFCOM.2002.1019257.

[24] Antonio Capone, Luigi Fratta, Fabio Martignon, and Student Member. Bandwidth estimation schemes for tcp over wireless networks. *IEEE Transactions on Mobile Computing*, 3, 08 2004. doi: 10.1109/TMC.2004.5.

[25] Johan Huizinga. *Homo Ludens Ils 86 - Nature and significance of play as a cultural phenomenon.* Routledge, 1949.

[26] Donald Woods Winnicott. *Playing and reality.* Routledge, 2012.

[27] Marc Prensky. *Don't bother me, Mom, I'm learning!: How computer and video games are preparing your kids for 21st century success and how you can help!* Paragon house St. Paul, MN, 2006.

[28] Marc Prensky. J computer games and learn| ng: D 1 g ital game-based learn 1 ng. 2007.

[29] James Paul Gee. What video games have to teach us about learning and literacy. *Computers in Entertainment (CIE)*, 1(1):20–20, 2003.

[30] David R Michael and Sandra L Chen. *Serious games: Games that educate, train, and inform.* Muska & Lipman/Premier-Trade, 2005.

[31] Sebastian Deterding, Dan Dixon, Rilla Khaled, and Lennart Nacke. *From game design elements to gamefulness: defining gamification.* 2011.

[32] Margaret Hofer. *The games we played: The golden age of board and table games.* Princeton Architectural Press, 2003.

[33] Yan Xu, Evan Barba, Iulian Radu, Maribeth Gandy, and Blair MacIntyre. *Chores Are Fun: Understanding Social Play in Board Games for Digital Tabletop Game Design.* 2011.

[34] Tison Pugh. The queer narrativity of the hero's journey in nintendo's the legend of zelda video games. *Journal of Narrative Theory*, 48:225–251, 07 2018. doi: 10.1353/jnt.2018.0009.

[35] Adil Khan. *A Competitive Combat Strategy and Tactics in RTS Games AI and StarCraft.* 09 2017. doi: 10.13140/RG.2.2.34444.21121.

[36] A J. Champandard. Ai game development: Synthetic creatures with learning and reactive behaviors. 01 2019.

[37] Priya Rana, Parthik Bhardwaj, and Jyotsna Singh. Artificial intelligence (ai) in video games. *International Journal of Computer Applications*, 181:1–3, 09 2018. doi: 10.5120/ijca2018917818.

[38] Serge Petralito, Florian Bruehlmann, Glena Iten, Elisa Mekler, and Klaus Opwis. A good reason to die: How avatar death and high challenges enable positive experiences. 05 2017. doi: 10.1145/3025453.3026047.

[39] Tom van Nuenen. Playing the panopticon: Procedural surveillance in dark souls. *Games and Culture*, 11, 02 2015. doi: 10.1177/1555412015570967.

[40] Dawn Spring. Gaming history: Computer and video games as historical scholarship. *Rethinking History*, 19, 04 2015. doi: 10.1080/13642529.2014.973714.

[41] M.J.P. Wolf. *Before the crash : Early video game history.* 01 2012.

[42] Heonsik Joo. A study on split screen according to the form classification of visual media. *Journal of the Korea Society of Digital Industry and Information Management*, 11:131–139, 06 2015. doi: 10.17662/ksdim.2015.11.2.131.

[43] Anders Hval Olsen. The evolution of esports: An analysis of its origin and a look at its prospective future growth as enhanced by information technology management tools. 09 2015.

[44] Steven Daniel Webb and Sieteng Soh. *Cheating in networked computer games: a review.* 2007.

[45] Gary McGraw and Cigital CTO. *Exploiting online games: cheating massively distributed systems.* Addison-Wesley, 2008.

[46] William R Cheswick, Steven M Bellovin, and Aviel D Rubin. *Firewalls and Internet security: repelling the wily hacker.* Addison-Wesley Longman Publishing Co., Inc., 2003.

[47] Gary McGraw. *Software security: building security in*, volume 1. Addison-Wesley Professional, 2006.

[48] Steven Webb, Sieteng Soh, and William Lau. Racs: a referee anti-cheat scheme for p2p gaming. In *Proceedings of the 17th international workshop on Network and operating systems support for digital audio and video*, pages 34–42. Association for Computing Machinery (ACM), 2007.

[49] Steven Webb and Sieteng Soh. A survey on network game cheats and p2p solutions. *Australian Journal of Intelligent Information Processing Systems*, 9(4):34–43, 2008.

[50] Eric Cronin Burton Filstrup Sugih Jamin. Cheat-proofing dead reckoned multiplayer games. *Ann Arbor*, 1001:48109–2122, 2003.

[51] Cristina Cifuentes and K John Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995.

[52] Jeff Yan and Brian Randell. *A systematic classification of cheating in online games.* 2005.

[53] Jeff Yan and Brian Randell. An investigation of cheating in online games. *IEEE Security & Privacy*, 7(3), 2009.

[54] Chris Chambers, Wu-chang Feng, Wu-chi Feng, and Debanjan Saha. *Mitigating information exposure to cheaters in real-time strategy games.* 2005.

[55] Su-Yang Yu, Nils Hammerla, Jeff Yan, and Peter Andras. *A statistical aimbot detection method for online FPS games.* 2012.

[56] Matthew George Tyler. Online gaming cheating prevention system and method, January 29 2013. US Patent 8,360,890.

[57] Brett Tjaden. *Fundamentals of Secure Computer Systems.* Franklin, Beedle & Associates, 2004.

[58] Fabien AP Petitcolas, Ross J Anderson, and Markus G Kuhn. Information hiding-a survey. *Proceedings of the IEEE*, 87(7):1062–1078, 1999.

[59] Patric Kabus, Wesley W Terpstra, Mariano Cilia, and Alejandro P Buchmann. *Addressing cheating in distributed MMOGs.* 2005.

[60] M Nazhif Rizani and Hiroyuki Iida. *Analysis of Counter-Strike: Global Offensive.* 09 2018.

[61] M Claypool, D LaPoint, and J Winslow. Network analysis of counter-strike and starcraft. pages 261 – 268, 05 2003. ISBN 0-7803-7893-8. doi: 10.1109/PCCC.2003.1203707.

[62] Robert Orfali, Jeri Edwards, and Daniel Harkey. Essential client/server survival guide. 01 1996.

[63] Olivier Delalleau, Emile Contal, Eric Thibodeau-Laufer, Raul Chandias Ferrari, Yoshua Bengio, and Frank Zhang. Beyond skill rating: Advanced matchmaking in ghost recon online. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(3): 167–177, 2012.

[64] Castano Henao Juan David Hincapie-Ramos, Andres Felipe. P2p game network framework - communications and context framework for building p2p multiplayer games (intro). June 2007.

[65] Nathan Sheldon, Eric Girard, Seth Borg, Mark Claypool, and Emmanuel Agu. The effect of latency on user performance in warcraft iii. 04 2003. doi: 10.1145/963900.963901.

Appendix A.

Manual

A.1. System Requirements

What is needed to run the Unity engine and games can be found on their website[1]. This is a summary of the requirements. Keep in mind that we only tested the project on Windows 8 and 10.

A.1.1. For development

- OS: Windows 7 SP1+, 8, 10, 64-bit versions only; macOS 10.11+
- CPU: SSE2 instruction set support.
- GPU: Graphics card with DX10 (shader model 4.0) capabilities.

A.1.2. Additional platform development requirements

- iOS: Mac computer running minimum macOS 10.12.6 and Xcode 9.0 or higher.
- Universal Windows Platform: Windows 10 (64-bit), Visual Studio 2015 with C++ Tools component or later and Windows 10 SDK

A.1.3. For running Unity games

- OS: Windows 7 SP1+, macOS 10.11+, Ubuntu 12.04+, SteamOS+
- Graphics card with DX10 (shader model 4.0) capabilities.
- CPU: SSE2 instruction set support.

A.2. Installation

The project is public on GitHub[2] and is under the GNU General Public License v2.0[3].

[1] Unity - System Requirements: `https://unity3d.com/unity/system-requirements`, accessed: 07.01.2019

[2] Rocket-Lanes GitHub repository: `https://github.com/alexisdavidson/Rocket-Lanes`, accessed: 07.01.2019

[3] GNU General Public License: `https://github.com/alexisdavidson/Rocket-Lanes/blob/master/LICENSE`, accessed: 07.01.2019

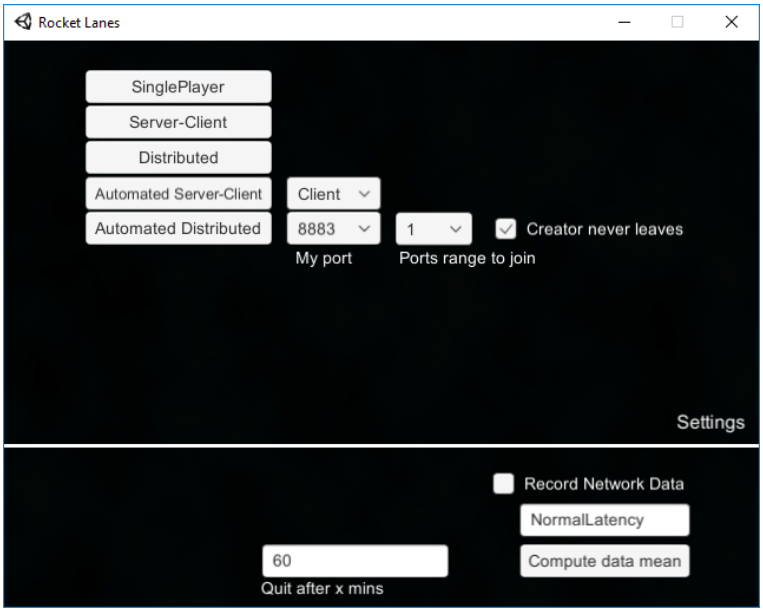

Figure A.1.: Main menus

First, download and install the Unity Engine version 2018.3.0b12 on the Unity website[4]. Then follow these steps:

```
- create blank project with Unity
- close Unity
- open project directory
- delete ProjectSettings, Packages and Temp folders
- open git bash in project folder
- $ git init
- $ git remote add origin
    https://github.com/alexisdavidson/Rocket-Lanes.git
- $ git pull origin master
```

To create an executable, open the Unity project and under the *File* menu, select *Build Settings...* and finally press *Build*.

A.3. How to use the software

A.3.1. Main menus

The main menus are separated in two parts (see Fig. A.1). On the top part of the screen you can choose which mode you want to play the game on, on the bottom part of the screen you can setup some settings.

[4]Download - Unity: `https://unity3d.com/get-unity/download/archive`, accessed: 07.01.2019

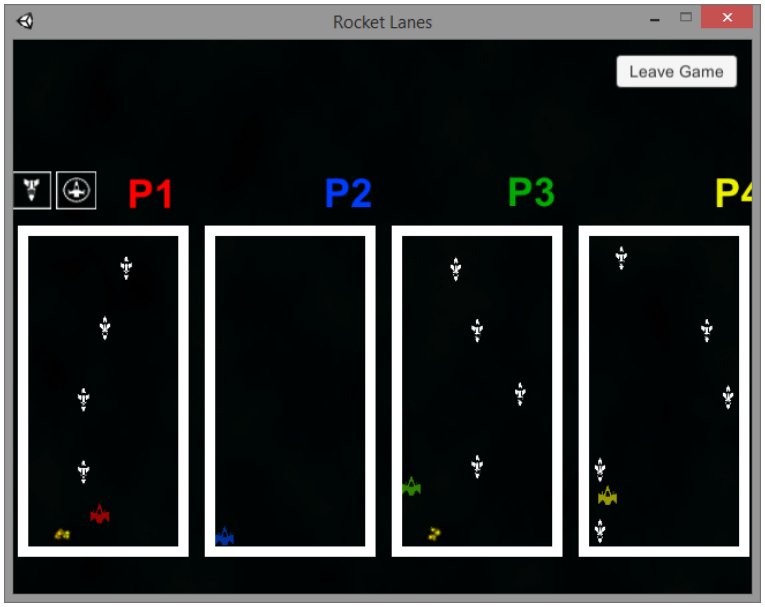

Figure A.2.: Single player

The modes

The first three modes are: the single player mode, the client-server mode and the distributed mode. The two last ones only exist for the sake of the experiment. They simulate a player that joins and leaves games regularly, on the client-server or distributed network architecture.

Settings

With the *Quit after x mins* field you can set the software to leave the game after a specific amount of time.

You can choose if you want to record network data when playing or not and set the name of the subfolder to save these data in. Finally, you can trigger the mean computation of all gathered data.

A.3.2. Single player

In the single player mode, you play against three AIs (see Fig. A.2). They move randomly and send rockets and cast shield at regular intervals. Use the *WASD* keys to move and click on the buttons on the top of your lane (the most left one) to send rockets or cast a shield.

A.3.3. Client-server

To create a server, choose your port and click on *Host*. To join a server, enter the targeted IP and click on *Join*.

When in-game, at the top of the screen, there is an option to use an AI to control your character. It can be toggled anytime.

A.3.4. Distributed

Enter your listening port on the corresponding field. When joining a game, enter the targeted port as well and click on *Join*. Otherwise, click on *New Game*.

A.3.5. Automated client-server

Before launching this mode, you need to specify if the current instance will be a host or a client. To do that, use the dropdown on the right of the *Automated Server-Client* button in the main menus.

A.3.6. Automated distributed

Before launching this mode, you need to specify which listening port the current instance will have. To do that, use the dropdown on the right of the *Automated Distributed* button in the main menus.

You also need to specify which range of ports it will try to join. If set to 1, it will only join the port 8881. If set to 7, it will join the ports between 8881 and 8887, randomly. It is important to set this number to be the same as the number of instances running the experiment.

Finally, the option *Creator never leaves* ensures that the creator of a game never leaves it. This is to simulate an experiment similar to the client-server one, where the host never leaves its game. To achieve this, on each instance, set this option to true and the range of ports to 1.

Here is an example of instructions to run an experiment with three instances:

- Launch three different instances of the game, place them meaningfully on your screen so you can see each clearly
- Set the *My Port* field to respectively 8881, 8882, 8883
- For each instance, set *Ports range to join* to 3
- Set *Creator never leaves* to false
- Click on *Automated Distributed* on each instance

With this setting, three instances will be creating, joining games and playing with each other.

Appendix B.

Code documentation

The codebase is divided into these major components:

- Game logic
- Singleplayer Controller
- Client-server Controller
- Distributed Controller
- Automated Runs Controller
- Data Recorder

We do not document the implementation of the user interface, the rendering and special effects since they are trivial and have no meaning in the scope of this master thesis.

B.1. Game

Components that are important to the gameplay are found in the *Assets/Scripts/Game* folder and include:

- The player
- Lanes and walls
- Game Controller
- Rockets
- Shields
- Artificial intelligence

B.1.1. Player and player controller

The following code shows the modelling of the movements of the player.

```
//PlayerController.cs
void Update () //called every frame
{
    Move(new Vector2(Input.GetAxis("Horizontal"), Input.GetAxis("Vertical")));
}

void Move(Vector2 direction)
{
    rigidBody.velocity = new Vector3(direction.x * speed, direction.y * speed, 0);
}
```

`Input.GetAxis("Horizontal")` is equal 1.0 when the user presses the right arrow, -1.0 when it presses the left arrow and otherwise 0.0.

`Input.GetAxis("Vertical")` is equal 1.0 when the user presses the up arrow, -1.0 when it presses the down arrow and otherwise 0.0.

The `rigidBody` attribute refers to a `RigidBody` Unity component that models a physical body with a mass, a size and other characteristics.

B.1.2. Lanes and wall

A lane consists of walls, a player in it and rocket spawners at the top.

Walls are simple graphical elements with `BoxCollider2D` components added to them, blocking the player movements.

A rocket spawner is an invisible object placed at the top of a lane. Its only important information is its position at which rockets are spawned.

When a player enters the game, spawns or dies, the according lane registers it. The following code is in the player logic and notifies the lane when entering or leaving.

```
void OnTriggerEnter2D(Collider2D other)
{
    if (other.GetComponent<Lane>())
        other.GetComponent<Lane>().Enter(this);
}

void OnTriggerExit2D(Collider2D other)
{
    if (other.GetComponent<Lane>())
        other.GetComponent<Lane>().Leave(this);
}
```

B.1.3. Game Controller

The game controller takes inputs from the interface and calls the right controllers to take action. It also keeps information about the current game like the number of players and keeps references to players and lanes.

An important job of the game controller is to keep a reference of the network interface currently used. This way, the game controller forwards action requests to the corresponding network architecture used.

B.1.4. Rockets

The code of a rocket is very simple. It has a constant initial velocity, and when colliding with something, it explodes. If it collides with a player with no shield, that player loses 1 health.

```
void OnTriggerEnter2D(Collider2D other)
{
    Player player = other.GetComponent<Player>();
    if (player != null)
    {
        if(!player.ShieldEnabled())
            player.LoseHealth(1);
    }
    Destroy(this.gameObject);
}
```

B.1.5. Shields

When a shield is cast, it disappears after 15 seconds. The most important characteristics are the parameters like the duration of the shield and the cooldown, stored in the `Shield` class.

```
public class Shield : NetworkBehaviour
{
    public bool shieldEnabled = false;

    public float durationShield = 2;
    float timeStartShield = 0;

    public GameObject shieldGO = null;

    public static float Cooldown = 15.0f;
}
```

B.1.6. Artificial intelligence

An artificial intelligence can be used to control a player. At random intervals of time, it picks a random direction to move along. Also at random intervals of time, it casts a shield on itself and sends a rocket to the next occupied lane.

All of this happens by using some timestamp manipulations and making calls to the game controller.

B.2. Singleplayer

The `SinglePlayerController` class can be found in the *Assets/Scripts/SinglePlayer* folder.

In the initialization, we spawn the main player and enable its `PlayerController` script. We give its reference to the `GameController` singleton. Then we spawn the other players and add an AI script to them.

```
public bool Initialize()
{
    //spawn player
    Player player1 = Instantiate(playerPrefab, spawns[0].transform.position,
        Quaternion.identity);
    player1.gameObject.GetComponent<PlayerController>().enabled = true;
    gameController.player = player1;


    //spawn AIs
    for(int i = 0; i < 3; i++)
    {
        Player playerAI = Instantiate(playerPrefab, spawns[1 + i].transform.position,
            Quaternion.identity);
        playerAI.gameObject.AddComponent<AI>();
    }
    return true;
}
```

Asking consent is never denied, this is why we directly apply the consent request

```
public void AskForConsent(ConsentMessage consentMessage)
{
    ApplyConsent(consentMessage);
}


public void ApplyConsent(ConsentMessage consentMessage)
{
    if(consentMessage.consentAction == ConsentAction.SpawnRocket)
    {
        gameController.lanes[consentMessage.parameters[1]].
            spawnManager.Spawn(consentMessage.result);
    }
    else if(consentMessage.consentAction == ConsentAction.CastShield)
    {
        lane.player.CastShield();
    }
}
```

Finally, we always handle collision since we have the authority in singleplayer.

```
public bool HandleCollisions(Lane lane)
{
    return true;
}
```

B.3. Client-Server

The code for the client-server implementation can be found in the *Assets/Scripts/Server-client* folder.

B.3.1. Network Manager

Initialization

```
bool void Initialize()
{
    NetworkServer.RegisterHandler(NetworkMessages.AskForConsentMsg, OnAskForConsentMsg);
    return true;
}
```

The `RegisterHandler` function called by the `NetworkServer` singleton initializes a register for the server to handle incoming messages that in our case represent clients asking for consent. Our initialization code is short thanks to the `NetworkManager` class which already has code handling network initialization that is executed when loading the scene.

Connection and spawning

```
//Called on the server when a client adds a new player with ClientScene.AddPlayer.
public override void OnServerAddPlayer(NetworkConnection conn, short playerControllerId)
{
    Lane lane = gameController.GetFirstUnoccupiedLane();
    GameObject playerGameObject = (GameObject)Instantiate(playerPrefab,
        lane.startPosition.transform.position, Quaternion.identity);

    NetworkServer.AddPlayerForConnection(conn, playerGameObject, playerControllerId);
}
```

The `OnServerAddPlayer` method is called when a client gets added to the scene, so in other words when a client joins the game. We override it in order to perform some gameplay specific actions. We find the next unoccupied lane which will be where the player spawns and we instantiate a player prefab in the position defined by this lane. Finally, we call the `AddPlayerForConnection` method to associate the player instance with the connection.

Asking for consent

```
public void AskForConsent(ConsentMessage consentMessage)
{
    networkClient = NetworkClient.allClients[0];
    networkClient.Send(NetworkMessages.AskForConsentMsg, consentMessage);
}
```

Asking for consent happens by sending a consent message to the server.

Receiving a consent request

```
void OnAskForConsentMsg(NetworkMessage netMsg)
{
    ConsentMessage msg = netMsg.ReadMessage<ConsentMessage>();
```

```
    if(consentMessage.consentAction == ConsentAction.SpawnRocket)
    {
        bool cheating = !gameController.lanes[consentMessage.parameters[1]].
            spawnManager.ValidIndex(consentMessage.result);
        if(!cheating) ApplyConsent(msg);
        else Debug.Log("Cheat! wrong rocket infos");
    }
    else if(consentMessage.consentAction == ConsentAction.CastShield)
    {
        Lane lane = gameController.lanes[consentMessage.parameters[0]];
        if(lane.player.ShieldReady()) ApplyConsent(msg);
        else Debug.Log("Cheat! Shield not ready");
    }

    //send apply consent confirmation
    NetworkServer.SendToClient(netMsg.conn.connectionId,
        NetworkMessages.ApplyConsentMsg, msg);
}
```

When the server receives a message, it analyzes which consent action was requested and its parameters. Then, depending on the result of its decision, the consent is applied or not.

Applying consent

```
public void ApplyConsentt(ConsentMessage consentMessage)
{
    if(consentAction == ConsentAction.SpawnRocket)
    {
        NetworkServer.Spawn(gameController.lanes[consentMessage.parameters[1]].
            spawnManager.Spawn(consentMessage.result));
    }
    else if(consentMessage.consentAction == ConsentAction.CastShield)
    {
        Lane lane = gameController.lanes[consentMessage.parameters[0]];
        lane.player.CastShield();
    }
}
```

In this example the client requested to spawn a rocket. To apply this request, we verify if the current application is the server and we spawn it to the corresponding lane. This occurs by calling the method `Spawn` which handles the spawning of network objects gracefully.

Collisions

```
public bool HandleCollisions(Lane lane)
{
    //only if server
    return NetworkServer.active;
}
```

Only the server handles gameplay-sensitive collision detection and reactions. The `NetworkServer.active` attribute is `true` if the current application is the server.

```
//Player.cs
public override void OnStartAuthority()
{
    GameController gc = GameObject.FindObjectOfType<GameController>();
    gc.player = this;

    PlayerController pc = GetComponent<PlayerController>();
    pc.enabled = true;
}
```

By overriding the `OnstartAuthority` method in a class inheriting `NetworkIdentity`, we can define behaviour to be called at the start of a scene only by applications having the authority on that instance. This means that this method will only be called by a client which the player belongs. This way, we can add the `PlayerController` script to the `Player` instance, allowing the current user to control it.

B.4. Distributed

The code for the implementation of the distributed architecture can be found in the *Assets/Scripts/P2P* folder, including:

- P2P Controller
- Connection and Connection Manager
- Listener
- Sender
- Consent Manager

B.4.1. P2P Controller

The P2P Controller class is charged of the initialization of the network transport system and implements our network interface.

Initialization

A simplified version of the `Initialize()` method looks like this:

```
public bool Initialize()
{
    NetworkTransport.Init();

    ConnectionConfig config = new ConnectionConfig();
    P2PChannels.ReliableChannelId = config.AddChannel(QosType.Reliable);
    P2PChannels.UnreliableChannelId = config.AddChannel(QosType.Unreliable);

    HostTopology topology = new HostTopology(config, 10);

    P2PConnectionManager.myHostId = NetworkTransport.AddHost(topology, myPort);
```

```
    if(P2PConnectionManager.myHostId == -1) //port busy or something
    {
        DisplayError("Port " + myPort + " is busy");
        return false;
    }

    initialized = true;
    return true;
}
```

We define different channels: a reliable one for important messages and an unreliable one. We set some topology parameters and double check errors.

Asking for consent

When asking for consent, if the player is alone in the game, it immediately applies it. Otherwise, it sends the request to all other peers.

```
public void AskForConsent(ConsentMessage message)
{
    message.consentId = P2PConsentManager.GetNextConsentIdAndIncrement();
    if(P2PConnectionManager.SuccessfulConnectionsCount() > 0)
    {
        P2PConsentManager.AddPendingConsent(message);
        P2PSender.SendToAll(P2PChannels.ReliableChannelId, message,
            MessageTypes.AskConsent);
    }
    else
    {
        P2PConsentManager.ApplyAndSpreadConsentResult(
            OnAskForConsentMsg(-1, -1, message));
    }
}
```

Receiving a consent request

When receiving a consent request, we verify if the request is legitimate. We send back our result. This code shows the example of a request to cast a shield:

```
if(message.consentAction == ConsentAction.CastShield)
{
    answerMessage.result = 0;

    Lane lane = gameController.lanes[message.parameters[0]];
    if(lane != null && lane.player != null && lane.player.ShieldReady())
        answerMessage.result = 1;

    P2PSender.Send(hostId, connectionId, P2PChannels.ReliableChannelId,
        answerMessage, MessageTypes.AnswerConsent);
}
```

If the shield of the player is indeed ready, our result is 1, meaning that we agree to the request. Otherwise, the result is 0, meaning that we do not agree.

Applying a consent request

Applying a consent request in the distributed architecture happens very similarly to the other implementations.

B.4.2. Connection and Connection Manager

The connection manager stores and updates the connection with peers. A connection contains the following information: host id, connection id, ip, port, lane and whether the connection was successful or not.

B.4.3. Listener

The listener listens to incoming messages. If the message contains a connect or disconnect event, a connection will be created or removed. Otherwise, the messages are forwarded to the P2P Controller.

```
public static void Listen()
{
    byte[] recBuffer = new byte[P2PController.bufferLength];
    int bufferSize = P2PController.bufferLength;
    int dataSize;
    byte error;
    NetworkEventType recData = NetworkTransport.Receive(out recHostId,
        out connectionId, out channelId, recBuffer, bufferSize, out dataSize,
        out error);

    while(recData != NetworkEventType.Nothing)
    {
        switch (recData)
        {
            case NetworkEventType.ConnectEvent:
                P2PConnectionManager.ConnectEvent(recHostId, connectionId);
                break;
            case NetworkEventType.DataEvent:
                CreateNetworkReader(recBuffer);
                break;
            case NetworkEventType.DisconnectEvent:
                P2PConnectionManager.RemoveConnection(recHostId, connectionId);
                break;
            default:
                break;
        }

        recData = NetworkTransport.Receive(out recHostId, out connectionId,
            out channelId, recBuffer, bufferSize, out dataSize, out error);
    }
```

```
}
```

In this code, the `NetworkTransport.Receive` method is called to fetch all incoming packets. In a while loop, we iterate through them and process them.

B.4.4. Sender

To send a message, we serialize its content using the `NetworkWriter` class. We compute the length of the data and send the packet with the `NetworkTransport.Send`, after which we double check for errors.

```
public static void Send(int hostId, int connectionId, int channelId,
    MessageBase message, short messageType)
{
    NetworkWriter writer = new NetworkWriter();
    writer.StartMessage(messageType);
    message.Serialize(writer);
    writer.FinishMessage();
    byte[] writerData = writer.ToArray();
    int bufferLength = writerData.Length;

    NetworkTransport.Send(hostId, connectionId, channelId, writerData,
        bufferLength, out P2PController.error);
    P2PController.CheckError("Send");
}
```

B.4.5. Consent Manager

The consent manager keeps tracks of consent requests sent that are waiting for answers, called *pending consents*. If no answers come for a too long time (timeout), the action is simply applied.

When enough votes for a request are received, the consent manager computes the final result to be applied and sends it again to other peers, as shown in the following code.

```
public static void ApplyPendingConsent(P2PPendingConsent pendingConsent)
{
    //Enough votes received. Pick the most occurring result
    int mostOccurringAnswerResult = pendingConsent.answerConsentMessages
                                                  .GroupBy(acm => acm.result)
                                                  .OrderByDescending(g => g.Count())
                                                  .Select(g => g.Key)
                                                  .FirstOrDefault();

    //Apply it
    AnswerConsentMessage mostOccurringAnswer = pendingConsent.answerConsentMessages[0];
    mostOccurringAnswer.result = mostOccurringAnswerResult;
    ApplyAndSpreadConsentResult(mostOccurringAnswer);

    pendingConsents.Remove(pendingConsent);
}
```

B.5. Automated Runs

The code for automated runs can be found in the *Assets/Scripts/Automated Runs* folder. The `AutomatedRunController` controls which state the run is at and transitions to another state when the time has come.

There are four different states in a run:

- Idle
- InError
- Joining
- Playing

Each state inherits the following `RunState` class.

```
public class RunState
{
    protected float timeStartState = 0;
    protected int timeUntilTransition = 1;

    public RunState()
    {
        timeStartState = Time.time;
    }

    virtual public bool ReadyToTransite()
    {
        return Time.time - timeStartState > timeUntilTransition;
    }

    virtual public RunState Transite()
    {
        return null;
    }
}
```

Upon each state transition, an action happens. The possible actions include joining a server, loading a new scene, displaying an error, etc.

This code shows the `Transite` method in the `StateJoining` class:

```
public override RunState Transite()
{
    if(GameController.gameStarted)
    {
        return new RunStatePlaying();
    }
    else if(GameObject.FindGameObjectWithTag("ErrorPanel") != null)
    {
        return new RunStateInError();
    }
```

```
    return new RunStateJoining(targetPort);
}
```

If the game has started, transite to the `Playing` state. If there is an error, transite to the `InError` state. Otherwise, keep trying to join.

B.6. Data Recording

The code for data recording can be found in the *Assets/Scripts/Data Recording* folder. The main components are the `Recorder` and `Session` classes.

The `Session` class contains many parameters that are meaningful for the experiment of the master thesis. There also are some methods for mean computation, updating average players count, etc.

When a game starts, if the recorder was enabled by the user, a new session is created. When the game ends, the session is closed and saved in a file on the disk in the *Network Data* folder. The name of the session file is composed of the name of the current network architecture used and the current date (*yyyy-MM-dd HH.mm.ss.fff*).

Finally, the `MeanComputer` class takes all session files of the same network architecture, computes the mean for each characteristic and stores them on a new file.