Maersk McKinney Moller Institute
SDU Software Engineering

# CoffeeBreak
# A spacially aware virtual communication breakroom

Bachelor Project Report

```
        socket.join(rid)
        socket.emit('onLoggedIn', {room: room, self: newClient, all: member
        roomio.to(rid).emit('onUserConnected', newClient)

        console.log(clientId + " succesfully logged in to room " + rid)
    } catch(error) {
        console.log(error)
    }
})

socket.on('onChatMessage', message => {
    roomio.to(rid).emit('onChatMessage', message)
})

socket.on('onMovePlayer', async movement => {
    let clientId = await clientMap.getClientId(socket.id)
    await userRepo.setPosition(clientId, movement)
    let nearby = await userRepo.getNearby(clientId, nearbyThreshold)
    roomio.to(rid).emit('onMovePlayer', {id: clientId, x: movement.x, y: mo
    socket.emit('nearby', {nearby: nearby, threshold: nearbyThreshold})
})
```

**CoffeeBreak**
A spacially aware virtual communication breakroom

Bachelor Project Report
June, 2021

By
Frederik B. Roth
Marcus L. Jensen

Published by: SDU, SDU Software Engineering, Campusvej 55, 5230 Odense M Denmark
www.sdu.dk/mmmi

## Approval

| | |
|---|---|
| Frederik B. Roth - 453515 | Marcus L. Jensen - 466991 |
| ............................................ | ............................................ |
| *Signature* | *Signature* |
| ............................................ | ............................................ |
| *Date* | *Date* |

## Abstract

Online communications become more and more prevalent these days, even more so during the Covid-19 pandemic. There are multiple common ways of facilitating online communications, some with direct connections between peers, some where the connections go through a relay between the peers.

This project aims to implement an online communication service that can help users have immersive online communication with each other in a virtual space. The goal is to implement a robust and scalable solution that can deliver on the above aspects and provide at good platform for communication. This will be implemented based on two different designs, in order to measure the performance difference between the two common solutions. In order to accomplish this, each solution is deployed on the same hardware, and hardware utilization is measured while increasing load. The result ended with a fully deployed communication system that enables spacially aware voice communication. Additionally, the resulting product and the tests performance, points towards that, at least for the client, that a relay between peers is a significantly better solution. This is due to linear increase in hardware utilization, in contrast to an exponential increase of connections directly between peers. However, the relay-based solution has greater complexity, and can be significantly more difficult to implement.

While it is easily tempting for a new, naive startup to think they should develop a similar system to allow as many people as absolutely possible through the best possible solutions, the complexities of the most difficult solutions can easily be prohibitive, and result in the final system actually being of worse quality than if they had chosen a simpler, yet worse performing solution. However disregarding these results, a fully functioning system was created based on the peer to peer design that upholds all established requirements set for the project.

# Acknowledgements

**Frederik B. Roth**
**Marcus L. Jensen**

We would like to thank ourselves for making it through this project despite the ever-encompassing dread of the quarantine along with the number of technical difficulties encountered throughout the project.

We would also like to thank the creator of this thesis template, because they asked to by leaving a line which these particular lines are based directly upon, and because the template has allowed us to focus on the important aspects of writing this report, rather than wasting time setting everything up.

But most of all, we would like to thank our supervisors:

**Leon Bonde Larson**, M.Sc. Robotics, The Maersk Mc-Kinney Moller Institute
We extend our gratitude to Leon for his significant help in getting us through this project with a satisfactory result. His steady, firm-but-fair, and thoroughly detailed feedback has been a persistent inspiration for improvements, and both team members feel that we have learned magnitudes from it. Even if the final exam goes poorly for whatever reason, the team believes that it will all have been easily worth it due to the many skills we have learned and improved.

**Mathias Neerup**, Mærsk Mc-Kinney Møller Instituttet
We extend our gratitude to Mathias for providing excellent technical support helping with the design and implementation of the system. He has gone above-and-beyond in providing the information we needed, when we needed it, and we feel confident that the final product would not have reached the same height without his help.

# 1 Reading Instructions

This report is written in an effort to explain and document the development of a CoffeeBreak solution, documenting every step of the way from inception to conclusion. Each individual chapter focuses on a particular aspect of the project as a whole, and seeks to be mostly understandable without having a full understanding of the project as a whole. Each chapter additional begins with a short introduction about the chapter in order to provide some context on what that chapter is about, and the project subdomain it encompasses. Some chapters include a conclusion of the chapter at the very end if applicable, which serves as a synopsis of the chapter as a whole, with the goal being that the reader is able to understand the most essential aspects of the chapter though the introduction and conclusion alone, however the reader is unlikely to achieve full understanding of the project by merely skimming these parts. The report is written as the final part of the project.

The report is written with the primary target audience being on an academical level similar to three years of formal education within the field of software engineering at least. The goal is for anyone with such qualifications to be able to follow and understand the majority of the report without much difficulty. The secondary target audience is anyone with familiarity with software development in general, who should be able to understand most of the report, aside from perhaps the most technical aspects. Finally, the tertiary target audience is the general public who should be able to understand the essentials of the report.

The primary goal of this project has been to develop a robust and fully featured CoffeeBreak solution, which will the focus of the majority of this report. Additionally, the secondary goal of comparing different solutions in terms of complexity and performance is the focus of some particular chapters, primarily the Abstract and Result chapters, as well as some parts of the Methodology and Process chapter.

Naturally, ambitious projects rarely live up to their original goals, in particular as the development of this project took place during the *hopefully as of writing* latter half of Covid-19 pandemic, and by extension difficulties with what is colloquially known as *Quarantine Depression*. Individual and team reflections discussing this and more, as well as individual and team learning from the project will be discussed towards the end of this report in the Evaluation and Reflections chapter.

Technologies, methodologies, facts, and products referenced in this report will be supported by the IEEE standard citation, wherein any of the aforementioned elements will be followed by a [x] notation, which links to a corresponding entry in the Bibliography chapter.

Occasionally, the reader may come across footnotes denoted by an $^x$ superscript notation, and a corresponding footnote at the buttom of the page.[1] These footnotes usually contain information which may provide further context or some other information that might be of interest to the reader.

[1]Like this.

# 2 Learning Objectives

The primary learning objectives for each team member has been varied, but both shared the goal of gaining a better understanding of the complete process of product development, in particular using service-oriented architectures and containerization technologies. Additionally, the team has worked to improve their report writing abilities as both has struggled somewhat with this in previous projects, and a project where each person has much greater overview of the project as a whole, as well as a greater feeling of product ownership, means that the project is a great opportunity for improving writing and communication skills.

Additionally, the team has been interested in the benefits of a highly agile development process compared to previous projects, that, while drawing the majority of inspiration from the Agile Iterative Process, does away with most unnecessary fluff and only use the parts that might actually be of use to a two-person team. Furthermore, the team wishes to try out the concept of Domain Driven Design.

Individually, Frederiks personal goals has been to deep dive into the Kubernetes and figure out how to set up a fully functioning deployment, from the individual pod and deployment configurations, to overall load balancing, deployment and maintenance on PaaS providers such as Google or Amazon. Additionally, he was intrigued by the possibility of adjusting and modifying an existing Kubernetes programmatically. Lastly, he wanted to further improve his knowledge of WebRTC, by investigating its different implementation designs and their benefits and flaws.

On the other hand, Marcus' personal goals has been to greatly expand upon his previously rather lackluster web development skill set, in particular by expanding HTML and CSS knowledge, and by working with Node.js and Socket.io to implement the back-end components for the system. Furthermore, he has yearned to improve his understanding of the design of service-oriented architectures, in particular microservice-based architectures. Finally, he wished to better his understanding of RESTful API design, and API design in general. All in an attempt to grow to a more well-rounded full-stack developer.

# Contents

# 3 Introduction

This chapter seeks to introduce the project, as well as the goals and ambitions for the project. Furthermore the product domain is outlined through an outline of similar and competing products and how they work. A general introduction to the technologies used by this project is included as well, as well as what these technologies are used for in the project. Finally, the teams learning objectives are outlined towards the end of the chapter.

## 3.1 CoffeeBreak - A spacially aware virtual communication breakroom

### 3.1.1 Goals

The primary goal of this project is to develop a robust and competently build product using the technologies and methodologies known and available to the developers. The product, CoffeeBreak, is in itself built to solve the issue of online meetings often being *flat*, in the sense that everyone in the meeting is broadcasting their voice and video to everyone else in the meeting. While this is great for actual meetings, it often quickly becomes troublesome when the meetings need break out into discussion groups. CoffeeBreak seeks to solve this by providing a virtual space wherein those in the meeting can freely move around inside the space, with proximity based voice and video chat which keeps individual user groups separate, and thus allow for multiple separate discussions to occur simultaneously in the same room.

The secondary goal is to try and design and implement different solutions to the same problem, then measure the performance of different solutions from a black-box perspective using the same tools and methods, as well as evaluate them on technical complexity, in order to find out the pros and cons of the different solutions.

### 3.1.2 Motivation

The motivation for the primary goal, development of the CoffeeBreak system, is purely educational, it is merely created mostly for learning about the various technologies and practices involved in creating such a system. The motivation for the secondary goal is to attempt to find out the best ways to implement such a system, in terms of both complexity and potential costs.

## 3.2 Related and Competing Works

### 3.2.1 Discord

Discord is a widely popular online communication service which supports all audio, video, and text communication. Communication often occurs in guilds[1], also known as *servers*, which contains a list of channels that the users can use for communications. Channels are either text channels, or voice/video channels and the users can, assuming they have permission to do so by the guild rules, freely chose which channels to communicate in. Communication can occur outside guilds in direct message channels, however this is not of importance to the project.[2]

Discord uses a HTTPS/REST API for general communications as well as WebSockets for sending real-time events and messages between the front-end and the back-end.[3]. It is described as being a distributed system which relies heavily on Elixir for back-end components, including a GenServer process for each guild, the effective equivalent of a room in the context of CoffeeBreak, as well as consistent hashing for distributing requests.[4]

Discord makes use of WebRTC[5] to facilitate voice/video communications across users in a voice/video channel, using a Media Relay Server which then relays user media streams to each other, without exposing their public IP addresses to each other or relying on ICE[6] protocols to facilitate connections, reducing potential issues with certain types of NAT. While this is secure and more reliable, the solution also appears to be somewhat complicated.[2]

Furthermore, the Discord desktop client makes use custom-build media engine built on top of the existing WebRTC native library, which allows them greater control of the media streams, however at the cost of requiring constant maintenance to keep updated with WebRTCs development.[2]

Discords back-end, in general, appears highly advanced, using a number of advanced techniques that are both beyond skill level of the CoffeeBreak team, and would probably take significantly more time to implement than is available. The material does, however, indicate that it is perfectly feasible to create a distributed WebSocket based system, such as what would be needed for this project. Additionally, the use of WebRTC could be an excellent candidate for facilitating video and voice communication within a room.

#### 3.2.2 Zoom

Zoom is another incredibly popular online communication service, one focused primarily around hosting meetings using voice/video communications, with text communication being available. This is commonly used for both for education and for organization communications as an alternative for physical meetings. With Zoom, a user with a subscription can host a zoom room which people can join using a special URL.[7]

As with Discord, Zoom voice and video chat works through WebRTC or technology build on top of WebRTC such as Vonage Video API[8], as well as a number of other APIs used to power the chat and room hosting.[7] There seems to be little in the material on Zoom which is able to help this project, except further supporting the use of WebRTC for video and voice connections. The back-end of Zoom can with some level of confidence be assumed to be scalable, due to its rapid, seemingly smooth growth during quarantine, though the inner workings of Zoom remains a mystery.

#### 3.2.3 Mumble

Mumble is an open-source self-hosted client-server based online communication service which allows for voice and text chat inside channels on servers hosted manually much like Discord albeit it arguably more primitive in terms of user experience, as Mumble is an older piece of software. Owing to Mumbles open-source self-hosted nature, communication is entirely private between everyone who is connected to a particular server.[9]

Mumble uses it's own custom low level protocol for all kinds of communication, including server events, text, and voice communication[1]. This allows them full control of what is transmitted between the users and the server, at cost of significant increase in complexity. The custom protocol allows for easy integration of positional XYZ data alongside the voice data, which can then be used by plugins to alter the gain of users based on their proximity to each other.[11]

In order to allow proximity based voice chat, CoffeeBreak needs to, in one way or another, be aware of the virtual world position of all room members. The Mumble documentation points towards them using their very own protocol which supports that, but it does also

[1]The Mumble website front page claims that Mumble uses VoIP communication, but the protocol documentation does not appear to expand on this.[10]

indicate that there is no more data necessary to be transmitted other than member coordinates, something which could easily be done using WebSockets or even simply HTTP.

## 3.3 Conclusion

There are two major goals with the development of the CoffeeBreak system, a primary which is to design and implement the system itself, for the great learning opportunities such a project provides. The secondary goal is to attempt to design and implement varieties to the solution, in order to measure and compare different solutions in terms of complexity and performance. A number of existing products were evaluated, which pointed towards the use of sockets and the ”WebRTC” technology as major parts of similar systems.

# 4 Methodology and Process

The goal of this chapter is to outline which formal methods were selected for the development of this project, including the development process model, tools used, planning and estimations, as well as the risks involved with the project and how the team dealt with said risks. Furthermore, the methodology for achieving the goal of measuring and comparing different solutions is outlined.

## 4.1 Agile Iterative Process

The Agile Iterative Process[12] (AIP) was chosen early on in development due to it's simplicity and natural fit with the developers usual preferred workflow. Some aspects of AIP were omitted to reduce unnecessary work overhead, such as a full complex software change process, and a dedicated process manager.

### 4.1.1 Iterations

The length of each iteration of the work process varied over the course of development due to changes in available work time, however it was agreed to be set to two weeks once full time was allotted to development. A time span of two weeks was chosen due to a natural fit with bi-weekly supervisor meetings and updates, with the goal being to have a run-able demonstration available for each meeting. Prior to full time work, no set iteration time span was decided, due to the unpredictability of available time.

### 4.1.2 Product Manager and Users

As with many projects of this nature, that being a student project, the product manager and users have in practice been the developers themselves. This does come with some difficulties, in particular in the role as users, as the developers might have some bias when considering the system from an outside black-box perspective, which could impact how the systems behaviour is viewed.

### 4.1.3 Backlog

The project backlog was initially based on a GitHub project[13] which was used to track each tasks in the backlog. Each developer wrote in a set of tasks which were to be completed, and was then assigned a number of them. However, due to the simplicity of the system and short time span of development, the developers found it easier to merely maintain mental notes of the backlog instead, though the actual backlog tasks in the GitHub project were closed as they were completed in order to provide some overview on how close the project was to completion.

## 4.2 Project Estimations

### 4.2.1 Gantt Diagram

A Gantt Diagram was created early in development in order to maintain a time table, such that it would be possible to track whether or not the project was behind, on, or ahead of schedule. The milestones of the first half of the diagram was based on supervisor lectures in related areas, while the second half was loosely estimated based on previous experience. Initially completed tasks were filled in, but this practice was abandoned during the later stages of development, as it was deemed unnecessary as the team could easily manage who had worked on what due to the small team size and frequent meetings.

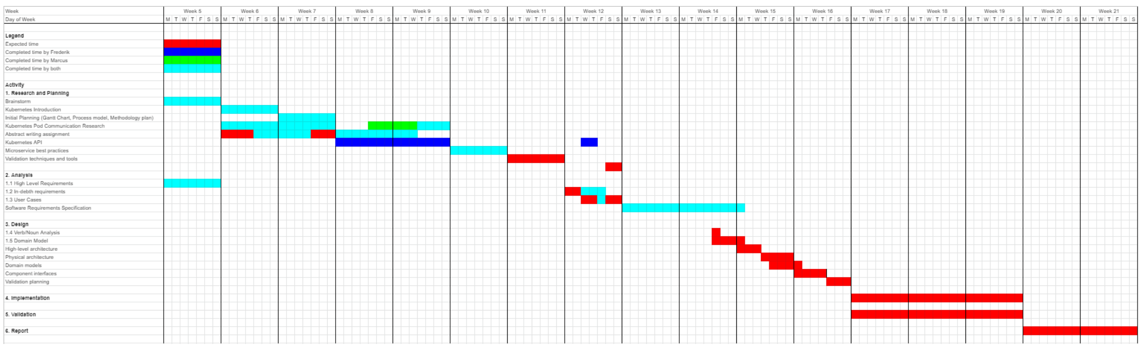


Figure 4.1: Gantt Diagram showing the general estimated time table between week 5 and week 21, with research and planning occuring between week 5 and week 11, analysis between 12 and 14, design in week 15 and 16, implementation and validation from from week 17 to 19, and finally report writing at the end in week 20 and 21. A larger more readable version is available as A.1

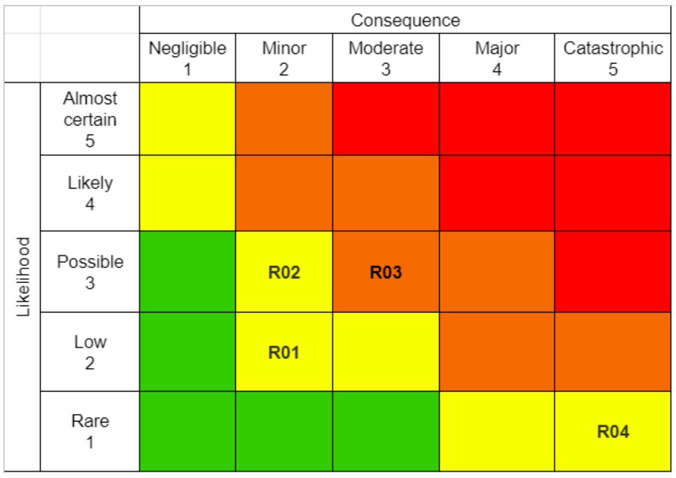
## 4.3 Risks

Naturally any project carries risks. A few in particular were identified as the most threatening towards this project, mostly relating to the inherent unpredictability of software, unfamiliarity with the technologies involved, and the ever-endless Covid-19 quarantine taking a toll on the mental health of the developers.

| ID | Description | Mitigation Strategy |
|---|---|---|
| R01 | The product would not be completed in time due to poor time management. Software development is known to be volatile and difficult to estimate, and so it happens that projects fail to be finished in time. | Reduction through estimating a tentative time table. |
| R02 | The project would fail to reach its goals due to technical difficulties with unfamiliar technologies. Neither developer are very experienced with the technologies involved beyond the basics, and there could easily unforeseen consequences with the choice of technology. | Reduction through prototyping a rudimentary system. |
| R03 | Lack of motivation due to mental health impact of quarantine would impact project quality. While both developers have dealt with the quarantine somewhat well prior to the project, the prolonged quarantine can easily affect motivation to get work done. | Acceptance |
| R04 | The project exam is failed. This would be a critical failure of the project as a whole, though it is unlikely to occur unless everything that could go wrong did go wrong. | Avoidance through work. |

Table 4.1: Table of the four most major estimated project risks. Each risk has an ID, a description and a mitigation strategy.

Each risk is assessed on likelihood to occur and severity of consequence on a scale between 1 and 5 using a Risk Assessment Matrix. The colors represent the severity of the potential risk, taking likelihood and consequence into account. Green is low, yellow moderate, orange high , while red is extreme.



| | | Consequence | | | | |
|---|---|---|---|---|---|---|
| | | Negligible 1 | Minor 2 | Moderate 3 | Major 4 | Catastrophic 5 |
| Likelihood | Almost certain 5 | 🟨 | 🟧 | 🟥 | 🟥 | 🟥 |
| | Likely 4 | 🟨 | 🟧 | 🟧 | 🟥 | 🟥 |
| | Possible 3 | 🟩 | 🟨 R02 | 🟧 R03 | 🟧 | 🟥 |
| | Low 2 | 🟩 | 🟨 R01 | 🟨 | 🟧 | 🟧 |
| | Rare 1 | 🟩 | 🟩 | 🟩 | 🟨 | 🟨 R04 |

Figure 4.2: The Risk Assessment Matrix for the project. It can be seen how no risks are within the red zone, but R01, R02, and R04 lies within the yellow zone. R03 lies within the orange zone, and is estimated to be the greatest risk to the project.

## 4.4 Conclusion

The development methodologies and processes have overall been highly agile, with little in terms of significant specific plans for what should be done when, and how it should be done, aside from Gantt diagram which ended up mostly as a loose time table. A slightly toned down variant of the Agile Iterative Process was used for development, with a varying iteration time span until the later stages, where full time work was a possibility, and the time span was set to two weeks. The goal of each iteration was to end with a presentable demo for the bi-weekly supervisor meetings. A written backlog was initially used, but soon grew out of favor as the small team size had no real need for it. There were some risks with the project, including unpredictable development times, difficulties with technology, and outright failing the final exam. These were, however, only estimated to be of only minor concern, while the ongoing Covid-19 quarantine and its impact on mental health was estimated to be a much more significant threat, in particular as there is not much which can be done to prevent it.

# 5 Requirements

This chapter seeks to outline the fundamental requirements that drive the development of the product, both in terms of functional and non-functional requirements, all prioritized using MoSCoW-style prioritization. Requirements are written in a *”Users must/should/-could/would be able to x”* format.

Below are the initial requirements decided on during the projects infancy. In the context of a commercially developed product for a particular client, they represent what perhaps would be received from the client when they first commission the system, for then to be further analyzed in the next chapter.

- *Multiple users must be able to connect to the same room.*
- *Users should be able to connect to each other with audio and video streams.*
- *Users should be able to connect to each other with audio and video streams.*
- *User should be able to hear other users audio streams.*
- *Users must be able to create different rooms.*
- *User must be able to connect to a room using an ID.*
- *Room owners could be able to delete created rooms.*
- *User should be able to move around the room.*
- *Users should be able to send text messages the room.*
- *The sound from other users could reflect the distance between each other.*
- *The user would be able to login/register a permanent account.*
- *Room owners would be able to mute certain users.*
- *Room owners would be able to send priority text messages.*
- *A browser-based session client must be available to users.*
- *The back end could support any type of user client through a public API.*
- *The back end HTTP servers must be encrypted through HTTPS to protect end-user data for security.*
- *The system must be able to scale horizontally.*
- *The system should be able to be deployed using containerization technologies.*

## 5.1 Conclusion

This chapter outlines a list of initial requirements as if received by a client commissioning the system, including requirements related to creating and joining rooms, as well as allowing users to stream proximity-based audio and video to each other. Furthermore, there are a number of requirements revolving around the qualities of the system, such as a security and scalability.

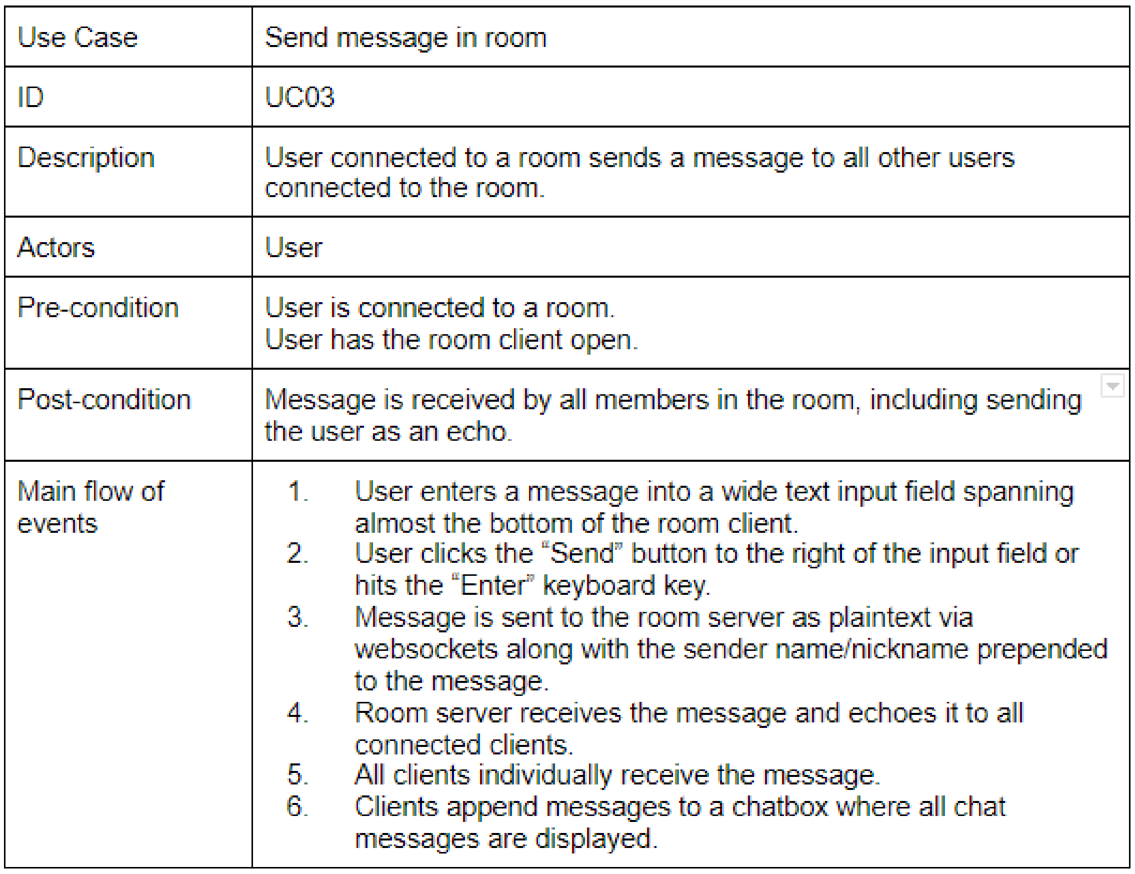
# 6 Analysis

This chapter will go through the the analyzing process of specifying and narrowing down, more specific requirements from the overall requirements established at the start of the project. Additionally, it will also discuss the fundamental changes that were made to the requirements and why it that was the case

## 6.1 Use Cases

An overall requirement list, such as the aforementioned, is a good beginning to establish the core features and requirements the system needed to have in order to have a working product. However it is not enough to more accurately describe the exact requirements needed in order to develop such a system from the ground up. In order to establish these requirements a more precise requirement list is needed to be found.

Firstly the primary requirements, consisting of requirements mandatory to the systems overall functionality, is rooted out. These requirements is then analyzed through the creation of Use Cases. These use cases represents a user going through a part of a system. This is also called the flow of events. An example of such a Use Case is shown here below. All of the Use Cases created for this project is in the Appendix, Figure A.6 to A.12



| | |
|---|---|
| Use Case | Send message in room |
| ID | UC03 |
| Description | User connected to a room sends a message to all other users connected to the room. |
| Actors | User |
| Pre-condition | User is connected to a room.<br>User has the room client open. |
| Post-condition | Message is received by all members in the room, including sending the user as an echo. |
| Main flow of events | 1. User enters a message into a wide text input field spanning almost the bottom of the room client.<br>2. User clicks the "Send" button to the right of the input field or hits the "Enter" keyboard key.<br>3. Message is sent to the room server as plaintext via websockets along with the sender name/nickname prepended to the message.<br>4. Room server receives the message and echoes it to all connected clients.<br>5. All clients individually receive the message.<br>6. Clients append messages to a chatbox where all chat messages are displayed. |

Figure 6.1: Use Case for when a user inputs a message to the group it is connected to.

Establishing these use cases for the primary parts of the system gives a more specific and described insight as to what requirements is missing, needs adjusting or is excessive. Through this process a more descriptive requirement specification is created.

## 6.2 Finalized Requirement Specification

This table below illustrates the different requirements established from the analysis above. In addition, a general MoSCoW prioritization is performed, to indicate what requirements is the most important to implement. This is illustrated in the ”MoSCoW” column. The ”Use Case” column shows how the the different requirements fit the Use Cases that the requirements is derived from. The ”Evaluation” column indicates how each requirement is tested and evaluated.

The requirements is structured as main requirements that describe the use case its derived from and sub-requirements that go into detail. The verification column point to a specific subsection of the Verification chapter, where information about the verification of the requirement is available.

| ID | Description | MosCoW | Use Case | Verification |
|---|---|---|---|---|
| **R01** | Rooms is composed of a Kubernetes deployment. | M | | 9.2.1 |
| R01.1 | Room pods must be able to send and receive live packages (socket server). | M | | 9.2.2 |
| R01.2 | Room pods must be automatically created on demand. | M | | 9.2.3 |
| **R02** | Individual users is consisting of a served website | M | | 9.2.4 |
| R02.1 | Webserver must be scalable using Kubernetes | M | | 9.2.5 |
| **R03** | Connect to rooms | M | UC01 | 9.2.6 |
| R03.1 | Rooms each have a unique URL with its ID. | S | | 9.2.7 |
| R03.2 | RoomFinder service uses room ID to retrieve unique room pod URL from database | M | | 9.2.8 |
| R03.3 | Database containing room information | M | | 9.2.9 |
| R03.4 | Joining room requires entering of a nickname | S | | 9.2.10 |

Table 6.1: Part 1/2 of the finished requirement specification

| | | | | |
|---|---|---|---|---|
| R04 | Send message in rooms | M | UC03 | 9.2.11 |
| R04.1 | Messages sent to room pod socket server | M | | 9.2.12 |
| R04.2 | Users connected to room pod socket server receives messages | M | | 9.2.13 |
| R05 | Create rooms | M | UC08 | 9.2.14 |
| R05.1 | Creates room with unique ID | M | | 9.2.15 |
| R06 | User movement | S | UC04 | 9.2.16 |
| R06.1 | User can click on area on screen and move to that point | C | | 9.2.17 |
| R06.2 | User position updates and sent to other users in group periodically | S | | 9.2.18 |
| R06.3 | Other users positions is updated to represent user movement | S | | 9.2.19 |
| R06.4 | Movement updates voice stream service with new position. | S | | 9.2.20 |
| R07 | Video and voice connection. | S | UC07 | 9.2.21 |
| R07.1 | WebRTC connection | S | | 9.2.22 |
| R07.1.1 | Signalling server as a Kubernetes pod | S | | 9.2.23 |

Table 6.2: Part 2/2 of the finished requirement specification

## 6.3 Changes to requirements after implementation iterations

The above requirements is the foundation and guidance of which the design and implementation takes from in order to establish a fully fledged system of the same vision that was originally envisioned. However when going through these phases, it becomes more apparent which requirements might not be viable, is not necessary or is blatantly wrong. These are deemed not viable, either due to ignorance around the would-be-used technologies at the time, not necessary due to design changes made in the design phase or implementation phase, or having the prioritizing change due to time constraints.

## 6.4 Selected Technologies

In order to implemented the requirements elicited from the analysis of initial requirements, a set of technologies and tools have been chosen to facilitate the system. These technologies are outlined here, as well as how they are used in the project.

### 6.4.1 Kubernetes

Kubernetes is an open-source platform for managing containerized deployments, services or workloads. The main difference between traditional deployments and containerized deployments is that a container is its own virtualized environment, which makes the deployment of said containers simple across many different platforms. With Kubernetes it is possible to automatically scale these containerized services, in the case of an increase in traffic, expose the services, so that it is accessible to users on the internet, and load balance the network traffic to ensure the system is stable. [14]

This project can use Kubernetes to expose and load balance the network traffic to the system. Additionally, core containers make use of Kubernetes workload management, and will create more containers if the system requires.

### 6.4.2 Node.js

In contrast to normal Javascript, which is mostly used for client-side functionality, Node JS is a Javascript runtime environment, primarily used for back-end development such as web servers and API's. Node JS uses an event-driven architecture, which makes Node JS great at handling the many output and input operations a web-server receives.[**nodejs**]

Node JS can be used extensively in this project, to implement the components in a quick and easy manner, in addition to handling the website for the clients. Node JS applications are also easy to establish tests for, with a myriad of different packages providing tools for testing.

#### 6.4.3 Socket.io

Socket.io is a way to communicate real time with the server on the browser. Socket.io sets up a Websocket connection between the browser and the server, which is used to transport the messages to and from both ends quickly and reliably. [15]

This project can use Socket.io to facilitate connections between different clients of the system, in order to give clients the ability to communicate with each other. Additionally, it can be used to keep track of clients' positions and make sure that position was updated on every other client's applications. WebSockets are a strong alternative contender, but it is felt that the additional features provided by socket.io, such as rooms, could come in use.

#### 6.4.4 WebRTC

WebRTC, short for Web Real Time Communication, is a framework that makes it possible to set up real-time communication pipelines between users directly, instead of going through a web-server. This is called peer-to-peer communication. WebRTC can be used to communicate with audio, video or basic data streams. In order to create these peer-to-peer connections, the users must first negotiate and discover each other. This is done in a process called signalling. Using a web-server, the users are sending each other information about themselves, in order to be able to connect. If successful, a WebRTC connection is established. [5]

WebRTC is an excellent candidate for a communications framework used to enable audio and video communication, as indicated by its frequent use in competing products. This has been implemented in two different ways in this project. The standard peer-to-peer connection and a Media Relay Server (MRS).

#### 6.4.5 GitHub Actions

GitHub Actions is a newer CI/CD framework compared to alternatives such as Travis CI. However, GitHub Actions is a particular favored framework due to it's great accessibility, and ease of use. Naturally, it is also about as integrated into GitHub, the version control service that this project has made use of, as possible. GitHub Actions function similarly to other CI/CD frameworks by having developers write some sort of YAML file which contains instructions on what actions the CI/CD pipeline must perform, such as a Docker Build or Publish action.[16]

This project can make use of GitHub Actions to automate unit and integration tests of system components, and have the tests execute each time a change is pushed to the GitHub repositories.

## 6.5 Conclusion

Going from the overall requirements determined during the projects inception, to a specified requirements list is an important process in order to specify what requirements are necessary in order to move on to the design phase. Using these requirements it was possible to keep track on what was required to take into account during the design phase, needed to implement during the implementation phase, and important aspects to evaluate during the evaluation phase. A number of specific tools and technologies such as perhaps most notably Node.js, Socket.io, and WebRTC, are strong contenders for use in creating the system.

# 7 Design

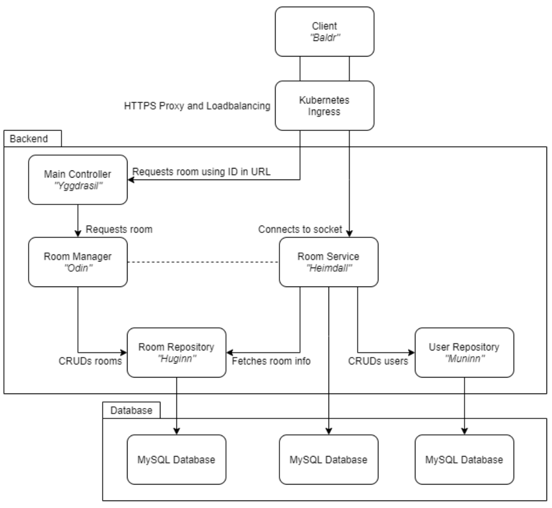
The goal of this chapter is to outline and discuss the overall architecture and design of the system, including the major components, their area of responsibility, and how they fit together. Additionally, a short discussion about the changes and simplifications the design have endured during development, and why these changes were made.

## 7.1 Architecture

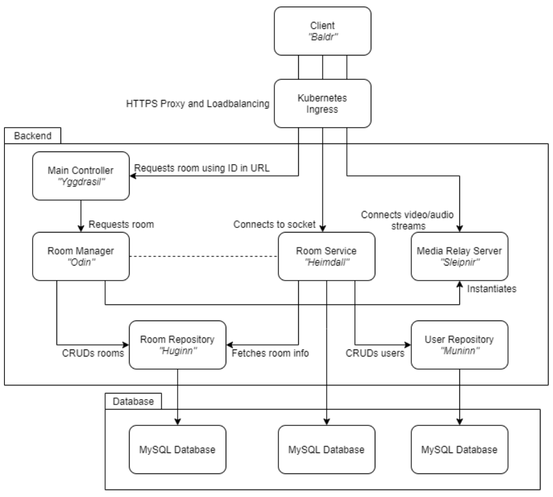
The overall architecture is based heavily on microservice principles, RESTful services, and ephemeral, stateless component instances. There is a single outlier to this in one particular component, the Media Relay Server, which is closer to a simpler client-server design. The primary driver for the choice of architecture is the need for the system to be easily horizontally scalable, as well as to be deployed through containerization technologies, in particular Kubernetes, which works well with such designs, as well as further provide reliability through redundancy.

## 7.2 Design

The system design is composed of eight major components used by a set of twin designs, those components being a main controller, a browser-based client, a room service, two types of room manager, a user- and a room repository, and finally a media relay server. The design is split into three discreet layers. These layers are the front-end with the browser-based client, the back-end where the system services lie within. Finally, the database layer which contains the databases needed to store persistent data. The front-end and the back-end are connected through a Kubernetes Ingress which acts simultaneously as an HTTPS proxy, as well as a loadbalancer.



(a) The final Odin design. Larger version available here: B.1



(b) The final Vili design. Larger version available here: C.1

Figure 7.1: Block diagrams of twin designs of the Odin and Vili designs. It can be seen how the majority of components are identical, except for the use of the different room managers, as well as the addition of a Media Relay Server in the Vili design

### 7.2.1 The Twin Designs

There are two designs for the system as shown in figure 7.1, one where voice and video communications are facilitated through P2P connections between clients, also known colloquially as the *Odin design*, and one known colloquially as the *Vili design* where voice and video communication is facilitated through a media relay server. These two designs are mostly identical in design, with most of the components shared between them. The room creator component is the major factor in each design[1], and as such is also where the names are derived from. The intent of the twin designs is to measure and compare their performances in accordance with the secondary project goal. The high level designs are shown in figures below.

Another major factor between the two designs is the WebRTC connections created between users. The P2P connection establishes a mesh connection between all other connected users, with all users having N-1 upstreams and N-1 down streams, N being user count. The media relay server (MRS) connection only allows users to have one upstream with the MRS. This stream is then shared between all other users connected. This would result in all users having 1 upstream and N-1 downstreams. An illustration of both of these implementations can be seen below [17]

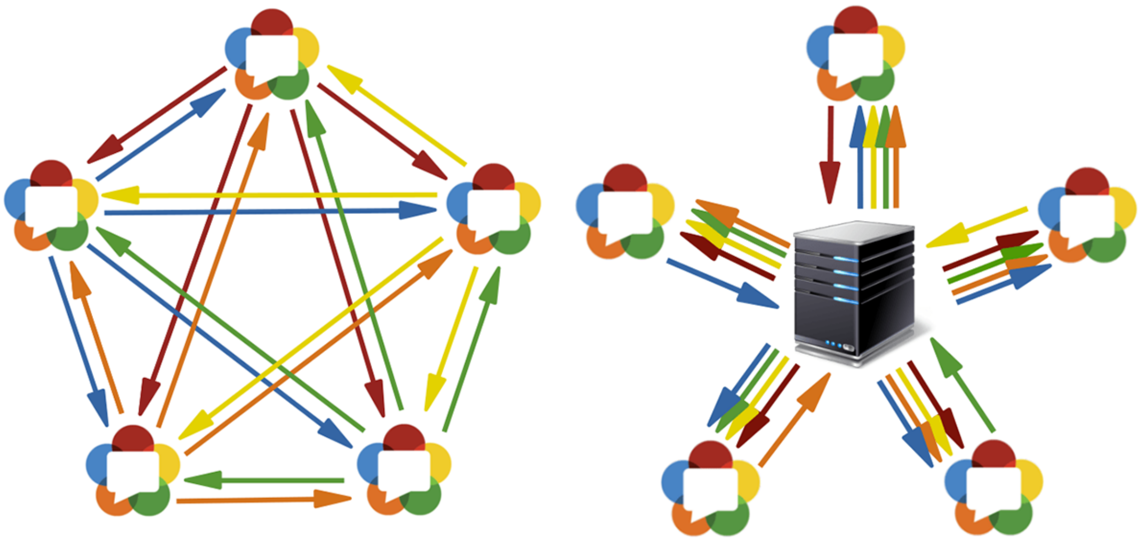


Figure 7.2: Two illustrations showing a Mesh P2P connection (Left) and SFU/MRS connection (right) courtesy of above citation.

### 7.2.2 Major Components

**Main Controller - Yggdrasil**

The Main Controller, nicknamed *"Yggdrasil"*, is responsible for, as the name implies, connecting the worlds. Acting mostly as entry for the rest of the system, it is the service that the client first interacts with when requesting a room, and it is what returns the room client web page to the clients browser. It fetches the rooms requested from a room manager, and depending on the room manager implementation, will return different types of room behaviour to the user.

**Browser-based Client - Baldr**

The browser-based client, also known as *"Baldr"* is what is returned to the client by Yggdrasil, and is what the client uses to login and interact with a room. It contains a number of sub components, including but not limited to one for connecting and interacting with the room server, one for connecting streams, and more.

[1]Vili never made it through to final implementation due to time constraints, but its influence remained.

**Room Service - Heimdall**
The service responsible for the rooms themselves is the one known as *"Heimdall"*. It provides a socket endpoint for the client to connect to and listen for events from, as well as send information to, as well as a socket for sending signalling information between users. Heimdall is responsible for maintaining the many rooms and the users connected to them, as well as keeping the many rooms separated. It uses user- and room RESTful repository services in order to store information. Furthermore, it directly accesses its own database in order to store meta-information about the connected sockets.

**Room Managers - Odin and Vili**
The two room managers, known as *"Odin"* and *"Vili"*. These, upon request from Yggdrasil, returns room information which differs depending on which of the managers is in use. Odin merely returns the information needed for the client to connect to the room service Heimdall, while Vili does the same as well as instantiates an instance of the Media relay server for the client to connect to.

**Room and User Repositories - Huginn and Muninn**
The room and user repositories, respectively nicknamed *"Huginn"* and *"Muninn"*, are simple services which provide a RESTful API, in order to allow other services to easily perform CRUD operations on rooms and users, without each consuming service needing to implement its own database access.

**Media Relay Server - Sleipnir**
The Media Relay Server named *"Sleipnir"* is as shown in figure 7.1 entirely unique to the Vili design, and acts as a relay between clients for voice/video communication, by having the clients connect to Sleipnir, which then relays clients voice/video streams to each other.

### 7.2.3 Major Changes Through Development

Throughout development, the design endured a number of simplifying changes to an initial more complex design with unique service instances for each user and room, all managed by complex programmatic manipulation of the Kubernetes cluster the system was deployed on. The original designs, while intriguing, were deemed too impractical for implementation, and simplified to what is documented in this chapter. A block diagram of the initially proposed design is available as appendix D.1 and E.1.

### 7.2.4 Pros and Cons

While the pros of a microservice-based design are clear in terms of scalability and reliability, there are however some significant cons with such a design. In particular as there is a significant development overhead in both complexity and labor needed to implement it. Following such a design poses limitations, such as the aforementioned statelessness and ephemeral nature, requiring further complexity when a component, for instance, need some sort of persistent storage. This is further discussed in the Evaluation and Reflections chapter.

### 7.2.5 Merging the Twin Designs

The possibility of merging the two designs into a single one, where the type of room would be selectable by the end-user, perhaps through a subscription or different endpoints, has been proposed and debated. The merging would in theory be simple, by merely adding support for accessing different room managers to Yggdrasil. However, it was deemed that this would likely lead to a number of unforeseen issues that would not be worth the trouble, and that time was better spent elsewhere.

## 7.3 Conclusion

The system architecture is for the vast majority of it based on microservice architecture principles, with the exception of a single component, which is closer to a client-server style architecture. The microservice-based architecture was chosen in particular for its scalability and good fit with containerization technologies. The design itself consists eight total components, used in two separate designs, which both share six central components. There are some additional parts to the design such as databases and a Kubernetes Ingress acting as a proxy and loadbalancer. The design has undergone a number of simplifying changes throughout development, as the initially proposed design was deemed too unfeasible and impractical to implement.

# 8 Implementation

The intent of this chapter is to explain and document the implementation of the CoffeeBreak system. The chapter will go over each of the eight components described in the previous Design chapter, and explain how they were implemented as well as the languages, technologies, and tools used.

All components but one are written in JavaScript using Node.js, and all use a small set of node modules to function. These modules are, in no particular order:

- Express, a webserver framework used to create and provide HTTP endpoints.[18]
- CORS[19], JSON[20], and cookie-parser[21] middleware for the Express framework, which respectively provide CORS, JSON and cookie support.
- Socket.io, a high-level client-server bi-directional event based communication framework with a variety of useful features such as room and namespace support.[15]
- Axios, a tool for creating and sending HTTP requests.[22]
- MySQL2, a framework for connecting, and making database queries, to a MySQL database.[1][23]

The final non-JavaScript component is the media relay server, which is written in Python and uses a set of plugins which are outlined in the components own section of this chapter.

All chosen technologies are chosen over alternative equivalents for no particular reason other than developer preference.

The components source code are available on GitHub here: https://github.com/CoffeeBreak-Architecture/[24]

## 8.1 Main Controller - Yggdrasil

As described in the Design chapter, Yggdrasil is responsible for providing an entryway for the rest of the system, and take in initial room requests from users. It is written in JavaScript using Node.js and Express as a web server framework, as well as a number of smaller, arbitrary modules. Being mostly boilerplate for the rest of the system, this component is simple in internal design, and the web server part consists of no more than about 80 lines of JavaScript code in total.

It needs to be linked to the room manager through the environment variable $ROOM\_MANAGER\_URL$, which should point at the room manager service.

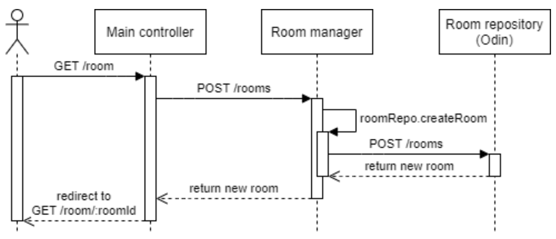
### 8.1.1 index.js

The entry script for the component is composed mostly of two primary Express-based HTTP endpoints, as well as an additional one needed by the Kubernetes environment for health checks. Furthermore, the Express module is requested to make use of the CORS, JSON, and cookie-parser middleware. In addition, it provides access to the socket.io node module so that clients may access the client framework.

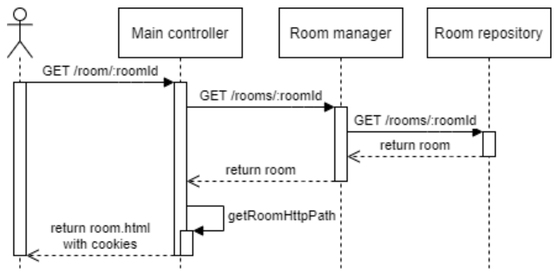
[1]The MySQL2 node modules was chosen over MySQL module, due to the MySQL module being outdated at the time of implementation.

**GET /room**
The first of the two primary HTTP endpoints. This is used by the client to request a new room. It functions simply by requesting the creation of a new room using the $createRoom()$ function from manager.js, and then redirecting the client to GET /room/:roomId where :roomId is the newly created rooms ID.

**GET /room/:roomId**
The second of the two primary HTTP endpoints. This is what clients use to connect to a particular room, such as if they have been provided a URL by a peer, or have been redirected by the previously mentioned GET /room endpoint. It works simply by requesting room from the room manager, and sending the room ID, as well as connection information, back to the client using cookies, as well as returning the room client HTML document which forms the base of the browser-based client. In this chapter, a colon prefixed in a part of the url as it is done in :roomId indicates that it the part is a parameter, and can be anything.



(a) The sequence of events when making a GET /room request for a new room.



(b) The sequence of events when making a GET /room/:roomId request for a specific room.

Figure 8.1: Sequence diagram for the two major endpoints provided by the main controller. Both figures are largely similar in sequence, except for the type of request made to room manager and repositories. Notice in as well how figure a redirects to the sequence in b at the end of the sequence.

**GET /**
This final endpoint defined by Yggdrasil is the one used by the Kubernetes environment for health checks. It merely needs to respond with anything, so as to function as a ping.

### 8.1.2 manager.js

The manager module is a simple Node module defined in manager.js, which provides two functions for easier and more testable access to the room manager component. The functions $createRoom()$ and $getRoom(roomId)$ are defined, and both simply use the Axios node module, a module which allows one to easily send asynchronous HTTP requests, to respectively send a POST request to the room managers POST /rooms endpoint, and a GET request to the room managers GET /rooms/:roomId endpoint.

## 8.2 Browser-based Client - Baldr

The browser-based client is what is used by the client to connect to and interact with the rooms themselves. It consists of a number of files provided by Yggdrasil, starting with the HTML document received when first requesting a room.

### 8.2.1 room.html and style.css

The room.html document is as mentioned the document first provided by Yggdrasil when requesting a room, and is where the structure of the web page itself is defined. The

HTML structure is simple, and merely defines a number of <div> tags as containers to be filled out, such as list of members, a hidden list of video streams, and a chatbox for chat messages. Furthermore, it defines a large canvas which dominates the middle section of the screen, where the virtual room itself is rendered, along with the user avatars and their video streams. Finally, a chat input area is defined in the bottom of the page. The style.css document is merely a stylesheet which helps position and color the various HTML elements. An image of the resulting web page is available in figure 11.1 in the User Manual chapter.

### 8.2.2 common.js

The common.js script is a simple script with a few utility functions an an initializing $onLoad()$ function. The $onLoad()$ function listens for keydown events for the enter key, and sends the currently written chat message when the event occurs. Additionally, it sets up the canvas. Utility functions $getCookie(name)$ and $invert(object)$ are defined, which respectively provide easy access to cookies, and invert JavaScript objects.

### 8.2.3 room.js

The room.js script is a longer file with the primary responsibility of interacting with the room service through the socket.io client API, as well as updating all room-related information on the web page, as well as for detecting when the user clicks the room canvas. Individual units of functionality are usually only a few lines, and work to connect the end-user with the room. It connects to a remote socket.io server using a URL provided through a $socketUrl$ cookie by Yggdrasil, and listens to a number of socket events, as follows:

**connect**

Emitted by socket.io itself when it connects to the back-end socket server. When this occurs, the client prompts the user for a name, and emits a ”login” message to the socket, along with the chosen name and ID of room the client wishes to login to. The room ID is provided to the client by the main controller as a cookie. The full sequence of events when logging in is documented in figure 8.2.

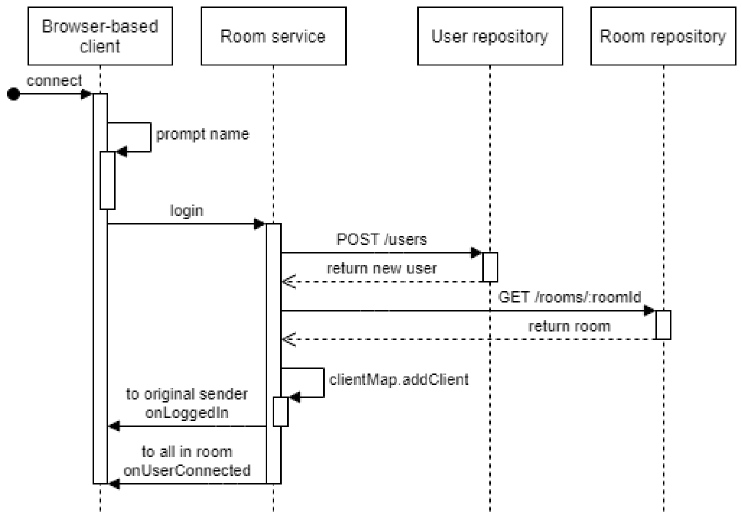


Figure 8.2: The sequence of events when the user logs in to the room service, and how the room service makes use of an internal client map as well as room- and user repositories to register and keep track of users. Room service, client map, and the repositories are all documented later in this chapter.

**onLoggedIn**
Response to the $login$ message previously sent. Comes with a list of existing connected users, the ID assigned to the user, as well as room information.

**nearby**
Updates the client on which other users are considered ”nearby”, as well as the arbitrary distance threshold to be used when calculating audio gain. The client additionally calls the $callNearby()$ function which is described in the streams-p2p.js section.

**onUserConnected, onUserDisconnected, onMovePlayer, onChatMessage, and onNameChanged**
These all merely update the client accordingly with received information, such as adding the new user to a local list of connected users, removing users from said list, moving them, adding a chat message to the chat list, and updating a users name. Additionally triggers updates of appropriate HTML elements through a variety of utility functions.

### 8.2.4 canvas.js

The canvas.js script is responsible for rendering the canvas and handling clicks. The scripts functionality is implemented as a series of functions that divide-and-conquer the rendering of all the user avatars in the room, either as a simple stick figure, or as a webcam stream if one is available. Rendering is done continuously through the built-in $requestAnimationFrame(...)$ function. The sequence of function execution is documented in figure 8.3.

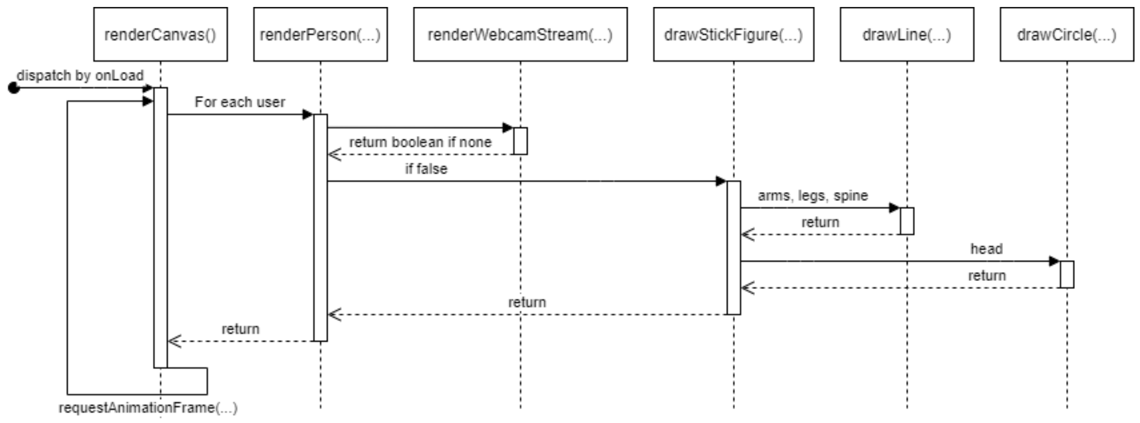


Figure 8.3: The sequence of function execution when rendering the canvas. Shows the divide-and-conquer approach to rendering the avatars within the canvas, and how whether or not rendering video stream vs rendering a stick figure is conditional. Notice the final $requestAnitationFrame(...)$ call at the end of $renderCanvas()$.

In order to handle canvas clicks, the function $onCanvasClick(...)$ subscribes to the ”click” event of the canvas, which is executed when the user clicks the canvas, in order to move their avatar. This function receives information about the click event in ”browser-space” as a parameter, and uses this to compute a corresponding position of the click in ”world space”, which the avatar is then moved to. It is worth keeping in mind that the implementation is not perfect and there is a slight offset, but it is the closest mapping after a while of trial-and-error due to unclear documentation.

### 8.2.5 stream-players.js

The stream-players.js script handles the media additions to the client, whether its audio or video received from the user itself or added from a remote user via WebRTC. When a media stream is received, it is split into its respective audio and video tracks. An addition to a dictionary, with the user-id as key, is created, where the respective tracks are added to for easy retrieval if required in other scripts. While the video track is only added as a video element, the audio track is used to create an $AudioContext$ and insert a $GainNode$, the latter which makes it easy to adjust the individual streams audio levels. This enables the individual user to compute the volume of each other user near them based on their distance received by the previously mentioned $nearby$ event.

### 8.2.6 streams-p2p.js - peer to peer configuration

The stream-p2p.js script handles the signalling and WebRTC connections between other users. When the client connects to a room it automatically connects to a socket server handling signalling. A number of events is then established client side that will execute different pieces of code depending on the message type received. In order to establish a WebRTC connection, the two users wanting to connect need to “signal” information between each other. This is done through the aforementioned signalling server.

When the signalling starts, a Peer Connection is created. This is the object which facilitates the WebRTC connection. An offer is then sent to all users in the vicinity with the information about the created local Peer Connection, through the $callNearby()$ function. Other users receiving the offer registers the information in their Peer Connection, and sends back their information. In addition to this, the network information is also required to be shared with potential users. This is done with an event listener added to the Peer Connection. When a Peer Connection receives an ICE candidate, the client sends it automatically to all other users via the signalling server. When a clients receives an ICE candidate, it is automatically added to their Peer Connection.

In addition to handling signalling, the Stream-p2p.js script also handles dead connections. An event listener is also applied to the clients Peer Connections that checks if the connection is stopped. If it is, it is deleted and stopped. Below is a sequence diagram showcasing the WebRTC connection flow of events

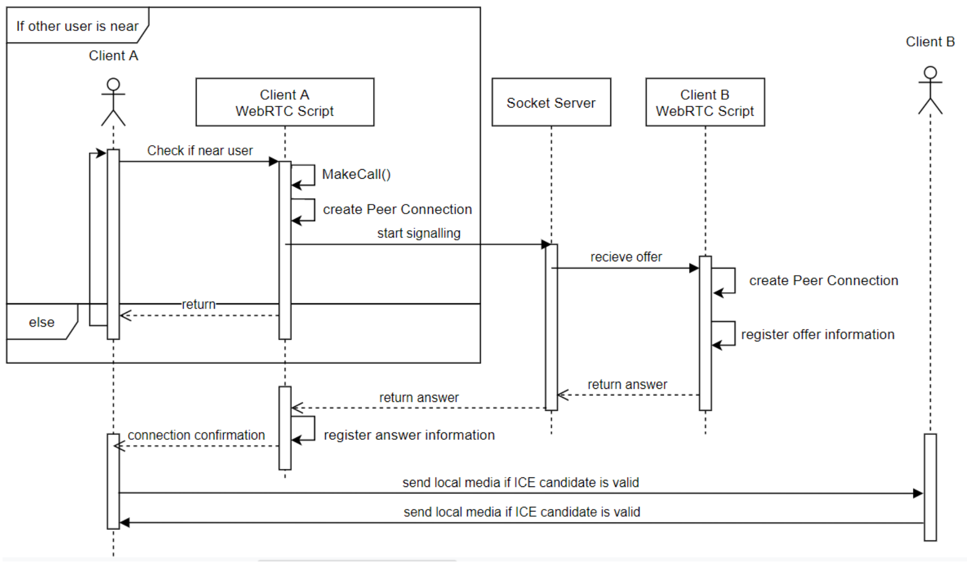


Figure 8.4: Sequence diagram showing the flow of events when users connect through WebRTC

## 8.3 Room Service - Heimdall

The room service known as Heimdall is responsible for handling the rooms themselves and the users connected to them. It is as most other services written in JavaScript using Node.js. The service is split in two major modules, rooms.js for handling rooms themselves, and signalling.js for signalling between users. Both these modules use socket.io, and socket.io namespaces to differentiate from one another. Additionally, a clientMap.js module is used to provide persistent mapping between client IDs and the IDs used internally by socket.io. Two repository modules provide access to user and room repository services. The index.js entry script is merely responsible for initializing the rooms and signalling modules with socket.io.

It needs a number of environment variables to function. A set of variables for connecting to a database: $MYSQL\_HOST$, $MYSQL\_USER$, $MYSQL\_PASSWORD$, and $MYSQL\_DATABASE$ to be set up according to MySQL specifications. Additionally, it needs the environment variables $USER\_REPOSITORY$ and $ROOM\_REPOSITORY$ to be URLs of respectively user- and room repository services.

### 8.3.1 rooms.js

This module is the most complex of Heimdall and as mentioned is responsible for the rooms themselves, and the users connected to them. It can be considered the server-side equivalent of the room.js script from the browser-based client, and handles all of the room-specific events. It consists primarily of following five event listeners listening to emissions from clients.

**login**
Emitted by the client when they wish to login to a particular room. It proceeds to request the creation of a new user from the user repository, as well as fetch room information and room members. The client is added to the clientMap modules client map. Room information, members, user information is all emitted to the client through the ”onLoggedIn” event. Additionally, the ”onUserConnected” event is emitted to the entire room in question, with information about the new user. The full sequence is available in figure 8.2

**onChatMessage**
Emitted by clients when sending a chat message. Merely relays chat message to the rest of the clients in the sending clients room.

**onMovePlayer**
Emitted by the client when moving their avatar. Updates the clients user information through the user repository, and fetches an array of the users considered ”nearby” to the client. Relays the movement to the rest of the clients room, and finally emits the ”nearby” event to the client, with the array of nearby users.

**onNameChanged**
Emitted by the client when requesting to change their name.[2] Merely updates the clients users name through the user repository, and relays the new name to all other clients in the clients room.

**disconnect**
Emitted by socket.io itself when a client socket is disconnected. Requests the deletion of the user from the user repository, and relays the disconnection to all other clients in the clients room through the ”onUserDisconnected” event.

[2]This feature is left out of the final version of the browser-based client, but the functionality is left in behind-the-scenes in case it would be re-implemented in the client.

### 8.3.2 signalling.js

This module is responsible for providing signalling between users, or users and the media relay server, so that they may call each other in order to open WebRTC connections. The module listens to two events emitted by clients, ”login” and ”message”. ”login” is emitted by the clients when logging in to to the room similarity to the rooms module, though it merely adds the client to the client map. The ”message” event on the other hand receives a message which has information about a sender, the ID of an intended receiver, a type, and a payload. Though the client map, the module finds out which connected socket to send the message to, and sends it.

### 8.3.3 clientMap.js

This module provides a set of functions for storing and accessing mappings between client IDs and the IDs used internally by socket.io to differentiate clients. The mappings are stored in MySQL database, with each mapping consisting of a socket ID, a client ID, and which namespace the mapping belongs to. The database is directly accessed using the MySQL2 module. This is used extensively throughout the service.

### 8.3.4 userRepo.js and roomRepo.js

These two modules are merely responsible for providing access to the user- and room repository services. They use Axios to make the HTTP requests.

## 8.4 Room Manager Odin

The ”Odin” room manager is responsible for creating simple client-server type rooms with P2P voice/video communication. The service very simple, consisting of barely 40 lines of JavaScript split into the index.js entry script, and the roomRepo module. It uses the Node.js framework, as well as Express to provide two webserver endpoints. These two endpoints are ”POST /rooms” and ”GET /rooms/:roomId”, the first of which requests the creation of a new room through the roomRepo module, while the other requests a particular room through the roomRepo module, and either returns the room, or a status code 404 if the requested room could not be found. The roomRepo module uses Axios to make HTTP requests, and the rooms created contains connection information which is predefined through environment variables, with this connection information usually pointing at the Heimdall room service.

This component merely needs two environment variables. $ROOM\_REPOSITORY\_URL$ pointing at the room repository service, and $SOCKET\_SERVER\_URL$ pointing at the room service, to provide as connection information to the clients.

## 8.5 Room Repository - Huginn

The room repository is a simple service designed to be RESTful access to CRUD room resources. It provides a layer between the rest of the system and the databases in which persistent data is stored. As with most other services, it is written in JavaScript using Node.js, and Express for providing endpoints. It uses the MySQL2 node module to connect to a MySQL database. The database is automatically initialized on first connection, if it is not already initialized, thus avoiding cumbersome database setup during deployment and testing.

This component needs environment variables to know how to connect to its database. The variables $MYSQL\_HOST$, $MYSQL\_USER$, $MYSQL\_PASSWORD$, and $MYSQL\_DATABASE$ needs to be filled out according to MySQL specifications.

## 8.6 User Repository - Muninn

The user repository is almost identical in implementation to the room repository Huginn, except naturally that it provides CRUD access to user resources instead.

## 8.7 Media Relay Server - Sleipnir

The Sleipnir service is a MRS implementation, created using the AioRTC python library. Instead of users connecting to each other directly, they connect to this MRS. When a user joins a room they automatically connect to the MRS as a client. This is the users local media being sent to the server for other users to connect to. This media is then played using a MediaBlackhole which takes in media streams, plays them, and then discards the tracks. The clients media tracks are saved to the server in a client dictionary that takes the clients userid as key. These tracks are added to a MediaRelay, of which can be subscribed to, granting easy access to the clients media tracks to multiple users. When a client is nearby another client, a listener connection is established for each client. A listener connection receives the corresponding media tracks in the client dictionary and subscribes to the MediaRelay. These tracks are then added to the listener Peer Connection as their local tracks and then triggers the client remote stream event listener. Below is two sequence diagrams illustrating the functionality

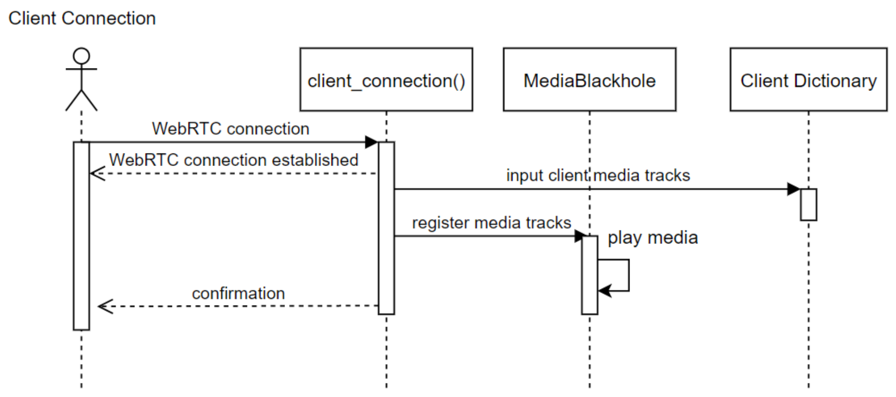


Figure 8.5: Sequence diagram showing the flow of events when connecting as a client

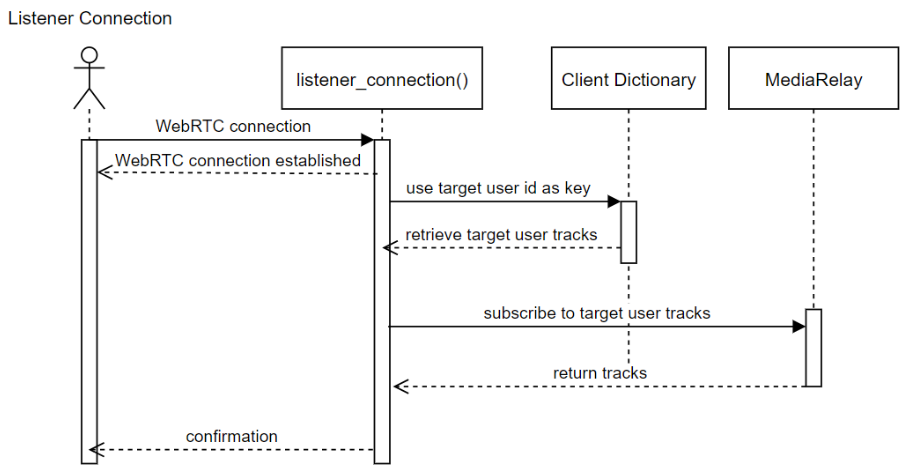

Listener Connection
listener_connection()
Client Dictionary
MediaRelay
WebRTC connection
WebRTC connection established
use target user id as key
retrieve target user tracks
subscribe to target user tracks
return tracks
confirmation

Figure 8.6: Sequence diagram showing the flow of events when connecting as a listener

### 8.7.1 Client Side changes - MRS configuration

The main functionality of the WebRTC connection is the same. However signalling is not done using the signalling server, but a series of CRUD operations between the MRS and the Client. When the user connects to a room a client connection is established, and Peer Connection information is sent between the MRS and the client The Client then awaits ICE candidates until all candidates is gathered and then returns the information to the MRS.

If a client, A, moves near another client, B, a listener connection is established. A connection is requested with Client B's media tracks, while a signal is sent using the socket server to Client B, which then establishes a connection with Client A's media tracks. This is illustrated with the sequence diagram below.

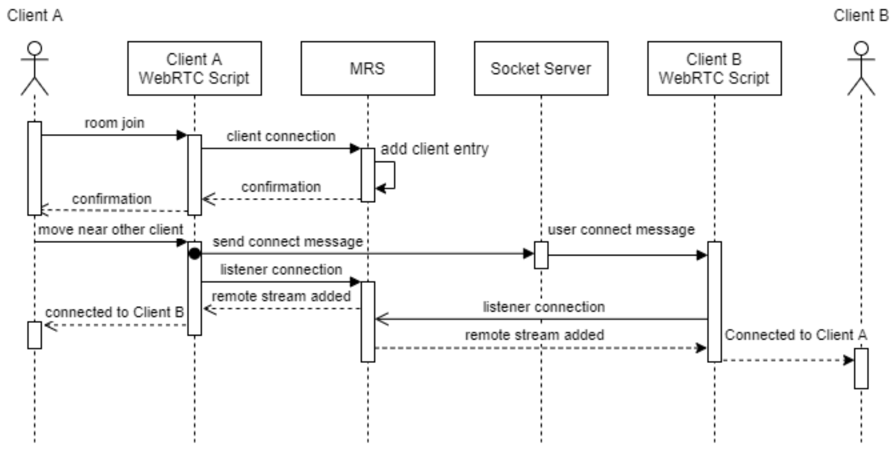


Figure 8.7: Sequence diagram showing the sequence of events when connecting to the MRS in the Vili design

# 9 Verification

This purpose of this chapter is to outline the methods of verifying that the CoffeeBreak system implements the required features. The technologies used for verification, including frameworks, libraries, and services used, are all also outlined. Each successfully implemented feature requested has a section of this chapter dedicated, explaining how it has been verified, as well as potential flaws with the verification for that particular feature, or, should it be the case, why the particular feature has not been verified.

## 9.1 Methodology and Technology

For all JavaScript components, including the scripts in the browser-based clients, the testing framework Mocha is used, alongside the Chai assertion library, for both unit- and integration testing. GitHub Actions is used to automate the testing process through continuous integration (CI), in order to ensure everything pushed to the remote version control repository is tested.[1] The one component implemented using Python, the media relay server, is merely tested using user acceptance test and thus does not use any particular technologies for testing, due to it being considered non-critical for the product itself, being merely an extra feature for the secondary project goal.

All unit and integration tests are available directly in the source code repositories, usually located in a folder called ”test”, ”tests”, or something similar, located in the root folder. This is with the single exception of the browser-based clients tests, which are located in the public/client_test folder within the main controllers repository. Tests are usually split into multiple files, depending on what type of testing or which module is being tested.

The CI scripts are located under the .github/workflows folder from the root folder. These might not be visible by default in some operating systems, but can easily be located on the repository website.

## 9.2 Requirement Verification

This section lists all the requirements found during analysis, how they have been verified along with important code snippets. Each subsection is titled with the requirements ID and some context on what is being tested, loosely based on the original requirement description.

### 9.2.1 R01 Pod-based Rooms

Verified through running commands in the Kubernetes API, where it is possible to check running pods in the cluster. The socket-server, Heimdall, that handles room connections and messages sent to rooms is present as a deployment.

### 9.2.2 R01.1 Room is Socket Server

This is verified indirectly through the client integration tests, as they all emit and receive events to and from the room socket server. For instance, the following test in client integration tests ensure that the client has been assigned an ID, something which cannot occur without outside manipulation, unless the client has logged in to the room socket server.

[1]True continuous integration tests are only done for a single component, the room repository, as a proof-of-concept. See the .github/workflows/integration.yml file in the repository repository for more information.

```
it ('Logs in', async function () {
    await delayUntill (() => localUserId)
    expect(localUserId).to.exist
})
```

Note that the $delayUntill()$ function is merely a utility for awaiting until the given predicate holds true. It is used extensively in client integration tests, and in itself prohibits the tests from concluding before the condition holds true. With the Mocha testing framework, tests are automatically failed if not concluded within a given threshold, by default 2000ms.

### 9.2.3 R01.2 Room Pods Automatically Created

This requirement is fulfilled through the configuration of the Kubernetes deployment of the room socket server service. The deployment consists of a Replica Set, of which can create more replicas of itself depending on traffic and is verified through stress testing of the service. When the system was stress tested, the amount of pods increased to match the need.

### 9.2.4 R02 Browser-based Room Client

Verified in main controller integration tests for both endpoints GET /room and GET /room/:roomId, as well as a negative test of GET /room/:roomId, by asserting that the endpoints return an HTML document when expected. A snippet of code as an example of one of these test, this being for GET /room:

```
it ('GET /room', async function () {
    let response = await axios.get(url + '/room')
    expect(response.status).to.equal(200)
    expect(response.request.res.responseUrl).to.exist
    expect(response.data.startsWith('<!DOCTYPE html>')).to.be.true
})
```

While it is generally considered a good rule-of-thumb to only have a single assertion per test case, so that no two assertions are dependant on each other, this rule-of-thumb was not used in most tests primarily to avoid even more duplicate code that the tests already had.

### 9.2.5 R02.1 Scalable Webserver

Same verification method as R01.2

### 9.2.6 R03 Connect to Rooms

This is verified through the test case shown with R01.1 verification, see 9.2.2.

### 9.2.7 R03.1 Room URL contains Room ID

Verified in main controller integration tests of the GET /room/:roomId endpoint, where the test first requests a new room through GET /room, then requests the room using the redirect URL returned by GET /room. The returned URL is tested against a Regular Expression in order to verify that the URL contains a room ID.

```
it ('GET /room/:roomId', async function () {
    let response = await axios.get(url + '/room')
    let responseToRoom = await axios.get(response.request.res.responseUrl)

    let redirect = response.request.res.responseUrl
    let id = redirect.substring(redirect.lastIndexOf('/') + 1)

    const uuidRegex = /\b[0-9a-f]{8}\b-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-\b
        [0-9a-f]{12}\b/
    expect(uuidRegex.exec(id).length).to.equal(1)

    expect(responseToRoom.status).to.equal(200)
    expect(responseToRoom.data.startsWith('<!DOCTYPE html>')).to.be.true
})
```

### 9.2.8 R03.2 Access Room Using ID

Verified in the same test case as described for R03.1 above. See 9.2.7

### 9.2.9 R03.3 Rooms Stored In Database

As described in previous chapters, room storage is done through the room repository component. Each endpoint provided by the room repository is tested, including POSTing new rooms. Two tests, one for POSTing a new room, and one for getting a room with a particular ID verify this particular feature.

```
it ('POST /rooms', async function () {
    let post = await axios.post(url + '/rooms', {name: 'someName', socketUrl:
        'http://someSocketUrl', signallingUrl: 'http://someSignallingUrl'})
    expect(post.status).to.equal(201)
})

it ('GET /rooms/:id', async function () {
    let post = await axios.post(url + '/rooms', {name: 'someName', socketUrl:
        'http://someSocketUrl', signallingUrl: 'http://someSignallingUrl'})
    let room = await axios.get(url + '/rooms/' + post.data.id)

    éxpect(room.data.name).to.equal('someName')
    expect(room.data.socketUrl).to.equal('http://someSocketUrl')
    expect(room.data.signallingUrl).to.equal('http://someSignallingUrl')
    expect(room.status).to.equal(200)
})
```

Notice that the second test in this snippet both POSTs a new room as well as fetches it, as the database is cleaned up in between different tests. Both tests are still included to verified that the POST functionality works independently.

### 9.2.10 R03.4 Requests Nickname on Room Join

Verified by a small test from the browser-based clients integration tests. The testing client sets $window.prompt()$, the function used to prompt the user for a nickname, to always return ”John Doe”. After the user has logged in, it then verified that the user has logged in with the nickname ”John Doe”.

```
window.prompt = function(str) { return 'John Doe' }
...
it ('Prompts nickname', async function () {
    expect(getLocalUser().nickname).to.equal('John Doe')
})
```

It is worth keeping in mind that this test occurs after the login test, meaning that the user is always logged in at this point. The testing framework runs all tests sequentially, regardless of any promises or any other sort of asynchronous programming. The two lines separated by three dots are not from the same file, they are merely displayed in the same snippet for brevity. The login occurs between the two parts of the code snippet.

### 9.2.11 R04 Send Messages to Room

This is verified through the clients integration testing with the room service. It sends a message with some dummy content to the room server, and awaits the message being added to the clients list of messages, then asserts that the content of the received message is what was originally sent.

```
it ('transmitChatMessage(author, content)', async function () {
    messages = []
    transmitChatMessage('someId', 'Some message content')
    await delayUntill(() => messages.length == 1)
    expect(messages[0].author).to.equal('someId')
    expect(messages[0].contents).to.equal('Some message content')
})
```

### 9.2.12 R04.1 Messages Sent Through Room Service

Already verified through the verification of requirement R04 just above, as it is through integration testing.

### 9.2.13 R04.2 Users in Room Receives Message

For the most part same as verification for R04.1. That other members of a room also receives the message is done through simple manual user acceptance testing.

### 9.2.14 R05 Create Rooms

While this is already tested by the main controller integration testing described with the verification for R02 and R03.1, their intent is mostly simply to test the main controller itself. Creating rooms is specifically tested through integration tests from the room manager, specifically the Odin room manager.

```
it ('POST /rooms', async function () {
    let response = await axios.post(url + '/rooms')
    expect(response.data.id).to.exist
    expect(response.status).to.equal(201)
})
```

The POST request returns the room and afterwards asserts that it exists through the room ID. The POST /rooms endpoint of the room repository is also tested in the repositories own testing.

### 9.2.15 R05.1 Room Has Unique ID

To verify this, 100 rooms are created in room repository tests, and all compared against every single other room to ensure that none of them share their ID.

```
it ('IDs are unique', async function () {
    posts = []
    for (let i = 0; i < 100; i++) {
        let post = await axios.post(url + '/rooms', {name: 'someName',
            socketUrl: 'http://someSocketUrl', signallingUrl: 'http://
            someSignallingUrl'})
        posts.push(post)
    }
    expect(posts.every(x => {
        let identicals = 0
        posts.forEach(y => {
            if (y.data.id == x.data.id) {
                identicals++
            }
        })
        return identicals == 1 // Each room will always come across itself.
    })).to.be.true
})
```

Notice that since UUIDs are used for room IDs, there is a infinitesimally small chance that two rooms actually do share their IDs, though in practice it can be safely assumed that this *hopefully* will not occur within the lifetime of this universe.

### 9.2.16 R06 Users Can Move

This feature is verified through the integration testing of the browser-based client. Similarity to verifying R04, this is done by sending the movement request and awaiting the response from the room service.

```
it ('moveLocalUser (x, y)', async function () {
    moveLocalUser(300, 300)
    await delayUntill(() => getLocalUser().x == 300)
    expect(getLocalUser().x).to.equal(300)
    expect(getLocalUser().y).to.equal(300)
})
```

### 9.2.17 R06.1 Users Move By Clicking Canvas

This is merely verified through manual user acceptance testing, due to the feature being dependant with directly handling an event raised by the canvas itself. This could potentially be made more testable by splitting the handling of the event and the computation of movement in two different functions, but even so, the verification of clicking would only truly be verified by an actual click.

### 9.2.18 R06.2 User Movement Sent to Room Members

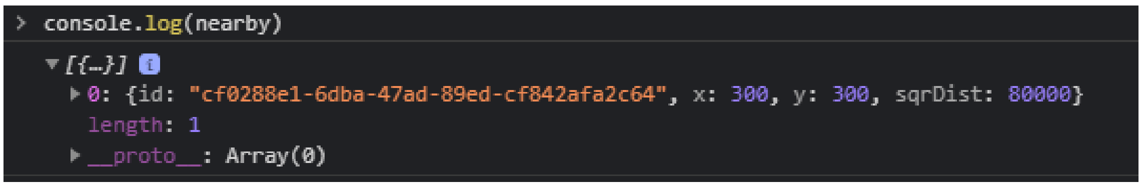
To some extend already verified by the verification of R06, though it is further verify by simple manual user acceptance testing.

### 9.2.19 R06.3 Room Members Movement Sent to User

Same as R06.2.

### 9.2.20 R06.4 Movement Updates Nearby Users

This has proven difficult to verify using automated testing methods alongside the rest of the tests, as it effectively requires two clients due to the implementation, something which could easily interfere with other tests. However, it can easily be verified by manually verifying by logging two browsers in to the same room, moving each avatar close to each other, and executing $console.log(nearby)$ in the browser console, and verifying that there is a single element within the array. This is shown in figure 9.1.



```
> console.log(nearby)
  ▼[{…}] i
   ▶0: {id: "cf0288e1-6dba-47ad-89ed-cf842afa2c64", x: 300, y: 300, sqrDist: 80000}
    length: 1
   ▶__proto__: Array(0)
```

Figure 9.1: Image showing the execution of $console.log(nearby)$ after two browsers joined the same room and moved near to each other. Shows how the printed object contains a single ”nearby” user. The actual numbers should be disregarded.

There was an actual test case implemented for this particular feature, but it was deeply flawed and would always pass no matter the state of the system, it has since been removed but can be found in previous versions of the system.

#### 9.2.21 R07 Video and Voice Connection

This particular feature is not automatically tested due to the complexity of setting up such a test, as well as the many unknowns involved. Through manual user acceptance testing, the feature is deemed *verified enough*. It can be verified that a connection is established by examining the chrome://webrtc-internals/ page included with Google Chrome. This page includes a large amount of information about current WebRTC connections.

#### 9.2.22 R07.1 WebRTC Connection

Same as R07.

#### 9.2.23 R07.1.1 Signalling Server

Signalling is verified in the client integration tests, by sending a message to itself through the signalling server, and waiting for a response, effectively pinging the signalling server.

```
it ('sendMessage(...)', async function () {
    await delay(1000)
    let response;
    signalling.on('message', message => {
        response = message;
    })
    sendMessage({ type: 'type', contents: 'contents' }, localUserId)
    await delayUntill(() => response)
    expect(response.message.type).to.equal('type')
    expect(response.message.contents).to.equal('contents')
})
```

## 9.3 Conclusion

For almost all components in the system, the Mocha testing framework is used alongside the Chai assertion library. This holds true even for the browser-based client. For continuous integration, GitHub Actions is used to automate the testing process. The one component not written in JavaScript has not been tested, as it is not considered critical for the project itself, instead being an extra feature. The large majority of requirements has been verified through unit- or integration tests, with a few exceptions due to prohibitive complexity, or some other reason. These are merely verified through manual user acceptance test or an arbitrary manual testing method.

# 10 Deployment

The deployment of the system is done on a Kubernetes cluster hosted on Google Cloud. The end result deployment architecture is almost identical to the established architecture in the design phase. Nothing of particular importance had to change in order to deploy the established design on a Kubernetes cluster, though some very minor changes had to be done. The deployment architecture shown below, is of the Odin configuration, which uses the Peer-to-peer WebRTC solution.

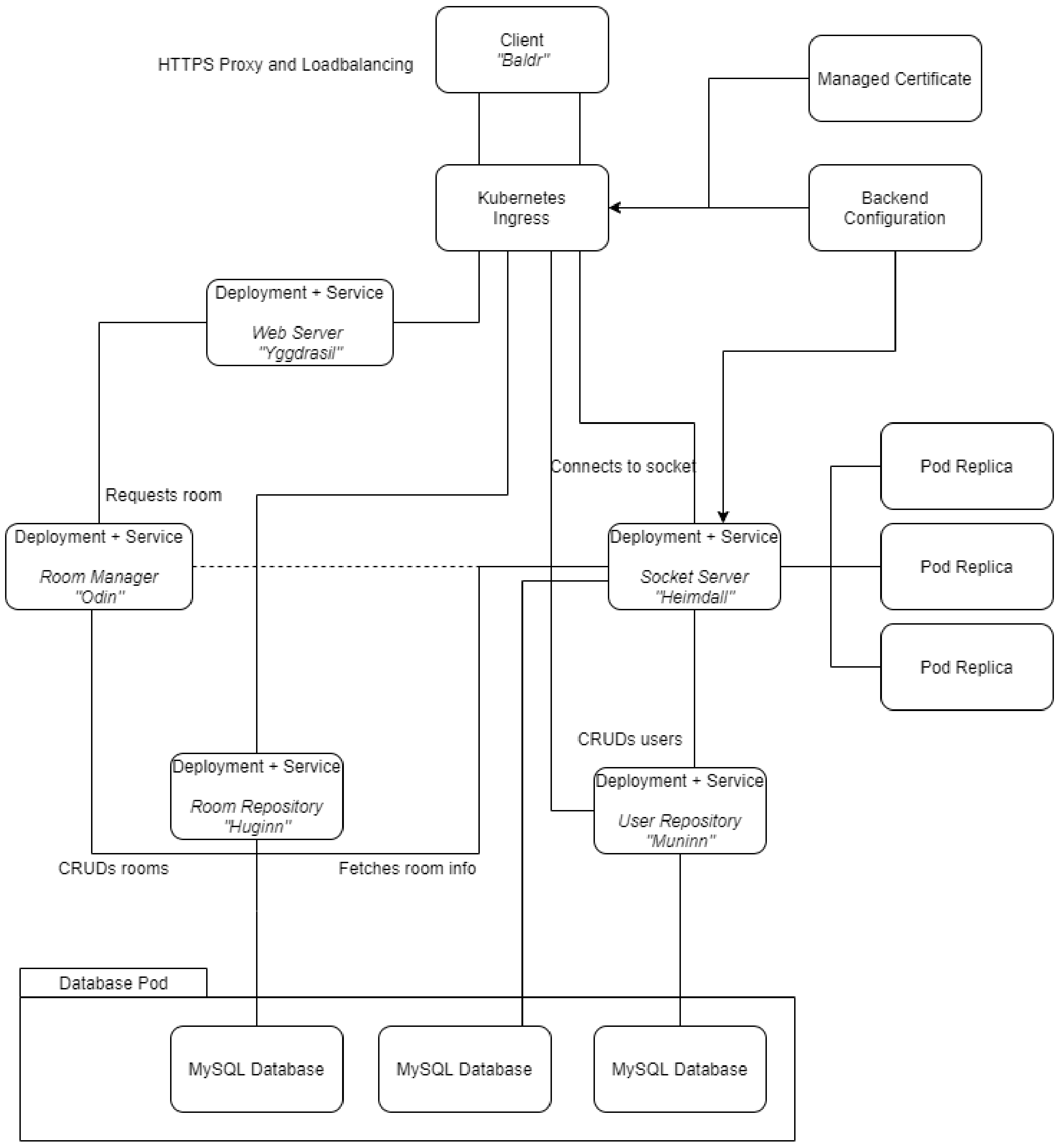


Figure 10.1: A diagram showing the Design architecture described with the deployments and services in the Kubernetes cluster deployment

All configuration files used can be found under this link [25]

## 10.1 Ingress

The system back end is behind an Ingress load-balancer that handles the external access to the system. The Ingress contains a number of paths that lead to specified services in the Kubernetes namespace. When these paths are connected to, either via an API or through a web browser, the Ingress leads to the service defined under said path.

To expose the Ingress an external IP is created and bound to the ingress in order to gain access remotely. This is done using Google Cloud. Through Google Cloud, a static external IP is created and attached to the Ingress. Through that IP it is now possible to access the services connected to the Ingress Load-Balancer. In addition to external access, a certificate is added to the Ingress in order to gain TLS on the connections. This is done through a ManagedCertificate configuration applied to the Kubernetes cluster. Googles Ingress Controller automatically talks with Googles Certificate Providers, so when applied, the certificate provision is started automatically. The only requirement is to connect a domain name to the external IP. This was done using Cloudflare.

After all these configurations, CoffeeBreak is publicly available on a self provided domain, with HTTPS connection.

## 10.2 Shared Components in the Twin Designs

The shared system components consists of ”Yggdrasil”, the web server that serves the website client. The room repository, ”Huginn”, that handles room CRUD operations with the client and room database creation and fetching. The user repository, ”Muginn”, that handles the user CRUD Operations with the client and user database creation and fetching. Lastly, the room manager, ”Odin” which handles room creation.

The above mentioned services all are configured to be a Deployment that has a single Pod, and a Service on top. The Service exposes the pod IP and communication in order for the Ingress to publicly expose them to the internet. The Deployment configuration contains the name of the application, what port the container should open, the image of the container and a number of environmental variables. These environmental variables contain the different information required to know, in order to connect to the other services in the Kubernetes cluster.

The Service configuration contains the label of which Deployment it should be connected to, target port of said deployment, and the port it should forward the target port to.

Setting it up like this makes it easy to give access to each Service externally through the Ingress or creating more replicas if required through the Deployment configuration. In this case, since most of the traffic goes through the Socket Server “Heimdall”, the Deployments does not have any replicas, other than ”Yggdrasil”.

## 10.3 The Socket Server

The socket server “Heimdall” is the heart of the system, serving both the room and the signalling socket server. The room socket server handles user positions, chat messaging, and more in each room, while the signalling server handles the WebRTC signalling between all users. This causes it to potentially receive a large amount of traffic depending on the amount of users. To counteract this a deployment solution had to be found in order to prevent a bottleneck in the system. The solution found is to have multiple replicas of the Socket Server service. This would split the incoming traffic between the replicas. However with socket servers, if two users connect to the same room, but different replicas, they would receive any messages sent to each other. A system needs to be set in place in order to update all socket server replicas.

This is done with Redis. Redis handles the messages sent to the separate replicas of the socket server and publishes them to all the others replicas subscribed to the redis implementation.

The Socket Server "Heimdall" has three replicas by default. A redis master-slave system is created in order to keep track of the different socket server replicas and update them in order for users connected to different replicas, still be able to communicate with each other. Below is an illustration of this configuration. In order for this configuration to work

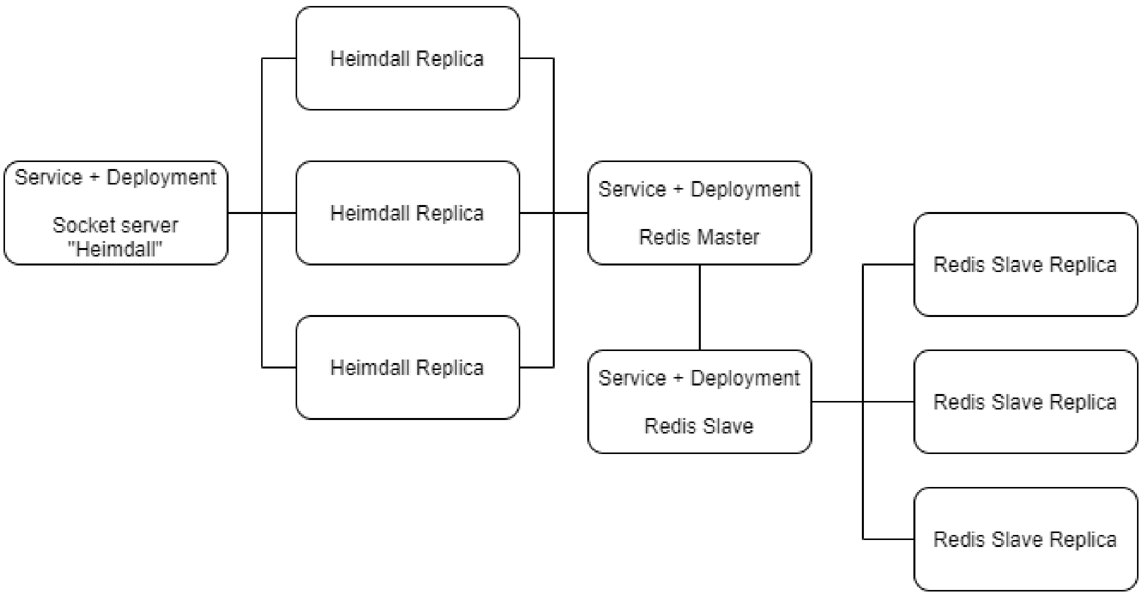


Figure 10.2: A diagram showing the relationship between the socket server replicas and the redis implementation

as intended it is important to implement a sticky session between the user and the connected Socket Server replica. This sticky session ensures that the user is connected to the same replica throughout the socket connection. A BackendConfig is applied to the cluster and inserted as an annotation in the Ingress configuration and the Heimdall service configuration file.

## 10.4 Pod Auto scaling

The web server, Yggdrasil, and the Socket Server, Heimdall, also has a horizontal pod autoscaler implemented. This is to automatically increase scalability in case of the initial three replicas is not enough to cope with the current load. A Metrics Server is applied to the cluster in order to monitor resource usage across pods on the cluster. Using this information, the autoscalers connected to the deployments create or delete replicas in line with workload.

## 10.5 The Database

The database is a simple pod implementation of a stock MySQL database. The IP of the pod is entered as an environmental variable in order for the services to connect.

## 10.6 Conclusion

Deployment of the system to a Kubernetes Cluster makes it simple to create scalable modifications to an existing architecture that seems rigid in nature. Using the different aspects of Kubernetes in general and Google Clouds' Kubernetes Engine, it is possible to establish a fully fledged system that is scalable in order to reach the goals established at the start of the project.

# 11 User Manual

This chapter outlines how to use the developed system, both in terms of using it as it is intended as a client, as well as how to deploy the system on ones own server for testing using Docker.

## 11.1 Client Manual

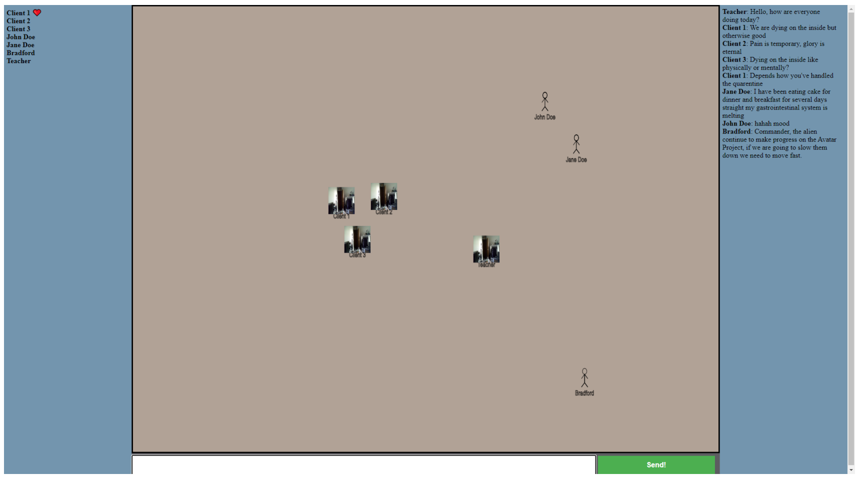
Usage of the system is simple for the client. There are two endpoints of direct interest for the client, both on the main controller. To put it simply, there are only two web addresses that the client needs to worry about, everything else is handled behind-the-scenes. Given, for instance, the website domain "https://federicoshytte.dk/"[1], the only addresses the client needs to worry about are:

- https://federicoshytte.dk/room
- https://federicoshytte.dk/room/:roomId

The first one requests a new room to be created. The second one requests a specific room denoted by the :roomId part of the URL, usually one the client has been linked to by another person. Requesting a new room automatically redirects the client to the new room, and the room URL can then be copied from the web browsers address bar to send to others. A specific room URL might look something like this:

https://federicoshytte.dk/room/64f869a5-0f2c-40fa-a31c-bc3019cf7174

Once a room has been joined, the client is greeted by a rough user interface. An example of this is visible in figure 11.1



**Teacher**: Hello, how are everyone doing today?
**Client 1**: We are dying on the inside but otherwise good
**Client 2**: Pain is temporary, glory is eternal
**Client 3**: Dying on the inside like physically or mentally?
**Client 1**: Depends how you've handled the quarentine
**Jane Doe**: I have been eating cake for dinner and breakfast for several days straight my gastrointestinal system is melting
**John Doe**: hahah mood
**Bradford**: Commander, the alien continue to make progress on the Avatar Project, if we are going to slow them down we need to move fast.

Figure 11.1: An image showing a room, with seven clients connected and spread throughout in a number of smaller groups. The picture is from the perspective Client 1, the leftmost client.

As can be seen in figure 11.1, the user interface is split in three major parts. A list of connected users to the left, the virtual room rendered in the middle with a input box for

[1]The service is expected to be deployed on this domain up to and during the project exam, and may be used freely. If it is for some reason down, please contact the team to request a reboot.

chat message in the button, and finally a list of chat messages to the right. The client can move around in the room by clicking the virtual room at their desired position. Once moved, they will connect and disconnect video and voice connections to other users as they have moved in and out of an arbitrary range. These connections are shown in the figure as nearby identical webcam streams, as the picture was taken with all clients on the same physical machine.

## 11.2 Deployment Manual

In the GitHub repository, a docker-compose file is included. Starting this will setup a local testing environment to experiment with the system. This will be the case for both of the implemented solutions described in the report.

Docker compose deployment containing all repos requires as submodules for the two solutions are linked bellow

The P2P solution

https://github.com/CoffeeBreak-Architecture/Docker-Deployment-P2P

The MRS solution

https://github.com/CoffeeBreak-Architecture/Docker-Deployment-MRS

To clone the repository with all submodules use the following command in Git

```
git clone --recurse-submodules -j8 "Solution git link"
```

Afterwards just docker-compose up to start the deployment.

If a ”no build” deployment is wanted instead, go to the link below. This repository contains docker-compose files building the system out of Docker Hub images, eliminating the need to build the containers yourself.

https://github.com/CoffeeBreak-Architecture/Docker-Compose-Files-Build-Free

# 12 Results

## 12.1 The Primary Goal - The Product

The end result of the development resulted in a fully deployed system that enables the user to communicate with other users connecting to the same room. The user is able to connect to rooms with other users around the world. The connected users are able to move around a virtual space, connecting to other users near their avatar and the volume of other users voices is automatically adjusted to simulate a real life environment.

The system is deployed on a Kubernetes cluster handling the lifetime of the different services that encompasses the entire CoffeeBreak system. Multiple of these services are automatically scaled to fit whatever need the system might have depending on user traffic levels. The cluster is hosted on Google Kubernetes Engine, which handles the overall cluster operations, in addition to the external IP and TLS configuration. This allows the system to operate over HTTPS giving the user a secure access to the system.

## 12.2 The Secondary Goal - Performance

In addition to the overall goal, performance was also tested between the two different solutions. The results indicate that the P2P solution is more heavily reliant on client side performance due to the client browser handling all connections to the other users, while the MRS solution splits the load more evenly between the server and client. These tests were originally done looking at the CPU load of the users browser when testing the P2P solution and looking both on the clients CPU load and node CPU load when testing the MRS solution.

As described during the design session, the results should end with the P2P solution having a exponential growth in CPU consumption due to the exponential growth in user connections. Also, the MRS solution should result in a linear CPU usage growth due to the linear growth user connections. Below is two graphs illustrating this behavior after testing both designs on a local machine.

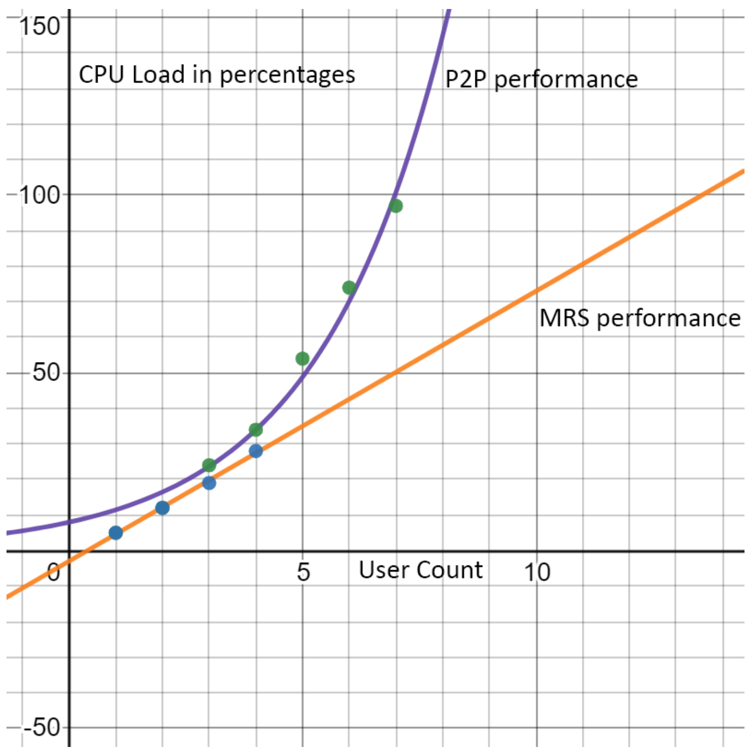


Figure 12.1: Illustrates the the effect of CPU usage based on user count for both of the implemented solutions

As illustrated above, the CPU usage increases exponentially with the P2P design, while it increases linear with the MRS design. This confirms the initial design assumptions.

Lastly, overall performance testing was also run on the overall cluster to stress test the deployment in high traffic situations. Through the use of Artillery, a number of stress tests was created to give insight into the overall system performance during high stress situations. Areas a such as Database read and write was tested through a influx of requests, rising in volume over a period of time. All tests resulted in a 99.99% success. This shows that the system is capable of massive influx of connections.

A rudimentary evaluation of technical complexity determines that the technical difficulty between implementing the twin designs is apparent by the number of system components alone. The Vili design has a total of seven components, one more than the Odin design, with that seventh component being of significant difficulty to implement due to the complexity of the involved technologies. Furthermore, the physical deployment of the system requires further miscellaneous changes to accommodate the media relay server. The team is confident in concluding that, while using a media relay server might have performance benefits, it does come with significant added development complexity, and is not worth doing unless necessary.

In addition to test the socket server, a stress test was created to simulate real users connection to the socket. These simulated users consists of three user archetypes. A ”Lurker”, which connects to the room and doesn’t do anything, a ”Quiet chatter”, which sends a message once, and a ”Frequent Chatter”, which sends a message every second for twenty seconds. Running this test shows that many users are capable of joining the same room and the socket server can handle many connections simultaneously, without any connection problems and the horizontal pod auto-scalers spin new pods up when

CPU usage reaches a certain milestone.

The full summary reports can be found in the appendix

# 13 Evaluation and Reflections

This chapter discusses the the project and how successful and satisfying it has been. Most of this chapter is dedicated to discussing the faults and failures of the development process, where things went wrong, what went wrong, what could have been done alternatively, or what could have been done to avoid it. The chapter is split into a section for each step of development corresponding to all chapters between this and Requirements, as well as a section on the impact of the Covid-19 quarentine.

## 13.1 Requirements

Developing the initial set of requirements were without too many issues. After the various proposed projects were presented for us, a number of teams chose the CoffeeBreak project, and none of us were really sure if all the teams were supposed to work together, or if we should make individual projects. This did somewhat slow down the initial requirement elicitation, though after debating the requirements with the other groups we did end up with what we felt, and still feel, is a fairly robust set of system requirements. While we are satisfied with the final result, the requirement elicitation did occupy our minds for longer than they needed to, at times where other initial project aspects might have needed attention.

As a result of this, we had little mental room to figure out the core academic purpose of the project early, preventing us from generating a solid footing and performing a good and proper literature review. This lack of a solid project core has affected the project throughout, mostly as the core we finally did find in the middle of the project, the measuring of performance of different implementation variations, was without much academic backing and without later potential consequences truly thought out. With this lesson learned the hard way, we hope to keep a good academic core in mind for future academic projects.

## 13.2 Analysis

Establishing a finalized requirement list was straight forward due to the already thoroughly established overall requirements. Creating use cases to further analyse the different system interactions gave us a deeper insight as to what was required. Frequent feedback reports from our supervisors during this phase of the project development, further gave us information in regards to how specific the requirements established was and if they needed adjusting or narrowed down. This feedback and close relationship between us and our supervisors gave us a solid requirement specification of which to work from in the design phase and beyond.

The analysis itself went alright, the resulting finalized requirement specification, as well as the throughout exploration of the potential system through detailed use case specifications did help us find the initial design which to some extend survived towards the end of the project. We are, however, uncertain if the requirements are specific and clear enough that they could be given to another team to implement correctly according to our vision. Making greater use of behaviour-driven development principles, in particular object-oriented analysis, might result in a clearer, more concise requirement specification, with less implementation-specific information scattered throughout, as well as more consistent levels of detail where needed.

## 13.3 Design

The design suffered heavy simplification through development. The impracticality of some of the originally proposed design became particularly clear. For instance, the originally proposed design had individual Kubernetes pods per rooms or per users, but during implementation it became clear that this design would become highly over complicated, and a much simpler design would work much better with Kubernetes. Looking back, we should have further investigated the different technologies we wanted to incorporate, and had a better understanding of their capabilities before we decided how we wanted to use them. This could potentially have been avoided by trying to build a simple prototype of the originally proposed design. At least then we would have been better prepared for the issues with the design, and might have been better at choosing more compatible technologies. While we are happy with the final design in the context of the product, the academic core of the project relied heavily upon it, and as a result suffered tremendously.

## 13.4 Implementation

The analysis and design phases gave a good overall overview of the systems architecture initially, and the initial few days of implementation started out looking at the feasibility of the design, by working with the technologies hands on. However when working with the different technologies hands-on, it gave a more thorough understanding of their strengths and weaknesses, in addition to what was possible within the design already established. This caused a lot of iterative changes to the design in order to implement certain features and resulted in a much different design of the system than initially thought out. The overall functionality was still implemented, and many of the established requirements were fulfilled, however many also had to be removed to fit the new design that was evolved during the implementation.

The implementation phase was an intensive period of making a functioning system. Due to the above problems, and no time to go back to the drawing board and recreating the design, ”on the fly” solutions were the answer to most if not all problems that occurred during the process. Overall we think that the implementation phase, given the circumstances went well. If more time were available, a second iteration of analysis and design would be fitting, in order to adjust with the experience in mind.

## 13.5 Verification

There is little to complain about with the process of verifying the system requirements. Some requirements turned out to be difficult to validate properly and automatically due to the complexities of doing so, or the lack of testable related code. In general the code was fairly, but not perfectly testable. While test-driven development principles were discussed early in development, we decided against it due to the development overhead commonly entailed, though it would likely have improved the quality and quantity of verification significantly. Most testable units as well as interactions between components have been tested, though not as thoroughly as they could be, as there is for the most time only a single test for each unit, with an occasional negative test to further verify. Optimally, each unit should have one positive test, one negative test, and one exception test in our opinion.

CI using GitHub Actions worked without significant hassle. There was some difficulty getting true integration testing working, and as a result only a proof-of-concept was done for a single component. Aside from this, GitHub Actions has proven a strong contender amongst other CI services.

## 13.6 Deployment

The initial deployment detail was early in development and went through many different iterations and complications, before finally finding a solution that solved all of our problems. These early problems caused the time it took to get a fully working deployment to increase massively and cause certain goals to be deemed unreachable due to time constraints. One of these was a fully deployed Kubernetes cluster containing the MRS solution, Vili. Due to the aforementioned time constraints, a proper Kubernetes deployment of the Vili solution never came to light.

The Vili solution, if implemented, would consist of a Media Relay Server that would represent each room. All media would have to go through the Media Relay Server instead of going straight to the users. The Kubernetes Cluster would essentially function the same, however room creation would rely heavily on a Kubernetes API implementation in the room manager. When a room is created a new MRS Kubernetes deployment would be created specifically for the room. This deployment would then be connected to from the client.

Unfortunately this never came to life, and the furthest extent of a proper deployment is a docker-compose of all the services working, with several aspects of the aforementioned proposed solution not implemented yet. However this solution to a deployment works flawlessly on a local machine, so showcasing the would be system is a possibility

## 13.7 Results

The results of the project is lackluster to say the least due to reasons mentioned in the design section of this chapter. With the hardship of finding a proper angle, it was tough to establish a proper aspect to assess the project. In the end, the result ended with being the established product. This might be acceptable in the context of primary goal established for this project, however when reflecting, it pains us that we couldn't establish a more academic angle and goal to research and answer.

In search of a more measurable goal and potential result, we established a secondary goal that consists of performance analysis of our two implemented solutions. However due to this being a secondary goal, the main focus was on creating a finished product that fulfilled our main goal. This caused the results gained from this to be thin and not measured in certain aspects either due to not having implemented certain features that makes measurement possible, which again is lacking due to time constraints. However the results gained, albeit small, align with the different WebRTC configurations in the Design phase. With more thorough testing on, it could have given us more solid data to display.

## 13.8 Impact of the Covid-19 Quarentine

As may be expected from any project developed during the Covid-19 pandemic between 2020 and 2021, the project has suffered somewhat due to the mental health impact of quarantine. The team suffered from a lack of motivation at times, and aspects of the project suffered, particularly where the team lacks significant skill such as in analysis. We believe that the project foundation would have been significantly stronger if, what is colloquially known as ”Quarantine Depression” had not had such a mental toll, but such is life sometimes. The worst of this was several times late in development where the team suffered severe burnouts, possibly due to the lack of daily structure, as well as an unhealthy work-life balance. This ultimately resulted in some lack of intended features and polish, but we feel that we got decently enough through it after all.

## 13.9 Conclusion of Evaluation

To say any project goes perfectly to plan would probably be a lie, and this project is no different. The greatest challenges faced with this project were difficulties getting truly going with a solid core. This affected large parts of the system, as a core had to be found halfway through, resulting in one that was somewhat poorly thought out. To further worsen this, the original design intended to work with this core turned out impractical, and was simplified to be much more practical, at the cost of detailed academic results. While the Covid-19 pandemic did take a heavy toll at times, and lessened the quality of the project, more severely in some aspects than others, we feel that we made it through well enough regardless.

Aside from that, each part of development each faced various issues, though we do not have particularity severe regrets about most other than analysis, which could have been significantly better using more robust principles.

# 14 Conclusion

The first system, ”Odin”, resulted in a Peer to peer communication solution hosted on a Kubernetes Cluster. The Kubernetes cluster is hosted on Google Cloud, resulting in a simple deployment. The final product is able to host multiple users in self created rooms. When users move around and near each other, audio and video connections is automatically established. The second system is almost identical, however a server handling the audio and video connections is used instead of a direct connection. This solution didn’t get a Kubernetes deployment due to time constraints.

The overall reception of the system is positive. As can be expected from a software product in development, many bugs were found due to external and internal testing, but the overall system and the way it works fundamentally was an interesting aspect to the users testing the system. The fact that users could move around and connect to specific groups based on those nearby in a dynamic fashion, and not just joining a room, was received positively.

To conclude on the overall goal, the group achieved to create the system CoffeeBreak, an application that enables communication in a virtual space. This was further determined by series of tests to ensure all requirements were implemented and completed.

A secondary goal was also established to determine the performance differences of the media transfer between the two established systems and also the overall performance of the created Kubernetes deployment of the ”Odin” solution. Through testing of both solutions, it was determined that the Vili solution would perform better on a higher scale, since the CPU usage increased in a linear fashion, in contrast to Odin’s exponential growth. The overall performance of the Odin implementation was tested through multiple stress tests that targeted the core parts of the system, and they resulted in that he systems overall performance under heavy stress was satisfactory.

At the end, the group achieved creating multiple proposals to a virtual communication platform, comparing them in order to find the performance differences. Through this it was concluded that a MRS design gives a lower growth of CPU performance compared to a P2P solution. Additionally, establishing a solid deployment architecture that is backed by stress testing of the core parts of the system. In conclusion, the group reached its goals established at the beginning of the project and have achieved their learning objectives.

# Bibliography

[1] Discord. *Discord Guild Resource*. URL: https://discord.com/developers/docs/resources/guild.

[2] Jozsef Vass. *How Discord Handles Two and Half Million Concurrent Voice Users using WebRTC*. 2018. URL: https://blog.discord.com/how-discord-handles-two-and-half-million-concurrent-voice-users-using-webrtc-ce01c3187429.

[3] Discord. *Discord API Reference*. URL: https://discord.com/developers/docs/reference.

[4] Stanislav Vishnevskiy. *How Discord Scaled Elixir to 5,000,000 Concurrent Users*. 2017. URL: https://blog.discord.com/scaling-elixir-f9b8e1e7c29b.

[5] MDN. *WebRTC API*. 2021. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API.

[6] MDN. *Introduction to WebRTC protocols*. 2021. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Protocols.

[7] Manish Jain. *Zoom Video Conferencing App: Features, Tech Stack, Monetization, Cost*. 2021. URL: https://www.konstantinfo.com/blog/zoom-video-conferencing-app/.

[8] Vonage. *Vonage API*. URL: https://www.vonage.com/communications-apis/video/.

[9] Mumble. *Mumble Protocol Documentation*. 2012. URL: https://mumble-protocol.readthedocs.io/en/latest/index.html.

[10] Mumble. *Mumble*. URL: https://www.mumble.info/.

[11] Mumble. *Mumble Voice Data*. 2012. URL: https://mumble-protocol.readthedocs.io/en/latest/voice_data.html.

[12] Václav Rajlich. *Software Engineering: The Current Practice*. Chapman and Hall/CRC, 2011. ISBN: 978-1439841228.

[13] GitHub. *Project management, made simple*. URL: https://github.com/features/project-management/.

[14] Kubernetes. *Kubernetes*. URL: https://kubernetes.io/.

[15] Socket.io. *Socket.io*. URL: https://socket.io/.

[16] GitHub. *GitHub Actions*. 2021. URL: https://docs.github.com/en/actions.

[17] burak. *WebRTC Servers and Multiparty Communication in WebRTC*. 2021. URL: https://antmedia.io/webrtc-servers/.

[18] OpenJS. *Express*. 2017. URL: https://expressjs.com/.

[19] OpenJS. *JSON*. 2017. URL: http://expressjs.com/en/resources/middleware/cors.html.

[20] OpenJS. *4.x API*. 2017. URL: http://expressjs.com/en/api.html.

[21] OpenJS. *Cookie Parser*. 2017. URL: http://expressjs.com/en/resources/middleware/cookie-parser.html.

[22] Axios. *Axios*. URL: https://github.com/axios/axios.

[23] MySQL2. *MySQL2*. URL: https://www.npmjs.com/package/mysql2.

[24] Marcus L. Jensen Frederik B. Roth. *CoffeeBreak Source Code*. 2021. URL: https://github.com/CoffeeBreak-Architecture/.

[25] Marcus L. Jensen Frederik B. Roth. *CoffeeBreak Source Code*. 2021. URL: https://github.com/CoffeeBreak-Architecture/Configuration-Files/tree/main/Kubernetes%5C%20Config%5C%20Files.

# A Larger Gantt Diagram

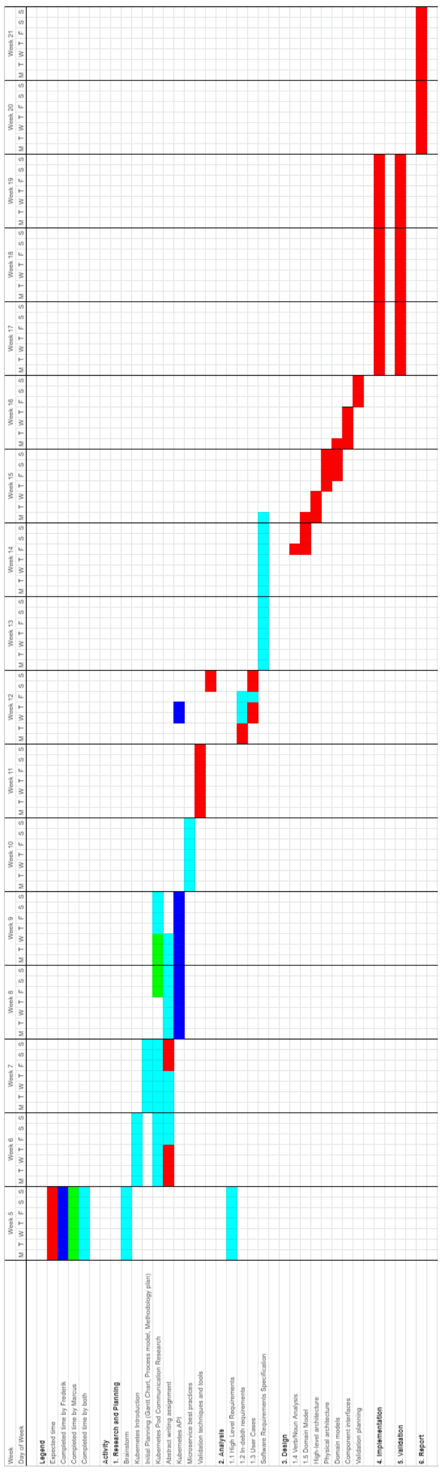


Figure A.1: Larger version of the figure shown in the Methodology and Process chapter. The picture is still not very clear, but the image is too large and is not of particular importance.

# B Full size Odin deployment design

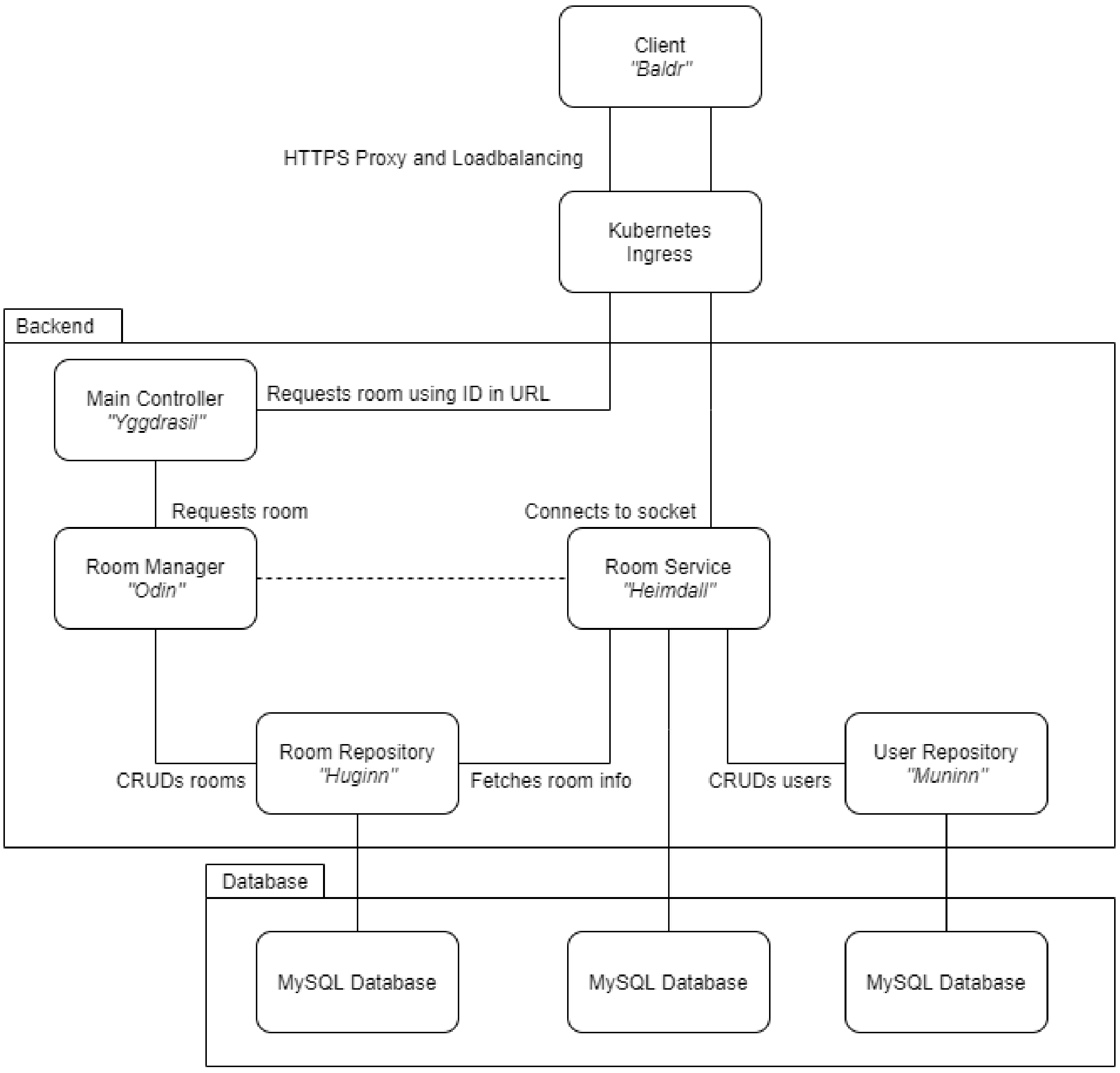


Figure B.1: The full sized image of the Odin deployment displayed in the Design chapter.

# C Full size Vili deployment design

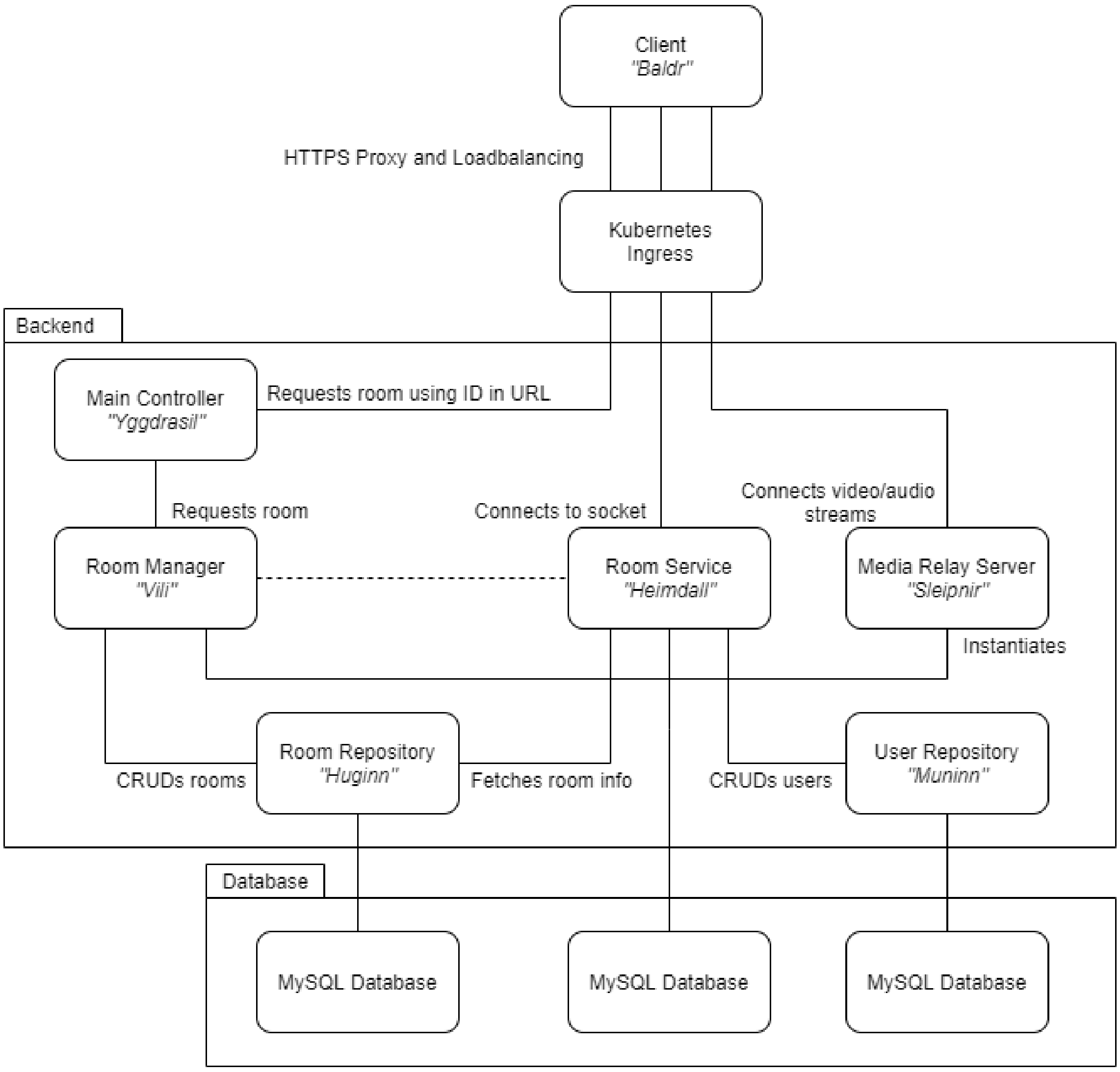


Figure C.1: The full sized image of the Vili deployment displayed in the Design chapter.

# D Initially proposed design, part 1/2

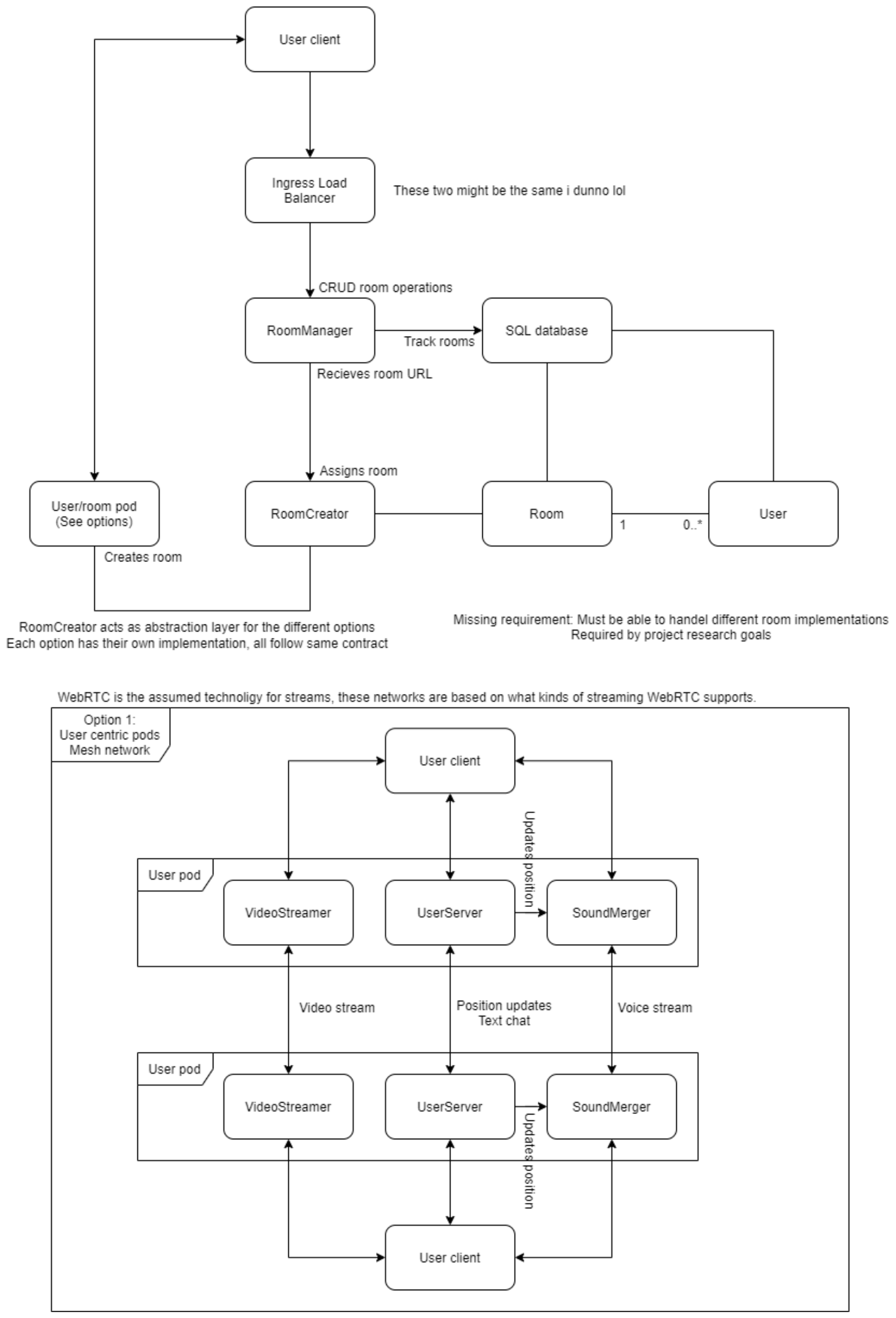

User client
Ingress Load Balancer
These two might be the same i dunno lol
CRUD room operations
RoomManager
Track rooms
SQL database
Recieves room URL
Assigns room
User/room pod (See options)
RoomCreator
Room
1
0..*
User
Creates room
RoomCreator acts as abstraction layer for the different options
Each option has their own implementation, all follow same contract
Missing requirement: Must be able to handel different room implementations
Required by project research goals
WebRTC is the assumed technoligy for streams, these networks are based on what kinds of streaming WebRTC supports.
Option 1: User centric pods Mesh network
User client
Updates position
User pod
VideoStreamer
UserServer
SoundMerger
Video stream
Position updates
Text chat
Voice stream
User pod
VideoStreamer
UserServer
SoundMerger
Updates position
User client

Figure D.1: The initially proposed design of the system post-analysis. Notice the much more complex room designs in the panel marked as Option 1, as well as option 2 and 3 displayed in E.1. This is the general area of the design that was heavily simplified. The upper part of the figure displays the initial design of shared components, many of which survived the changes, such as but not limited to the RoomManager which became Yggdrasil, the RoomCreator which became Odin/Vili, and the user/room pod which became Heimdall.

# E Initially proposed design, part 2/2

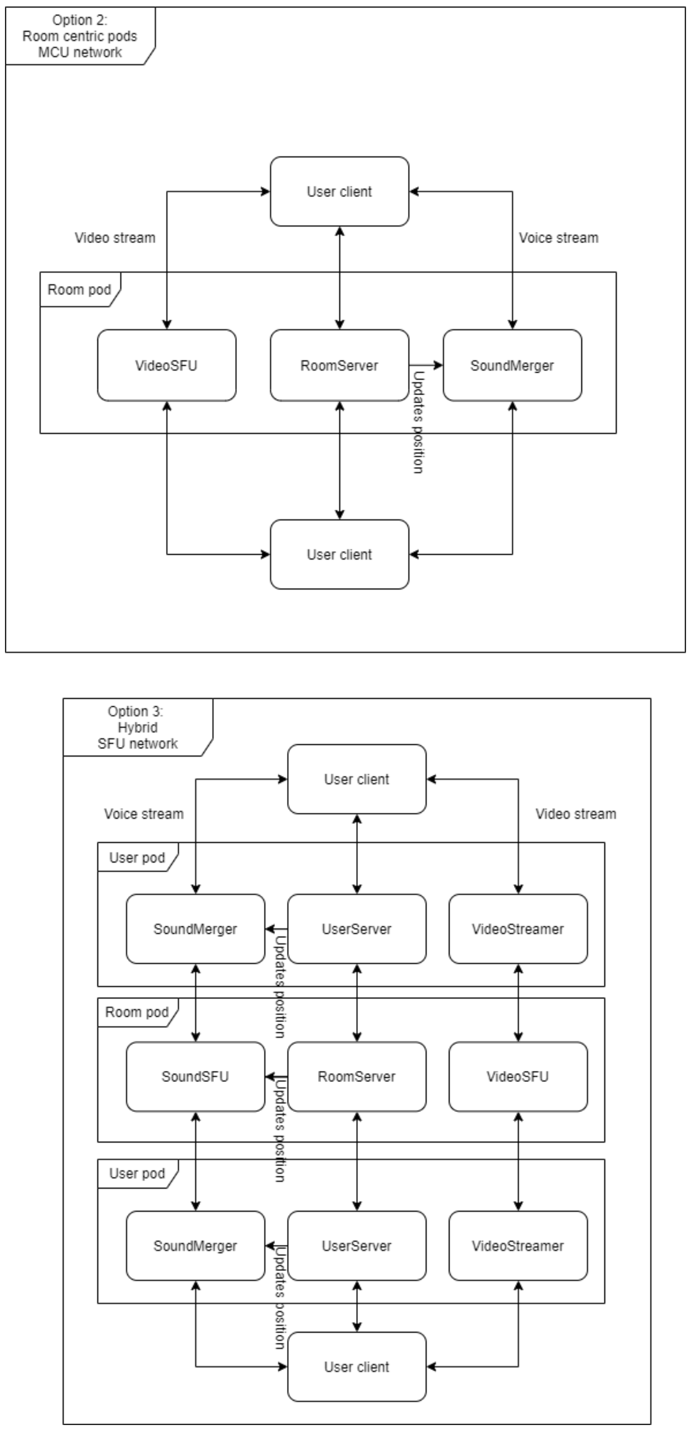
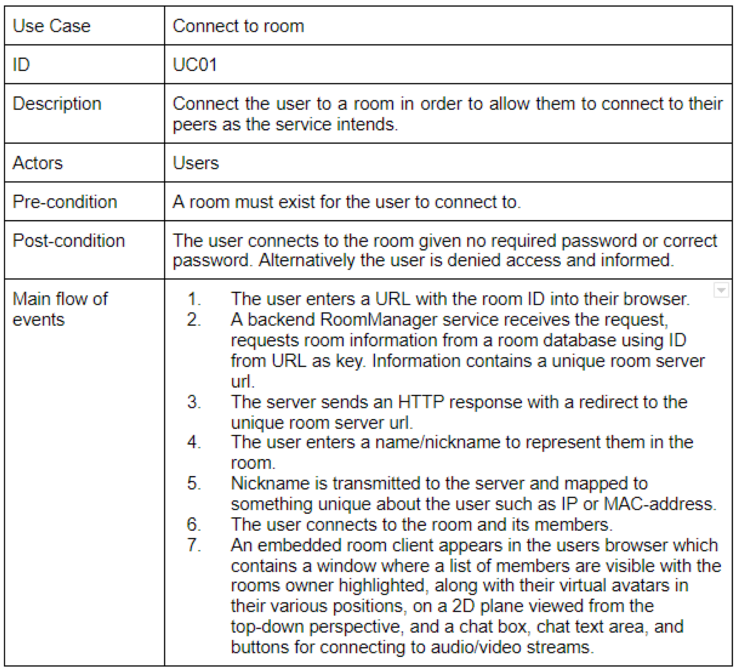


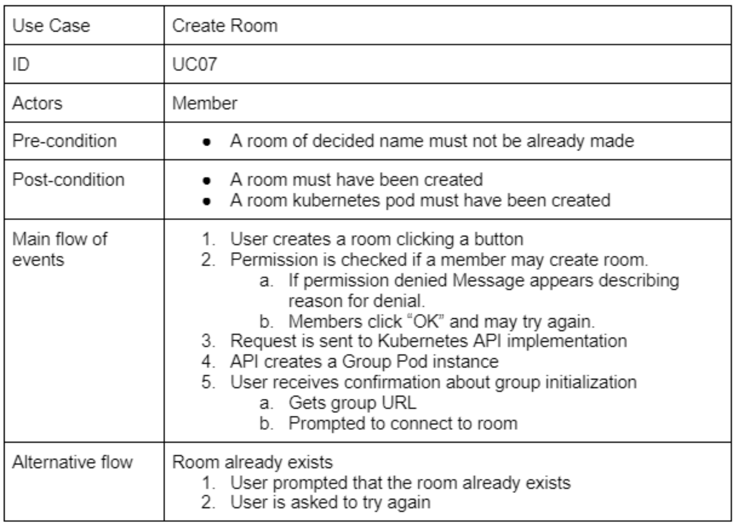
Figure E.1: The second part of the initially proposed design, displaying two other room design variations. Option 2 displayed here is closest to the final design, with RoomServer being roughly equivalent to Heimdall, and SoundMerger and VideoSFU being rougly equivalent to Sleipnir.

# F Use Cases created in the analysis phase of the project



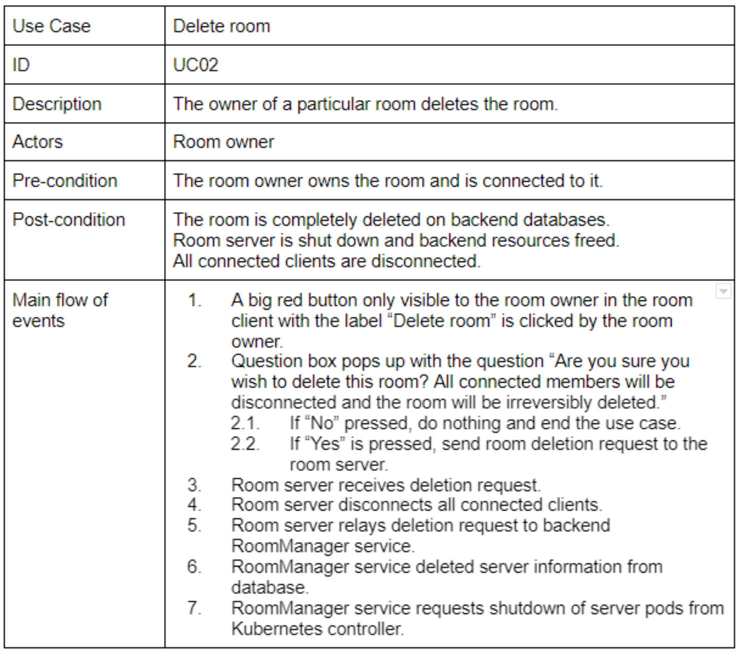
| | |
|---|---|
| Use Case | Connect to room |
| ID | UC01 |
| Description | Connect the user to a room in order to allow them to connect to their peers as the service intends. |
| Actors | Users |
| Pre-condition | A room must exist for the user to connect to. |
| Post-condition | The user connects to the room given no required password or correct password. Alternatively the user is denied access and informed. |
| Main flow of events | 1. The user enters a URL with the room ID into their browser. 2. A backend RoomManager service receives the request, requests room information from a room database using ID from URL as key. Information contains a unique room server url. 3. The server sends an HTTP response with a redirect to the unique room server url. 4. The user enters a name/nickname to represent them in the room. 5. Nickname is transmitted to the server and mapped to something unique about the user such as IP or MAC-address. 6. The user connects to the room and its members. 7. An embedded room client appears in the users browser which contains a window where a list of members are visible with the rooms owner highlighted, along with their virtual avatars in their various positions, on a 2D plane viewed from the top-down perspective, and a chat box, chat text area, and buttons for connecting to audio/video streams. |

Figure F.1: This Use Case illustrates the flow of events of when a user connects to a room.



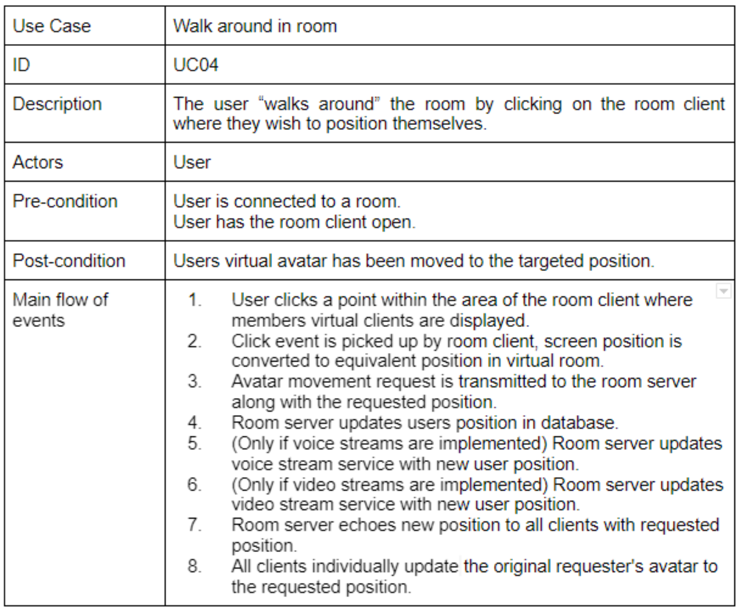
| | |
|---|---|
| Use Case | Create Room |
| ID | UC07 |
| Actors | Member |
| Pre-condition | • A room of decided name must not be already made |
| Post-condition | • A room must have been created • A room kubernetes pod must have been created |
| Main flow of events | 1. User creates a room clicking a button 2. Permission is checked if a member may create room. a. If permission denied Message appears describing reason for denial. b. Members click "OK" and may try again. 3. Request is sent to Kubernetes API implementation 4. API creates a Group Pod instance 5. User receives confirmation about group initialization a. Gets group URL b. Prompted to connect to room |
| Alternative flow | Room already exists 1. User prompted that the room already exists 2. User is asked to try again |

Figure F.2: This Use Case illustrates the flow of events of when a user connects creates a room.



| | |
|---|---|
| Use Case | Delete room |
| ID | UC02 |
| Description | The owner of a particular room deletes the room. |
| Actors | Room owner |
| Pre-condition | The room owner owns the room and is connected to it. |
| Post-condition | The room is completely deleted on backend databases.<br>Room server is shut down and backend resources freed.<br>All connected clients are disconnected. |
| Main flow of events | 1. A big red button only visible to the room owner in the room client with the label "Delete room" is clicked by the room owner.<br>2. Question box pops up with the question "Are you sure you wish to delete this room? All connected members will be disconnected and the room will be irreversibly deleted."<br>2.1. If "No" pressed, do nothing and end the use case.<br>2.2. If "Yes" is pressed, send room deletion request to the room server.<br>3. Room server receives deletion request.<br>4. Room server disconnects all connected clients.<br>5. Room server relays deletion request to backend RoomManager service.<br>6. RoomManager service deleted server information from database.<br>7. RoomManager service requests shutdown of server pods from Kubernetes controller. |

Figure F.3: This Use Case illustrates the flow of events of when a user deletes a room.



| | |
|---|---|
| Use Case | Walk around in room |
| ID | UC04 |
| Description | The user "walks around" the room by clicking on the room client where they wish to position themselves. |
| Actors | User |
| Pre-condition | User is connected to a room.<br>User has the room client open. |
| Post-condition | Users virtual avatar has been moved to the targeted position. |
| Main flow of events | 1. User clicks a point within the area of the room client where members virtual clients are displayed.<br>2. Click event is picked up by room client, screen position is converted to equivalent position in virtual room.<br>3. Avatar movement request is transmitted to the room server along with the requested position.<br>4. Room server updates users position in database.<br>5. (Only if voice streams are implemented) Room server updates voice stream service with new user position.<br>6. (Only if video streams are implemented) Room server updates video stream service with new user position.<br>7. Room server echoes new position to all clients with requested position.<br>8. All clients individually update the original requester's avatar to the requested position. |

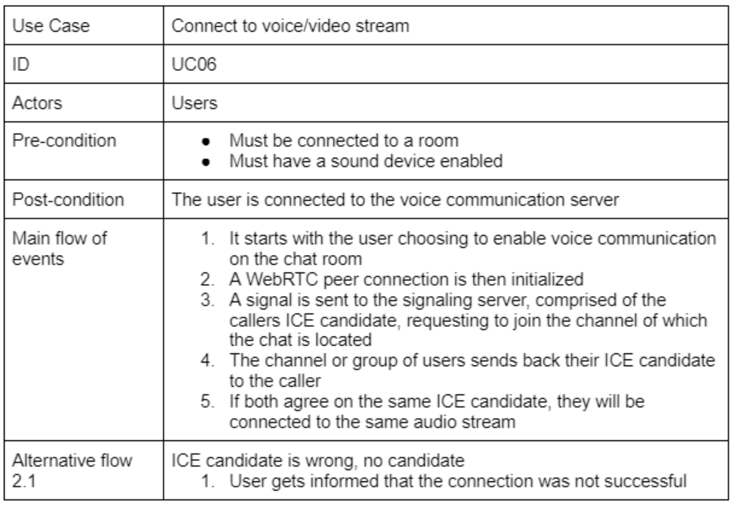
Figure F.4: This Use Case illustrates the flow of events of when a user moves around the room canvas.

| | |
|---|---|
| Use Case | Register as member |
| ID | UC05 |
| Description | Allows a user to register as a member and allow them to authenticate. Only important if the system is expected to need authentication for user permissions further down the line. Otherwise DO NOT IMPLEMENT. |
| Actors | User |
| Pre-condition | User is not already logged in. |
| Post-condition | The following if the request information was validated:<br>Users have been registered to the system with their account information stored in a database.<br>User is logged in to their new account. |
| Main flow of events | 1. User enters a "Register account" web page and is presented with a register form with fields for username, password, repeat password, and email.<br>2. User enters information and hits "submit".<br>2.1. If any field is empty, display an error message ("<Field> cannot be empty") where <Field> is the field in question.<br>2.2. If passwords don't match, display error message ("Passwords doesn't match.")<br>2.3. If any information doesn't adhere to certain restrictions such as minimum username or password length, display an error message explaining as such.<br>2.4. After any of these, the user is free to try again.<br>3. Send registration request to an authentication service as an HTTPS request.<br>4. Backend service receives request, validates information in accordance to same rules described in step 2.1 through 2.3<br>4.1. If validation fails, respond with HTTP error code 400 Bad Request. End use case without registration.<br>5. Backend adds new account to account database with the received and validated information except password is hashed.<br>6. Backend creates a user token, adds a token to the database of currently connected users.<br>7. Backend returns a response which redirects the user to the main page with their token as a cookie. |

Figure F.5: This Use Case illustrates the flow of events of when a user registers as a member. This is functionality never made it to the final product.

| | |
|---|---|
| Use Case | Send message in room |
| ID | UC03 |
| Description | User connected to a room sends a message to all other users connected to the room. |
| Actors | User |
| Pre-condition | User is connected to a room.<br>User has the room client open. |
| Post-condition | Message is received by all members in the room, including sending the user as an echo. |
| Main flow of events | 1. User enters a message into a wide text input field spanning almost the bottom of the room client.<br>2. User clicks the "Send" button to the right of the input field or hits the "Enter" keyboard key.<br>3. Message is sent to the room server as plaintext via websockets along with the sender name/nickname prepended to the message.<br>4. Room server receives the message and echoes it to all connected clients.<br>5. All clients individually receive the message.<br>6. Clients append messages to a chatbox where all chat messages are displayed. |

Figure F.6: This Use Case illustrates the flow of events of when a user sends a message to the room it is connected to.



| | |
|---|---|
| Use Case | Connect to voice/video stream |
| ID | UC06 |
| Actors | Users |
| Pre-condition | ● Must be connected to a room<br>● Must have a sound device enabled |
| Post-condition | The user is connected to the voice communication server |
| Main flow of events | 1. It starts with the user choosing to enable voice communication on the chat room<br>2. A WebRTC peer connection is then initialized<br>3. A signal is sent to the signaling server, comprised of the callers ICE candidate, requesting to join the channel of which the chat is located<br>4. The channel or group of users sends back their ICE candidate to the caller<br>5. If both agree on the same ICE candidate, they will be connected to the same audio stream |
| Alternative flow 2.1 | ICE candidate is wrong, no candidate<br>1. User gets informed that the connection was not successful |

Figure F.7: This Use Case illustrates the flow of events of when a user tries to connect to another user with audio and video through WebRTC.

# G Artillery Stress Test Summaries

```
Room Repository Stress Test

All virtual users finished
Summary report @ 16:21:31(+0200) 2021-05-31
  Scenarios launched:  33554
  Scenarios completed: 33548
  Requests completed:  33548
  Mean response/sec: 42.97
  Response time (msec):
    min: 26
    max: 4801
    median: 32
    p95: 39
    p99: 56
  Scenario counts:
    Get rooms: 33554 (100%)
  Codes:
    200: 33548
  Errors:
    ETIMEDOUT: 6

User Repository Stress Test

All virtual users finished
Summary report @ 18:46:32(+0200) 2021-05-31
  Scenarios launched:  33658
  Scenarios completed: 33658
  Requests completed:  33658
  Mean response/sec: 43.1
  Response time (msec):
    min: 32
    max: 289
    median: 35
    p95: 42
    p99: 48
  Scenario counts:
    Get users: 33658 (100%)
  Codes:
    200: 33658

Socket Server Stress Test

All virtual users finished
Summary report @ 18:50:25(+0200) 2021-05-31
  Scenarios launched:  120
  Scenarios completed: 120
  Requests completed:  517
  Mean response/sec: NaN
  Response time (msec):
    min: 0.1
    max: 2.3
    median: 0.1
    p95: 0.2
    p99: 0.3
  Scenario counts:
    Busy boys: 19 (15.833%)
    Lurker: 84 (70%)
    A mostly quiet user: 17 (14.167%)
  Codes:
    0: 517
  engine.socketio.emit: 517
```