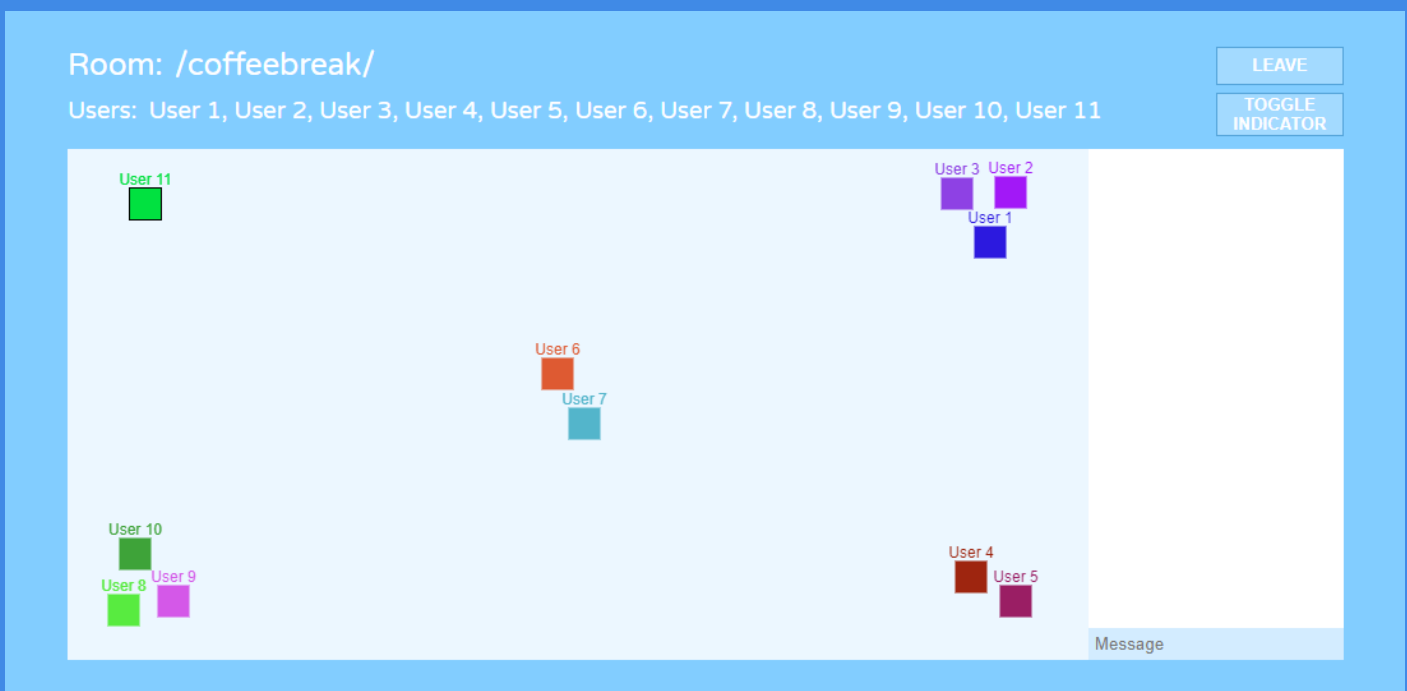


CoffeeBreak

A scalable 2D Audio Conferencing Platform

Bachelor thesis for Software Engineering, June 2021

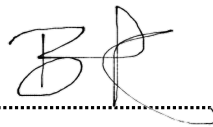


Approval

This thesis has been prepared as a submission for the final exam of the Bachelor's degree of Software Engineering at the University of Southern Denmark.

It is assumed that the reader has basic knowledge in software development, both in theory and practice.

Benjamin Hamborg Clement Klerens - bekle18



Signature

0 1 / 0 6 / 2 0 2 1

Date

Kevin Lavlund Hansen - kehan18

Rasmus Sjøholm Stamm - rasta17



Signature

0 1 / 0 6 / 2 0 2 1

Date



Signature

0 1 / 0 6 / 2 0 2 1

Date

Abstract

During work tasks that involve participants which are physically distant from one another, virtual live conferencing applications are used for collaboration, mimicking physical meetings. While these software applications provide a good alternative to physical meetings, especially given the current pandemic, they lack certain aspects in terms of fully simulating them.

In popular voice and video transmission applications such as Zoom, Microsoft Teams, or Discord, the typical approach is for each participant to be able to hear everyone at full volume at all times, disallowing the ability to have multiple conversations at the same time. However, this is possible in a real, physical space, as participants can move around and adjust their volume of speech. This limitation creates issues of disruptance when multiple conversations are attempted to be conducted at the same time, as every participant hears every other participant at the same, constant volume level.

CoffeeBreak shows that alleviating these issues is possible through the implementation of proximity-based volume adjustments. It is a system for simulating meeting participants being physically apart, automatically adjusting voice levels thereof. This system alleviates the problem by allowing participants to initiate isolated conversations with select other participants, while still being able to overhear other conversations, without being forced into using restrictive concepts, such as Zoom's "breakout rooms".

The result is achieved by utilizing the modern containerization technology of Kubernetes to dynamically distribute the service, which itself accomplishes real-time voice communication using the API specification of WebRTC. Tests show that the developed method of deployment and distribution in theory scales the application to upwards of 100 concurrent rooms, and that the implemented networking approach provides a seamless mouth-to-ear experience for users.

We imagine the current iteration of CoffeeBreak to be a cornerstone for future implementation of its concepts and findings, should such a development ever take place. This could perhaps be as an addition to existing solutions, either natively or as an "add-on", such that these solutions consume the efforts of this project.

Reading Guide

Report structure

This technical report is structured to tell a chronological story, starting from the inception of the project and ending at its conclusion. To experience the report as it was intended to, please read through each chapter in their numerical order, however this is not to say that each chapter cannot be individually understood. Each main chapter assumes that the reader has read and understood any preceding chapter, and so retrospective cross-referencing to previous findings will occur.

Technical wording

Throughout the report, many technical and often context-specific terms and expressions are used to describe various systems and concepts. These can relate to units of software, modeling concepts, existing system specifications, etc. An attempt has been made to generate a glossary (page **iv**), which aims to elaborate on any term and expression which might fit the aforementioned description.

References

Where applicable, the text will cite sources to support certain statements, meaning the statement is based on the material of the source, either in the shape of verbatim citations or reworded paraphrasings. A citation is labeled with a reference number surrounded by brackets, where the number refers to a corresponding reference in the References section (page **59**).

Appendix

As a general rule of thumb, figures which have been scaled to fit with surrounding content, along with significant results of the project, can be found in the highest resolution and/or quality possible in the Appendix section (page **60**). The appendix is indexed by numbering and sections, making it easy to navigate.

Glossary

| Expression | Full extension | Description |
|-----------------|----------------------|--|
| Docker | | |
| Docker | Docker | Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers [1] |
| Image | Docker Image | A Docker Image built from a Dockerfile |
| Container | Docker Container | A Docker Container created from a Docker Image |
| Kubernetes | | |
| k8s, Kubernetes | Kubernetes | Kubernetes, commonly stylized as K8s, is an open-source container-orchestration system for automating computer application deployment, scaling, and management [2] |
| Cluster | Kubernetes Cluster | A set of Node machines running containerized applications |
| Namespace | Kubernetes Namespace | A virtual Cluster inside the physical Cluster |
| Pod | Kubernetes Pod | A group of one or more containers within a Cluster or Namespace |
| Ingress | Kubernetes Ingress | A Kubernetes resource for managing access to the internal Services of a Kubernetes Cluster |
| Web development | | |
| Node, Node.js | Node.js | Node.js is an open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser [3] |

| | | |
|-------------------------------|-----------------------------|--|
| HTML | HyperText Markup Language | The HyperText Markup Language, or HTML is the standard markup language for documents designed to be displayed in a web browser [4] |
| CSS | Cascading Style Sheets | Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language such as HTML [5] |
| JS | JavaScript | JavaScript, often abbreviated as JS, is a programming language that conforms to the ECMAScript specification. JavaScript is high-level, often just-in-time compiled, and multi-paradigm [6] |
| Real-time voice communication | | |
| RTC | Real-time communication | Typically relates to the concept of communication via audio and/or video over the internet in real-time |
| WebRTC | Web Real-Time Communication | WebRTC (Web Real-Time Communication) is a free, open-source project providing web browsers and mobile applications with real-time communication (RTC) via simple application programming interfaces (APIs) [7] |

Table 1: Glossary

Contents

| | |
|--|-----------|
| Abstract | ii |
| 1 Introduction | 2 |
| 1.1 Project goals | 2 |
| 1.2 Learning objectives | 2 |
| 1.3 Motivation | 3 |
| 1.4 Process | 3 |
| 1.5 Hosting | 4 |
| 1.6 Source code | 4 |
| 2 Literature Review | 5 |
| 2.1 Exploring existing solutions | 5 |
| 2.2 Technology research | 7 |
| 2.3 Conclusion | 10 |
| 3 Analysis | 11 |
| 3.1 System description | 11 |
| 3.2 Use-case modeling | 11 |
| 3.3 Requirements specification | 13 |
| 4 Design | 16 |
| 4.1 Domain modeling | 16 |
| 4.2 Containerization modeling | 17 |
| 5 Implementation | 19 |
| 5.1 Microservices & containerization | 19 |
| 5.2 Backend | 28 |
| 5.3 Frontend | 34 |
| 5.4 Deployment | 40 |
| 6 Verification | 43 |
| 6.1 Functional requirements | 43 |
| 6.2 Nonfunctional requirements | 46 |
| 7 Conclusion | 53 |
| 7.1 System requirements | 53 |
| 7.2 Project goals | 54 |

| | |
|--|-----------|
| 8 Reflections | 55 |
| 8.1 Product | 55 |
| 8.2 Process | 57 |
| Literature | 58 |
| A Appendix | 60 |
| A.1 Use Case diagrams | 60 |
| A.2 Domain model | 61 |
| A.3 Containerization models | 62 |
| A.4 Sprint backlogs | 63 |
| A.5 Sequence diagrams | 64 |
| A.6 User interface | 69 |
| A.7 Lobby Dockerfile | 71 |
| A.8 Room Dockerfile | 72 |
| A.9 Kubernetes deployment file | 72 |
| A.10 Network test | 74 |

1 Introduction

This chapter aims to introduce the purpose of the project, its environments, goals, as well as ambitions and motivations of the project group. Additionally, the development process and methodologies that the project has followed are shortly documented as well.

1.1 Project goals

The end goal of this project is to develop a proof of concept application, called **CoffeeBreak**, as a **scalable 2D voice communication platform** which incorporates proximity of users to adjust volume levels between them dynamically. CoffeeBreak seeks to alleviate the issues which arise when real-time voice calls have many actively speaking users, the most noteworthy being that only one conversation can take place at any given time, since all users are speaking to one another at the same volume. CoffeeBreak's gimmick is that any user in a given "room", will only hear other users which are "close" to them, at a **coherent** volume level, while those far away can still be heard, however at a **non-disruptive** volume level.

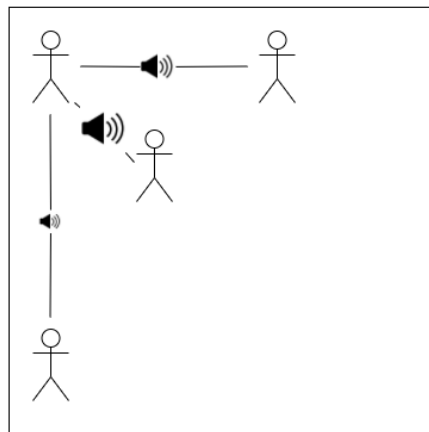


Figure 1.1: CoffeeBreak concept

From the perspective of formality, a goal of the project is also to meet a set of **learning goals** (not to be confused with the *learning objectives* of the following section), such that the project group illustrates their skills within the practice of Software Engineering.

1.2 Learning objectives

During the project, the group seeks to learn about scalable technologies, as their growing importance in the industry is becoming clearer, mostly looking at the last decade. In context of the project, scaling is very important, as we want to learn how to serve multiple users, and continue giving them a high quality audio experience, even at higher workloads.

Additionally, the group seeks to learn about web-based technologies which will assist the project in establishing a solid and simple frontend for the users to communicate through. Lastly, for the project to be able to connect audio between users, the group will strive to learn about technologies which offer a high quality and low delay solution for voice communication, so that users communicating through the system are satisfied and can conduct conversations seamlessly.

1.3 Motivation

Multiple well-established solutions for voice communication already exist, such as Discord, Zoom or Microsoft Teams. It would be hard to imagine CoffeeBreak making it as a standalone solution, having to compete with aforementioned giants, but the learning opportunities deriving from developing such an application are, in the eyes of the project group, extensively valuable. While the members of the group have limited experience in web development, none of them have previous experience working with audio streaming or scaling of applications. This proves a great opportunity for learning, and will help keep the project challenging and engaging.

1.4 Process

This project was not conducted following any strict software process. While the group has first-hand experience with using such processes, such as the *Unified Process*, it was ultimately decided not to constrict progress of such rules. The reason for this, was that the group, with aforementioned experience at hand, did not feel that a full-size process was applicable in context of the scale of the project. Instead, the core principles of *agile development*, such as *incremental delivery* and *rapid prototyping* were adopted for the group work. These were, on the contrary, very applicable, since the group was essentially working with technology of which they had no experience, and so being agile was essential.

To manage the many work tasks, a simple implementation of Scrum was used. This helped to keep track of *what had to be done*, *what was being done* and *what was done*. The project essentially consisted of three separate sprints, each with their own backlog of tasks.

The first sprint, initiated at the very start of early development, aimed to produce a functional prototype with a bare-bones user interface, without featuring containerization and scalability. The backlog for this sprint is available in appendix A.4.1.

The second sprint aimed to build on the results of the first, by introducing containerization of existing software, and deploying them to the Kubernetes Cluster in a scalable manner. The backlog for this sprint is available in appendix A.4.2.

The final sprint aimed to manage finalizing the project report, highlighting potential points of shortage.

1.5 Hosting

As part of the project supervision process, a Namespace and Service Account within an externally hosted Kubernetes Cluster were made available for the group to develop for, and deploy within. This is the Cluster that CoffeeBreak has been tailored to, and is also currently running on. Only having access to a Service Account meant that the group did not have *full* control over the Cluster, so certain limitations were present. Actions such as adding an Ingress controller of our choice was not a possibility, however the Cluster included an NGINX Ingress Controller for handling the Ingress configuration, and this proved sufficient for what was required.

1.6 Source code

As the purpose of this report is to document a piece of developed software, it would be remiss not to include said software in its purest form, this being its **source code**. This can be accessed by visiting the GitHub repository, at

<https://github.com/KevinLHansen/Bachelor-CoffeeBreak>.

Please note that the report documents the following "release":

<https://github.com/KevinLHansen/Bachelor-CoffeeBreak/releases/tag/v0.2>

2 Literature Review

This literature review aims to explore available literature regarding topics relevant to the core problem of the project. In other words, technologies or subjects which require a general understanding in order to be implemented in the project. These topics are found using the initial project description, to envision what the project result might be, and formulating a general problem statement to represent it. For the context of this literature review, the problem statement is formulated as such:

"How can we develop a dynamically scaling application system for real-time voice communication between users, and which practices and technologies exist and are relevant to accomplishing this? Furthermore, how do we measure the level of scalability of the system, such that we can deem the implementation a success?"

Using this statement as a starting point, we will look upon existing technologies and solutions, such as Zoom, Discord and Microsoft Teams to extract knowledge that will act as a solid baseline to help answer the problem statement. A technology research will be conducted as well, to find applicable solutions for the project's problems of scalability and voice communication.

2.1 Exploring existing solutions

This section will briefly look upon existing solutions for scalable real-time voice communication implementations. Zoom, Discord and Microsoft Teams have been placed in focus, as these applications are relatively new in terms of age, and have gained a decent amount of popularity in contrast to other similar applications.

2.1.1 Zoom

Zoom is a cloud-based video and audio service allowing individuals and businesses to interact in virtual rooms. It offers a number of core functionalities, such as meetings with participants sharing video and audio, public and private chatting in-room, selective screen-sharing, etc. Additionally, Zoom's daily meeting participant count has skyrocketed during the pandemic, from 10 million in early 2019, to 350 million as of december 2020 [8]. Providing a real-time video and audio application with an extensive toolkit to create meetings for a variety of purposes like educational, fitness or family meetings has paved the way for success.

Zoom has a wide technology stack, using Objective C and Swift for iOS applications, and Java/Kotlin for Android applications. For web based applications, WebRTC-based API's are used for real time communication with HTML, CSS and JavaScript for the presentation [9].

2.1.2 Discord

Discord is a guild-based, or often referred to as server-based, voice chat application for organized communication. A guild represents an isolated collection of users and channels, and the ability to create guilds allows for users to build communities for any purpose. Text communication in Discord is either directly between users as "direct messages", or within text-channels of guilds. The same goes for voice calls, albeit in the voice channels of guilds. Guilds are also role-based, which means that users are allowed or disallowed certain access depending on what role they have [10].

Discord offers a standalone application that has been built using Electron, which is a framework for building desktop applications using web technologies like HTML, CSS and JavaScript. Electron also uses Chromium and Node.js. Like Zoom, Discord also runs in almost any browser, and is available on Android and iOS.

While Discord works both in-browser and as standalone application, RTC approaches of these differ slightly. The in-browser RTC features of Discord are implemented using the WebRTC API offered by any modern browser, while the RTC features of the application consist of a C++ library which is built on top of the native WebRTC library. Having control of the native library allows modification of the lower level WebRTC API, letting Discord exchange minimal information when users initiate voice communication [11].

2.1.3 Microsoft Teams

Microsoft Teams is an application developed by Microsoft that focuses on meeting, chatting and collaborating in one place. As with any Microsoft app, it offers powerful Microsoft Office integration as well as comprehensive third-party options [12]. Teams primarily attempts to cater to the business environment, focusing heavily on team-based communication to increase productivity. However, it does see uses within the home and education environments as well. The application also supports compliance with numerous privacy regulations, heavily supporting the healthcare industry.

Like Discord, Teams has both an in-browser and a standalone application, with both of them having their real-time communication based on the WebRTC API [13]. Using these already existing technologies such as WebRTC is preferable because of the simplicity and availability they provide, among many other benefits. The standalone application is also based on the Electron framework [14]. Discord and Teams are not very different in terms of technologies used for the communication part of the overall solutions. However, the way the frontend is delivered, along with the structure of their backends, deviates significantly. It is not unlikely these days to see modern software communication solutions to be based on WebRTC, either with or without a standalone desktop application. Web-based applications are usually cross-platform, since every standard desktop operating system supports at least one or more modern browsers, and the WebRTC API is integrated into them.

2.2 Technology research

2.2.1 Real-time voice communication

During research of existing real-time communication methods and technologies, it becomes clear that the WebRTC API is a very popular contender, while older methods such as VoIP (Voice over IP) fall behind. [15] states that:

"Web Real-Time Communication (WebRTC) is a new standard",

implying that the framework is broadly implementable and functionally competent. The broad implementation of WebRTC in popular applications, such as Discord and Facebook Messenger [16], supports the idea of WebRTC being a leading standard of live audio/video conferencing. [15] also describes the work put into making WebRTC a very standardized API, able of working with any modern internet browser. While methods for obtaining, streaming and displaying media between clients have been standardized, the process of *signaling* is left open to custom implementation. [15] suggests SIP (Session Initiation Protocol) as a potential candidate, but also states that this is ultimately up to the developer. This is especially relevant in context of the project, since the service for establishing and handling connections we envision to develop is to be containerized, and so must be built as such. Furthermore, the browser-capabilities of the framework allows us to build a highly compatible application while using development platforms we are already familiar with, these being HTML5 and JavaScript, granting a boost to project work efficiency.

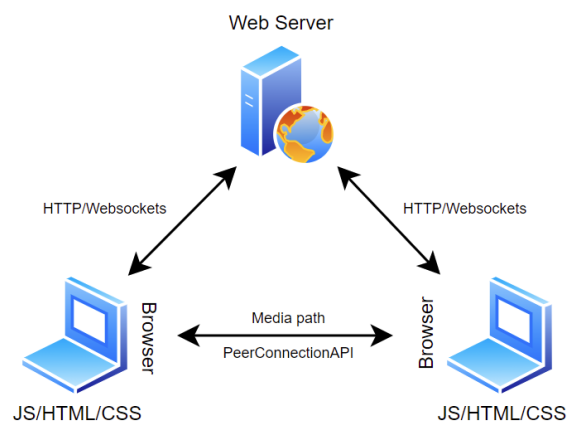


Figure 2.1: The WebRTC triangle

[15] also describe what they call the "WebRTC triangle" (figure 2.1), which models the networking architecture of using WebRTC in browser-based applications. This also speaks to the project, as this modeling closely matches our initial impressions of the desired system architecture of the project solution, where the web server, responsible for connections and serving the app to clients, is containerized and made scalable.

2.2.2 Virtualization

Virtualization has allowed developers to craft virtual environments tailored to specific requirements, such as allowing one machine to host multiple guest operating systems. While virtual machines still have their uses, containers have become the popular approach when working with microservices. As stated in [17],

"The late 2010s saw significant adoption of containers over traditional VMs."
[17, p. 120]

It also mentions that:

"In general, it is best to avoid running microservices architectures on virtualized infrastructure."
[17, p. 120]

This information would indicate that we should mainly look at containers when trying to make our system scalable, as starting a new container is faster and is less resource intensive than starting a new virtual machine.

According to [18], containerization is an alternative or companion to virtualization. It is incredibly lightweight compared to a virtual machine, as a container does not require the overhead of associating an operating system with an application, because it shares the host's operating system kernel. Containers are also inherently smaller in capacity and require less start-up time, allowing far more containers to run in the same environment compared to a virtual machine. Traits such as portability and scalability are products of the aforementioned qualities, which are highly sought after in large scale projects, as well as in our own.

2.2.3 Microservices and containerization

The microservices architecture is a relatively new concept conceived in the early 2010s, gaining a popularity throughout the decade. However, the principles that the architecture is built upon originates all the way back to the late 1960s, from three computer science and software engineering areas: *programming language concepts*, *systems architecture concepts* and *software architecture concepts* [17].

Microservices, according to [19], use containerization to deliver small, single-function modules, which work in tandem to create more agile, scalable solutions. This approach to creating applications eliminates the need to build and deploy an entirely new software version, whenever a function is changed or scaled. In addition to that, [17] states that these units and microservices are built, modified and retired and then replaced without impacting the overall solution.

The traditional architecture, the monolithic architecture, is often mentioned when researching the microservice architecture. The monolithic architecture is where the solution is all composed in one piece, as opposed to the microservice architecture, where components are split into

different services, thus the name, microservices.

Comparing the architectures gives us insight into which one is suitable for the project. The monolithic approach has some advantages over microservices architecture, but this also works the other way around. The monolithic architecture typically benefits teams with simple applications, since the microservices architecture adds complexity to the solution which requires specific expertise in the field. Another benefactor of choosing monolithic over microservices is testability. This is because testing multiple independently deployable components, which is a result of the microservice architecture, is a rather difficult task. However, one of the most important traits of using a microservices architecture, is increased scalability. A lot of companies end up rebuilding their monolithic architectures into a microservices architecture, since the entire development and maintenance process becomes more cost and time effective. Usually you would need to scale the entire solution, but this is no longer the case with the microservices architecture, since whichever component requires scaling, can be scaled independently [20]. In context of the project, it might then prove beneficial to implement parts of the solution as microservices, as described above, to obtain the benefits hereof. For example, we know that a service for routing communications data is required, and such might be a valid candidate for implementation as a microservice.

2.2.4 Scalability

"Scalability is the ability of systems to handle an increased workload"
[17, p 35]

While looking into the concept of scaling, it becomes clear that there are many factors and unknown parameters that needs to be accounted for, when trying to scale a system. Thus many assumptions have to be made, such as which operations will be the most and least common. This is important to know as to not create bottlenecks in the system while trying to scale it [21]. Multiple models exist for analyzing the scalability of a system, each with their own parameters, but general units of measures could be size or load, and work or throughput [22]. These units can be used to show the correlation between how much throughput we get when we increase the size, e.g. the amount of containers, of a system.

[23] proposes measuring the scalability of a system with a generalized mathematical formula. This approach is aptly named the *Universal Scalability Law*, or USL for short, and it aims to calculate the increase in *throughput* as the system's number of *nodes* increase.

$$C(p) = \frac{p}{1 + \sigma(p - 1) + \kappa p(p - 1)}$$

USL can be formulated as above, where C is the resulting throughput of having p nodes in a system with σ unparallelizable workload and κ cross-talk penalty. In relation to the project, p would represent the number of pods running in a Kubernetes cluster serving the same purpose,

σ the required load of instantiating new pods dynamically, and κ the potential communication between pods and some central information storage (e.g. database or similar).

2.3 Conclusion

In this section, we aim to answer the problem statement introduced in the introduction, using the newly reviewed literature, while highlighting any shortcomings and uncertainties.

Looking into architecture selection, both the microservices and monolithic architecture provide beneficial factors towards the project. However, the microservices architecture excels in providing scalability for the solution, whereas the monolithic architecture's support for scalability is quite limited in comparison. Therefore while both architectures are compatible, microservices just provide a significant edge in terms of functionality for scalable solutions.

In terms of making the solution scalable, it is important to realize while creating the system which components will be utilized the most and which services they rely on. This creates a gap in current knowledge, as at the time of writing, we are not yet at a point in development to be able to do this. We have only slightly looked into the concept of containerization, which is important for the microservices architecture. However, using containerization for large scale solutions requires a container orchestration tool such as Kubernetes, which will be a future area of study. In regards to measuring the scalability of our system, the Universal Scalability Law has been deemed a good approach, as it relies on seemingly easily obtainable parameters.

The reviewed literature regarding WebRTC and its specification has given us an entrance into the technology of real-time communication, while also providing us with a potential solution baseline for implementation. Using WebRTC in connection with experience we already have with HTML5 and JavaScript seems to be the way forward. This solution allows us to create a highly compatible and distributable application with our desired functionality.

While we have confirmed that WebRTC allows an abstract method of signaling, it is still unclear exactly how this part should be built, and the potential limitations hereof. The literature mentions ideas such as SIP (Session Initiation Protocol) for initiating the connection between peers, so this topic might prove a good point of research for future review.

3 Analysis

This chapter aims to analyze the initial system description, by modeling interaction between user and system from the user's point of view, and translating these models into specific functional and non-functional system requirements for future implementation. This approach ensures coherency between how the user wants to use the system, and how the system actually functions.

3.1 System description

CoffeeBreak is an isolated concept whose purpose is to simulate varying audio levels based on the physical distance between the listener and speaker. This physical distance is simulated by including a two-dimensional interface, where each user can pick up and place their avatar, where the measured distance between two avatars will determine the audio level between the two users. This way, it becomes possible to have multiple conversations in one **room**, without having the participants potentially yelling over each other. Participants are able to move their avatars and "join" into other ongoing conversations, much the same way a person would be able to walk over to a group of people talking about a topic which might interest them.

3.2 Use-case modeling

To quickly and efficiently translate the user's description of what they desire from the system, standard UML use-case diagrams are generated from said descriptions. This process will divide the provided description(s) into smaller, more specific functionalities, providing an overview of the user's "case of use".

3.2.1 User stories

Based on the system description in section 3.1, *user stories* are written, to put into words how a user expects to use the system. A user story is a short, generalized way to describe the user experience, and typically follows the following format:

As a [role], I want to [action], so that [goal].

An example of using this format in relation to the project would be as such:

*As a **user**, I want to **join a room**, so that **I can communicate with others**.*

The aim is to write enough user stories, so that the suggested functionality of the system description is fully covered in this format. For the sake of brevity, user stories are compiled in a table.

| ID | Role | Action | Goal |
|-----|------|-------------------------------------|---|
| US1 | User | create a room | communicate with others |
| US2 | User | close room | terminate session |
| US3 | User | join a room | communicate with others |
| US4 | User | leave a room | end communication with others |
| US5 | User | invite other user | communicate with others |
| US6 | User | assign & change nickname | achieve personal identification |
| US7 | User | specify own location in room | communicate with other nearby users |
| US8 | User | hear a simulated stereo sound-scape | immerse oneself in the session, identify speakers from volume level and direction |
| US9 | User | mute and unmute own microphone | control communications |

Table 3.1: User stories

3.2.2 Use-case diagrams

Now that a list of user stories has been identified, they can be compiled into use case diagrams, as to provide a broader perspective of the system as a whole. The primary goal here is to show the relation between user stories and potential future requirements, while also exploring potential external/supporting actors of the system.

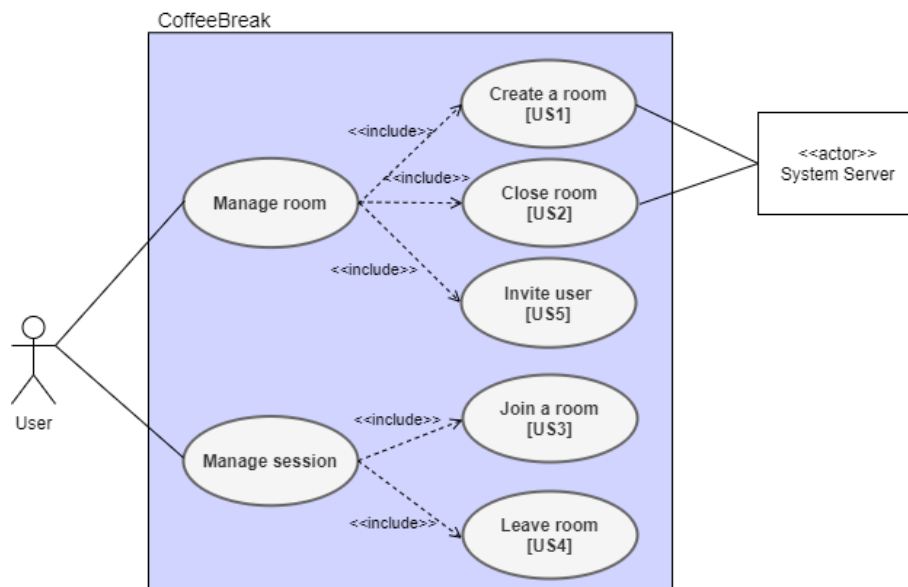


Figure 3.1: Use Case diagram 1, CoffeeBreak

Figure 3.1 models user stories US1 through US5, which have been found to fit under two, more general use cases, these being *Manage room* and *Manage session*. While this discovery does not affect the system functionally, it helps to model it, and the hierarchization will be reflected in the resulting requirements specification. Another thing to note is the addition of the supporting actor *System Server*. This is the first step of exploring how the system and its supporting actors might react to user actions.

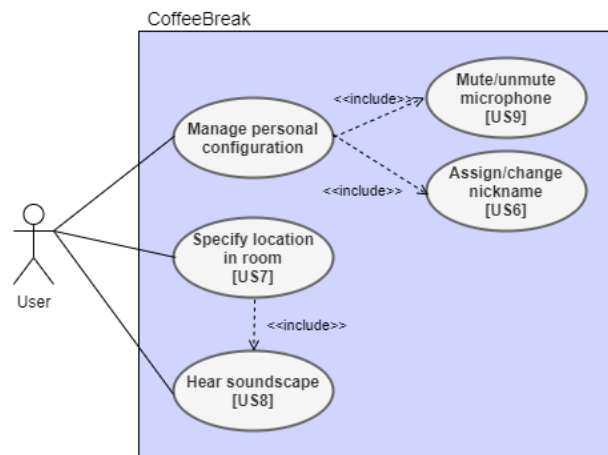


Figure 3.2: Use Case diagram 2, CoffeeBreak

Figure 3.2 models user stories US6 through US9. Like with figure 3.1, it identifies a new generalized use case, *Manage personal configuration*.

3.3 Requirements specification

User stories and use case models are translated to a more implementation-specific format, the requirements specification. The aim of this model is to be universally understandable, and implementable regardless of context.

The requirements specification is divided into two groups, one being that of *functional requirements*, and the other, *nonfunctional requirements*. This is done to simplify prioritization, where successful implementation of the *functional group* is vital for the functionality of the system, while the *nonfunctional group* speaks to its measure of "goodness".

All functional requirements have been further prioritized using the MoSCoW method, as a way of focusing work on the most important tasks at hand. The prioritization listed in table 3.2 was carried out with the goal of producing a minimal viable product, or "proof of concept" within the allocated project duration. For the uninitiated, MoSCoW prioritization is denoted as follows:

- M: Must-have, vital for the success of the system
- S: Should have, important in terms of functionality, but not 100% necessary
- C: Could have, not vital, but would still benefit the system

- W: Will not have, no resources will be allocated to this at this iteration

| ID | Description | MoSCoW | Evaluation |
|------------------------|--|--------|----------------------|
| Rooms | | | |
| FR1 | The system has rooms which encapsulate communication sessions and their participant users | M | Design inspection |
| FR1.1 | Users can create rooms | M | User test, Unit test |
| FR1.2 | Users can invite other users to their room | S | User test |
| FR1.2.1 | A room has a unique ID which can be given to invitees | M | Design inspection |
| FR1.3 | Users can join rooms they are invited to | M | User test |
| FR1.3.1 | A room can be joined using its unique ID | M | User test, Unit test |
| Personal configuration | | | |
| FR2 | Users can manage their identity | C | Design inspection |
| FR2.1 | Users can assign and change their nickname | S | User test, Unit test |
| FR3 | Users can manage their sound settings | C | User test |
| FR3.1 | Users can mute and unmute their microphone | C | User test |
| FR3.2 | Users can adjust incoming sound | C | User test |
| FR3.2.1 | Users can adjust level of all incoming sound gradually | C | User test |
| FR3.2.2 | Users can mute individual participants in the room | C | User test, Unit test |
| Soundscape | | | |
| FR4 | Users can hear a simulated soundscape based on location of self and other users | M | Design Inspection |
| FR4.1 | Volume of speakers varies based on distance from user (farther = lower) | M | Unit Testing |
| FR4.2 | Sound of speakers play for user in their relative direction via stereo | C | User test |

| | | | |
|-------|---|---|-------------------|
| FR5 | User has an avatar in a 2D representation of the room and can manage it | M | Design inspection |
| FR5.1 | User can place and move their avatar within the 2D plane | M | User test |

Table 3.2: Functional requirements, CoffeeBreak

| ID | Description | Evaluation |
|------|--|-------------------------------------|
| NFR1 | Capable of 50 concurrent users in a room | Performance test |
| NFR2 | Scalable up 100 rooms | Design inspection, performance test |
| NFR3 | Low latency for voice communication | Network test |

Table 3.3: Non-functional requirements, CoffeeBreak

NFR1 was set with the idea that the system should cater to larger workspaces with many employees. The same can be said for NFR2, being focused more around supporting larger groups and companies, the need for many rooms increases. NFR3 is a global rule for communication, if latency goes above a certain threshold in a system with multiple actors, communication eventually becomes unbearable.

4 Design

This chapter aims to model an initial concept of the software architecture of the system. The goal is to imagine how the system might be built, and how it might interact with its environment. This process is based on knowledge gained in previous sections, and acts as the final step of preparation before implementation begins.

While you might expect this chapter to contain one or more UML Class diagrams, we ultimately decided not to construct any at this stage. This was largely due to uncertainties and abstractions in regards to programming language (which was yet to be chosen) and how this might play into the process of containerization.

4.1 Domain modeling

Using concepts identified in section 3, following domain model has been constructed, with the goal of mapping relationships between said concepts.

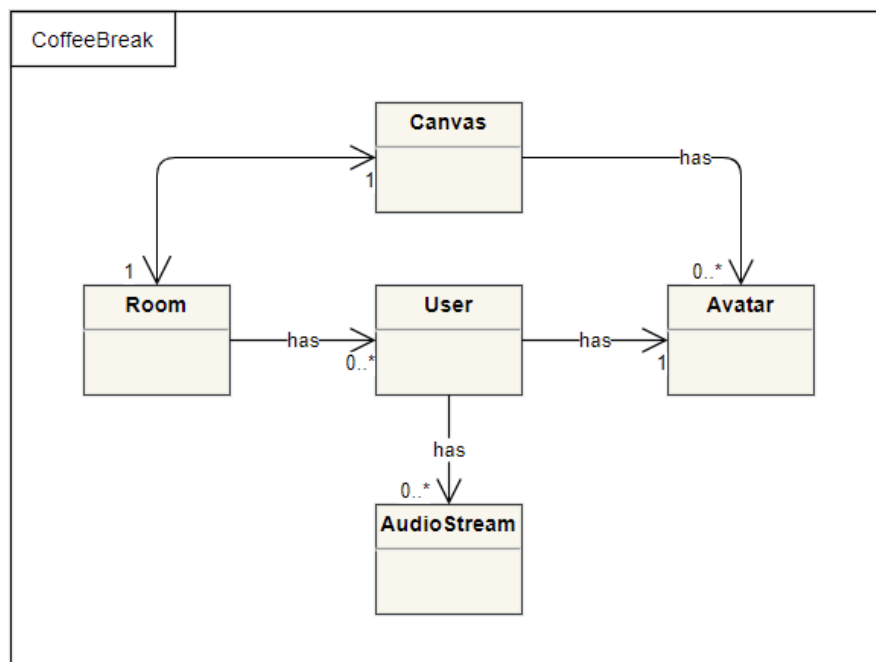


Figure 4.1: Domain model, CoffeeBreak

In section 3, we learned that the concepts of rooms, users, and a 2D canvas with avatars should be present, and in early research of the WebRTC API, it was found that media streams of users exist as separate objects, and all of these concepts are represented in the domain model. While this model does provide a broader perspective, it fails to imagine how the system might scale

in regards to containerization. The model also leaves out how new rooms are created. This is done intentionally, as this functionality is very likely to be part of the containerization solution, and as such must be further explored to model.

4.2 Containerization modeling

To imagine how we might use Kubernetes to containerize and scale the system, a conceptual model is constructed. This model aims to plan how, when and where units of the system (e.g. containers) are created, which functionality they serve, and how they interact with the system. Since we plan to use Kubernetes to orchestrate containers, the terminology hereof is used. Please note that this model is highly conceptual and abstract, and does not follow any standardized modeling language, such as UML.

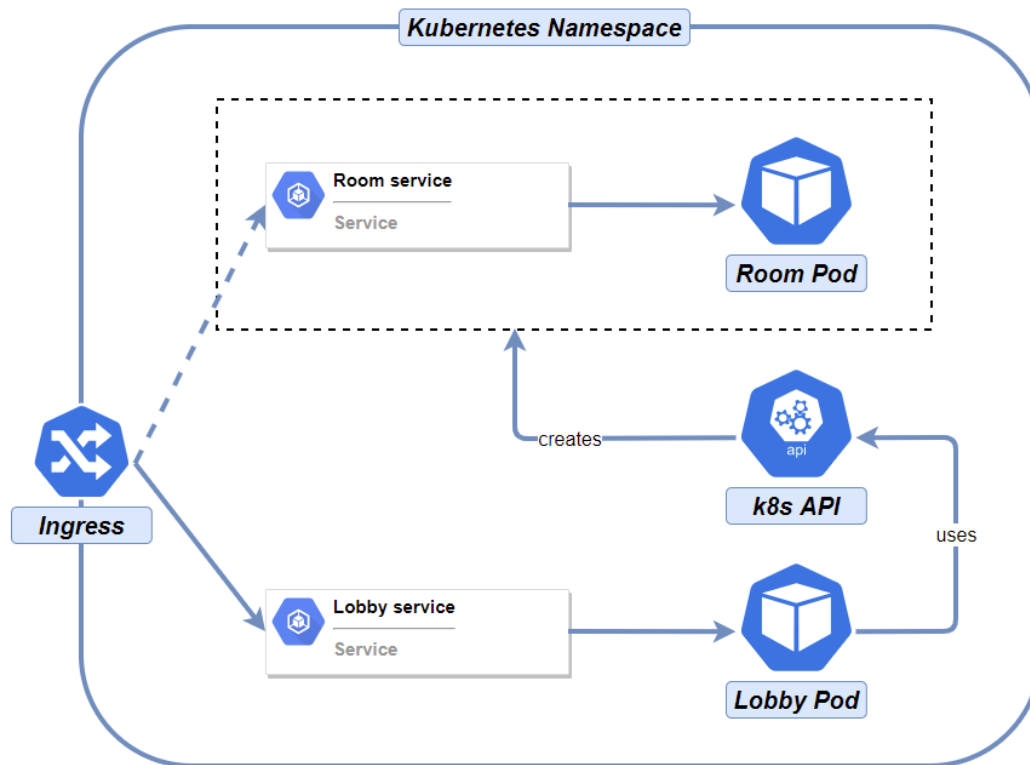


Figure 4.2: Containerization model 1, CoffeeBreak

In context of the project, we were given access to an externally run Kubernetes Cluster, albeit restricted to a specific Namespace. For this reason, figure 4.2 is confined as such. Essentially, the idea is to have an Ingress object act as the entry point of the system, redirecting users to a stateless *Lobby Pod* through an accompanying *Lobby Service*. The user, using the functionality contained within the Lobby Pod, can create rooms, causing the Pod to utilize the Kubernetes API to create a new stateful *Room Pod* along with its necessary components. These are: the Pod itself, a Service pointing to the Pod, and redirection configuration data for the Ingress object.

With this concept, we imagine the entire system to be scalable. The Lobby Pod, being fully stateless, requires no persistent knowledge besides what the k8s API can provide, and thus should be automatically scalable, e.g. by a load-balancer. Room Pods on the other hand, while being stateful, are only created when their existence is desired (i.e. a user creates it when they need it). This means that, as long as automatic down-scaling is implemented for Room Pods (e.g. automatic close when zero users), the system should be fully scalable.

4.2.1 Kubernetes API

Figure 4.2 shows the Lobby Pod using something called *k8s API* to create various objects within the Namespace. This is the Kubernetes API. A Kubernetes instance has an internal API server which, as its name suggests, makes an API available for manipulating the state of the internal API objects, these being Pods, Namespaces, etc. The Kubernetes API is standardized, and thus has implementations for various programming languages, making it broadly implementable. In context of figure 4.2, it is the link which enables the Lobby Pod to create new instances of the room-relevant components. Additionally, it is also what allows us, the developers, to develop and deploy within the Cluster, by using tools such as *kubectl*.

5 Implementation

In this chapter, how CoffeeBreak was implemented, as a response to previously defined goals and requirements, is documented. This includes, but is not limited to, the containerization structure of the system, how it has been deployed, and the essential software behind the back- and frontend. The aim is to explain any and all vital systems and components, however without creating a line-by-line walk-through of the codebase.

5.1 Microservices & containerization

5.1.1 Containers: Docker

In the CoffeeBreak project, a total of two container Images were designed and built with Docker. The first Image in table 5.1 is *coffeebreak-lobby*, which is a microservice that, in simple terms, functions as a waiter for incoming users. The second Image, *coffeebreak-room*, is the destination which the waiter (*coffeebreak-lobby*) directs a user to.

DockerHub is a service developed by Docker, where teams and users publicly and privately share their Docker Images through Image repositories. The Images are hosted online and integrate seamlessly with the Kubernetes declarative deployment files and command-line tools. The Docker Images developed during the course of the project are listed with links to their respective DockerHub repositories in table 5.1.

| Docker Image Overview | |
|-----------------------|---|
| Name | Link |
| coffeebreak-lobby | https://hub.docker.com/r/benjaminhck/coffeebreak-lobby |
| coffeebreak-room | https://hub.docker.com/r/benjaminhck/coffeebreak-room |

Table 5.1: Docker Container Images of the CoffeeBreak system

The base Image used for both Images is Node Alpine, specifically named *node:16-alpine3.11*. Alpine is a small, simple and secure distribution of Linux. Node.js is not usually associated with just Linux or Alpine, but this specific Image, which is maintained by the Node.js developers themselves, has the Node.js framework included with the Alpine Linux distribution. This allows us to run our Node.js project files within the Image Container, without having to install anything on it, thus greatly reducing spin-up time.

By using Node Package Manager, which is included with the Node framework, it is possible to install the project's dependencies internally on the Images upon setup using their respective

Dockerfiles. Once again, this greatly reduces spin-up time, since all dependencies are immediately available from the Image which spun up containers are based on. This is utmost necessary, as a large portion of the codebase consists of imported libraries such as the Node packages of Kubernetes Server-Client API and WebSockets, which are non-native to the JavaScript standard library, and thus must be installed on any service using them.

In table 5.2, a brief description of the *coffeebreak-lobby* Image is presented. It shows the name, DockerHub repository, the base Image it was created from, as well as the Node dependencies and exposed ports. These ports will become very important at a later stage when Kubernetes is introduced. Without exposing ports on the container, we wouldn't be able to reach the implementation inside of it.

| Image description | |
|----------------------|---|
| Name | coffeebreak-lobby |
| DockerHub repository | https://hub.docker.com/r/benjaminhck/coffeebreak-lobby |
| Base Image | node:16-alpine3.11 |
| Node dependencies | @kubernetes/client-node, express, request-ip, ws |
| Exposed ports | 80, 8082 |

Table 5.2: Detailed description of the coffeebreak-lobby Image

In table 5.3, a brief description of the *coffeebreak-room* Image is presented. What is noteworthy in the differences between the two Images, *coffeebreak-lobby* and *coffeebreak-room*, is the dependencies in the Node framework. The room Image itself does not implement any features from the Kubernetes API, and this is because it does not require access to it. Secondly, it would also prove a security concern, since this would place a service responsible for interacting with the k8s API very close to the end user, enabling potential exploitation.

| Image description | |
|----------------------|---|
| Name | coffeebreak-room |
| DockerHub repository | https://hub.docker.com/r/benjaminhck/coffeebreak-room |
| Base Image | node:16-alpine3.11 |
| Node dependencies | express, request-ip, ws |
| Exposed ports | 80, 8082 |

Table 5.3: Detailed description of the coffeebreak-room image

5.1.2 Container Orchestration: Kubernetes

To orchestrate a project with containerized applications, Kubernetes, an open-source system for automating deployment, scaling and management, is used. Although there are similar alternatives, Kubernetes is very widespread and supported, and it offers a very detailed documentation. It was originally designed and built by Google, and is now maintained by Cloud Native Computing Foundation. Kubernetes works by abstracting machines, storages and networks further away from their physical implementation. Using Kubernetes, it requires a single interface to deploy containers to virtual machines, physical machines, and even the majority of cloud providers like Amazon Web Services or Microsoft Azure.

There are several ways to deploy containers to Kubernetes, either by using the command-line, declarative deployment files, or through implementations of the Kubernetes API, which happens to be supported in multiple programming languages. CoffeeBreak uses a declarative deployment file for Kubernetes resources which we do not wish to change, and we will refer to these in the project as static Kubernetes resources. Resources created at runtime by the user are generated dynamically with the API, and these will be referred to as dynamic Kubernetes resources in the project.

5.1.3 Static Kubernetes resources

This section will describe the baseline Kubernetes resources in the Cluster.

As the project solution dynamically generates new resources with the Kubernetes API on user action, the resource list is ever-changing, and will therefore not be fully described in this section. The descriptions below will be based on the Cluster setup upon initial deployment of the solution.

The baseline Ingress resource consists of two rules: One to route requests from the domain root "/", and another to route requests from the '/ws' endpoint. A new rule is generated by the Lobby microservice through the Kubernetes API upon a user creating a new room.

| Ingress Description | |
|---------------------|---------------------------|
| Name | ing-coffeebreak |
| Kind | Ingress |
| API | networking.k8s.io/v1 |
| Ingress Rules | |
| Backend Path 1 | |
| Host | group2.sempro0.uvm.sdu.dk |
| Path | / |
| Service Name | svc-lobby |
| Service Port | 8888 |
| Backend Path 2 | |
| Host | group2.sempro0.uvm.sdu.dk |
| Path | /ws |
| Service Name | svc-lobby |
| Service Port | 8082 |

Table 5.4: Description of the Ingress resource

The svc-lobby Service resource is the destination for the rules from the baseline Ingress resource, or otherwise recognized as the middleman for requests between the Ingress controller and the destination Pod. The Service consists of two different port definitions, one for the Web Server targeting port 80 and one for the WebSocket Server targeting port 8082. The destination for these ports is on the selector Pod, which in this instance is the Lobby Pod.

| Service Description | |
|---------------------|---------------|
| Name | svc-lobby |
| Kind | Service |
| API | v1 |
| Type | Load Balancer |
| Selector | lobby (pod) |
| Port Definitions | |
| Definition 1 | |
| Name | web |
| Protocol | TCP |
| Port | 8888 |
| Target Port | 80 |
| Definition 2 | |
| Name | socket |
| Protocol | TCP |
| Port | 8082 |
| Target Port | 8082 |

Table 5.5: Description of the svc-lobby Service resource

Finally, the terminus of the route, the Lobby Pod. The pod resource itself does not contain that much information other than specifying its base Image and exposed ports, as most of the complexity is wrapped and stored in the Pod Image itself. The ports themselves must of course correspond to the port implementation inside the Image, otherwise the request will not reach the server inside the Pod.

| Pod Description | |
|-----------------|--------------------------------------|
| Name | lobby |
| Kind | Pod |
| API | v1 |
| Specification | |
| Image | benjaminhck/coffeebreak-lobby:latest |
| Container Ports | 80, 8082 |

Table 5.6: Description of the Lobby Pod resource

5.1.4 Dynamic Kubernetes resources

With the baseline Kubernetes resources established, the dynamic resources generated with the Kubernetes API can be introduced. While the two types, static and dynamic, are similar, their core difference is that static resources are generated on deployment, while dynamic resources are generated at runtime.

The dynamic Ingress solution itself is different to how the two other resource solutions are implemented, being Service and Pod. Instead of creating an entirely new Ingress resource, the existing one is altered to route a path to the newly created dynamic Pod through its accompanying Service.

The table below shows how the Ingress resource is altered upon creation of a new Room through the Lobby microservice. The Kubernetes API generates two new paths with values based on the desired name entered by the user, one routing to the Web Server and the other to the WebSocket Server.

| Ingress Description | |
|------------------------------|---------------------------|
| Name | ing-coffeebreak |
| Kind | Ingress |
| API | networking.k8s.io/v1 |
| Ingress Rules | |
| Backend Path 1 | |
| ... <i>svc-lobby service</i> | |
| Backend Path 2 | |
| ... <i>svc-lobby service</i> | |
| Backend Path X | |
| Host | group2.sempro0.uvm.sdu.dk |
| Path | /<Room Name> |
| Service Name | svc-<Room Name> |
| Service Port | 8075 |
| Backend Path Y | |
| Host | group2.sempro0.uvm.sdu.dk |
| Path | /<Room Name>/ws |
| Service Name | svc-<Room Name> |
| Service Port | 8082 |

Table 5.7: Description of the altered Ingress resource

The dynamic service on the table below is created with a name corresponding to the user input from the Lobby microservice frontend. This name is important, as it has to correspond to the Service, that is referred to in the Ingress resource. The newly created Service runs on its own IP address in the Cluster, which becomes handy when creating multiple Services in a single Cluster. As long as every individual Service does not have two equal definitions with the *Port* attribute, port collision is avoided. The *Selector* attribute is the option that points the service to a specific Pod, which in this case is based the user entry from the Lobby microservice.

| Service Description | |
|---------------------|--------------------------------|
| Name | svc- <Room Name> |
| Kind | Service |
| API | v1 |
| Type | Load Balancer |
| Selector | room- <Room Name> |
| Port Definitions | |
| Definition 1 | |
| Name | web |
| Protocol | TCP |
| Port | 8075 |
| Target Port | 80 |
| Definition 2 | |
| Name | socket |
| Protocol | TCP |
| Port | 8082 |
| Target Port | 8082 |

Table 5.8: Description of the dynamic service resource

Lastly, the dynamic Pod resource. Instead of being based on the coffeebreak-lobby Image, the dynamic Pod resource uses the coffeebreak-room Image. The Pod itself is created with a name that is cross-referenced in the dynamic Service resource. The description table is shown below:

| Pod Description | |
|-----------------|-------------------------------------|
| Name | room-< Room Name > |
| Kind | Pod |
| API | v1 |
| Specification | |
| Image | benjaminhck/coffeebreak-room:latest |
| Container Ports | 80, 8082 |

Table 5.9: Description of the dynamic Pod resource

Considering above documentation, it becomes apparent what the dynamic Kubernetes resources actually do. It is essentially building Ingress, Service and Pod resources with specific attributes. These three different Kubernetes resources have to be configured specifically to be able for the resources be linked together and scaled extensively.

5.1.5 Kubernetes Architecture

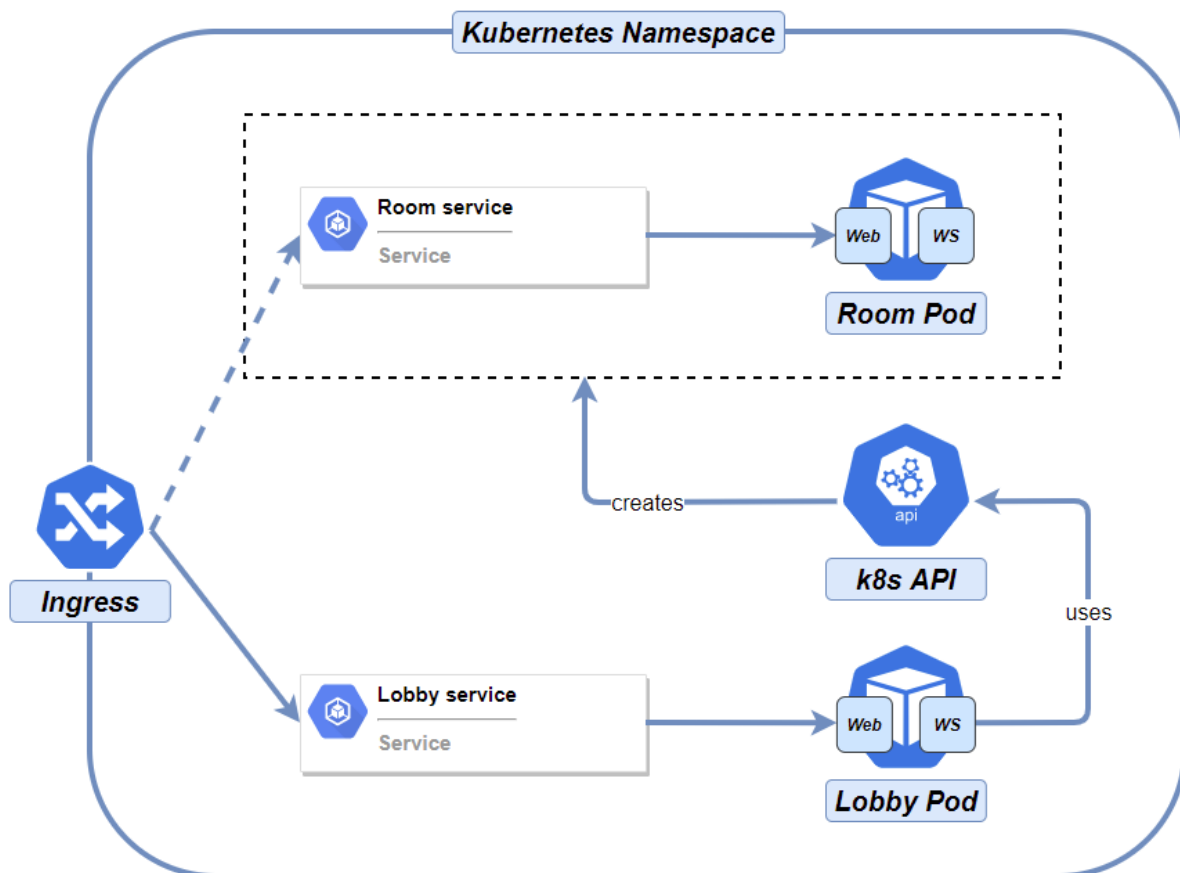


Figure 5.1: Kubernetes Cluster Overview, CoffeeBreak

Figure 5.1 shows the final modeling of the Kubernetes Namespace structure. While the structure is mostly the same as what was initially proposed in section 4.2, a noteworthy change is the introduction of a WebSocket Server ("WS") and Web Server ("Web") to each type of Pod. This change signifies that the Web Server of each Pod is responsible for receiving the user, as redirected by the Ingress resource, while the WebSocket Server of the Lobby Pod is responsible for interacting with the Kubernetes API.

5.2 Backend

Now that a general understanding of the overall structure of the system has been acquired, we can dive into the inner workings of each type of Pod, these of course being the *lobby* and *room* Pods.

5.2.1 Lobby

The Lobby is responsible for receiving the user and providing them with the functionality of creating and/or joining Rooms. This is accomplished by running both a Web Server and a WebSocket Server simultaneously.

The Web Server acts as the initial recipient of client requests, serving them the frontend static resources (HTML, CSS, JS), which enables the user to automatically connect to the WebSocket Server. From this point on, the WebSocket receives all Client interaction, and is responsible for reacting to the user's interaction with the frontend interface.

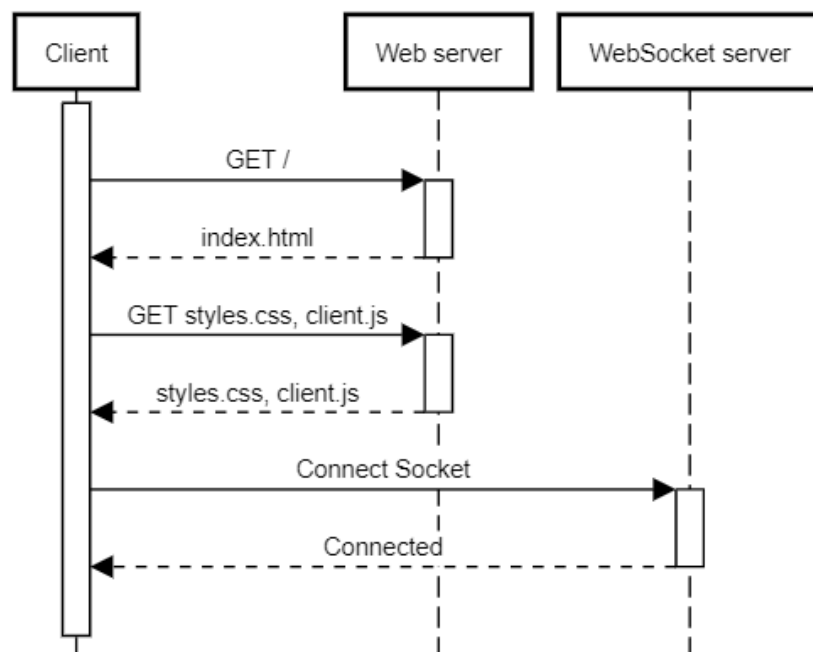


Figure 5.2: Sequence diagram: Client connecting

To fulfill the desired functionality of manipulating the Kubernetes Cluster, the Lobby WebSocket

Server imports and uses the `@kubernetes/client-node` Node package. Using this library, the Kubernetes configuration is gathered automatically, granting the WebSocket server full access to the Namespace within the Cluster. This is made possible by the library's *in-cluster configuration instantiation*, meaning that the configuration is loaded from the Cluster from which the code is run. Since this happens within a Pod in the Cluster, it all lines up.

The client-side UI (`index.html`, `styles.css`) and functionality (`client.js`) allows the user to trigger two different events: *Create Room* and *Join Room*, both accompanied by a room name, entered by the user.

Create Room

When the user clicks "Create Room", the currently entered room name is sent to the WebSocket Server in shape of a "Create Room" message. When the Server receives this message, it will immediately query the k8s API to do the three following things ([name] represents the room-name input by user):

- Create a Room Pod named "room-[name]"
- Create a Service pointing to the Room Pod named "svc-[name]"
- Patch the Ingress resource's paths configuration so that path `"/[name]"` redirects to the Service

This process successfully scales the system with an entirely new Room Pod, available for connections at path `"/[name]"`, ultimately resulting in the creation of resources described in section 5.1.4. When the Room Pod has been spun up, the user is automatically redirected to its respective path, seamlessly connecting user and room.

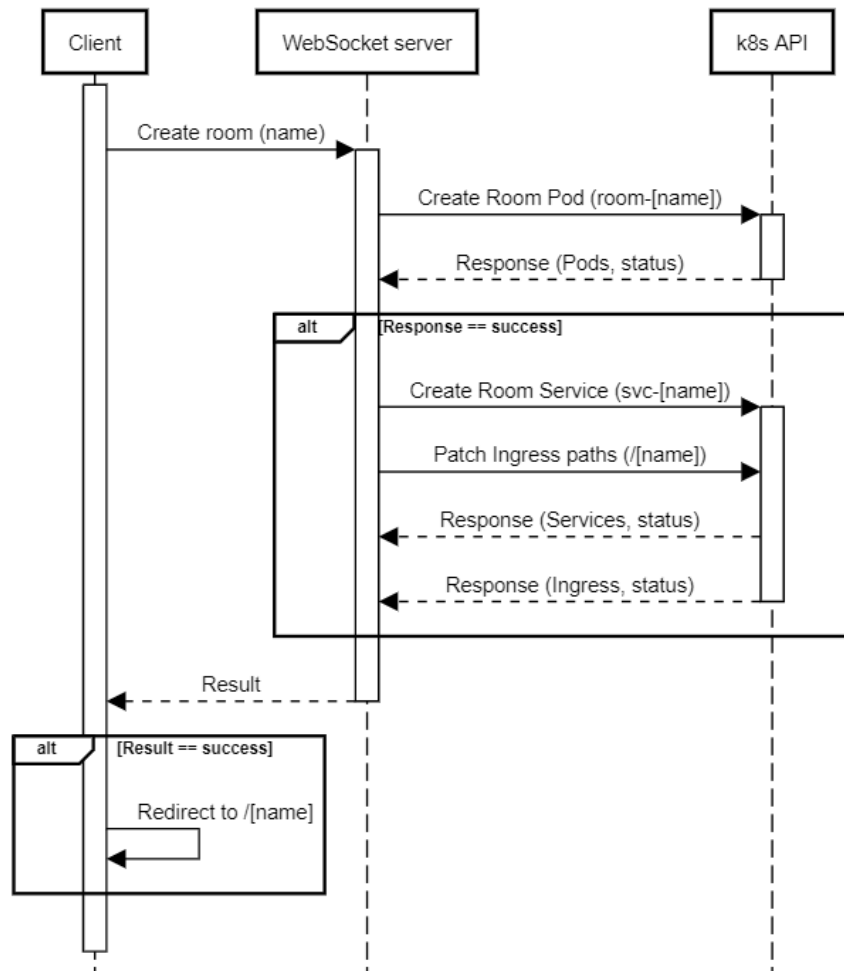


Figure 5.3: Sequence diagram: Create room

As figure 5.3 shows, the WebSocket Server attempts to create the Room Pod *before* creating its accompanying components. This is done so that if the Room Pod creation fails (typically if the room name is occupied), false Services and Ingress paths are not created. In this case, the WebSocket Server will also notify the Client of the error, informing the user that the room was not created.

Join Room

When the user clicks "Join Room", the currently entered room name is sent to the WebSocket Server in shape of a "Join Room" message. Upon receiving this message, the Server will check if a Room Pod matching the given room name is live by attempting a GET request to the corresponding path, the path of course being `/[name]`. If the request results in a successful response (i.e. a HTTP status code of 200-299), the Server notifies the Client of a successful request, and the user is redirected to the room. Alternatively, if the request results in an unsuccessful response (i.e. HTTP status outside previously mentioned range), the Client is notified of a failure, and the user is notified that the room does not exist.

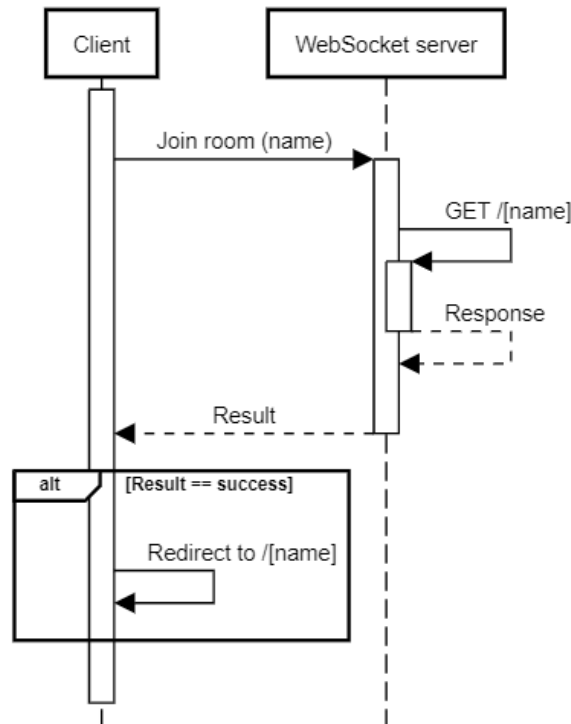


Figure 5.4: Sequence diagram: Join room

5.2.2 Room

A Room is responsible for handling the users within it, creating WebRTC connections between them, and orchestrating the general "room" environment. Similarly to the Lobby, it also provides the user an interface for utilizing its functionality, and this is likewise accomplished by running both a Web Server and a WebSocket Server simultaneously.

The Web Server acts identically to the one from the Lobby, so documenting it would be redundant. In short, it serves the user the static resources of the site (HTML, CSS, JS), after which the Client connects to the WebSocket Server. The WebSocket Server, on the other hand, is largely different in what it aims to achieve. It is designed to have predetermined reactions to different types of incoming messages. These message types are:

- Login
- Chat
- Canvas Update
- Offer
- Answer
- Candidate

Login

When the user first loads the page, they are presented with a login-screen, where they simply have to input a username, and then click "Login". On click, a message of type "Login" along with the entered username is sent to the Server. When the Server receives this, it first checks whether the name is already occupied, by checking it against its internal list of users and connections. If the name is available, the username is registered along with its corresponding WebSocket connection, after which a message is sent back to the Client, informing it that the login was a success, at which point the UI will transition from the login-screen to an in-room screen. If the username happens to be occupied, the Server instead notifies the Client of a failure, and the user gets a "Username taken" message.

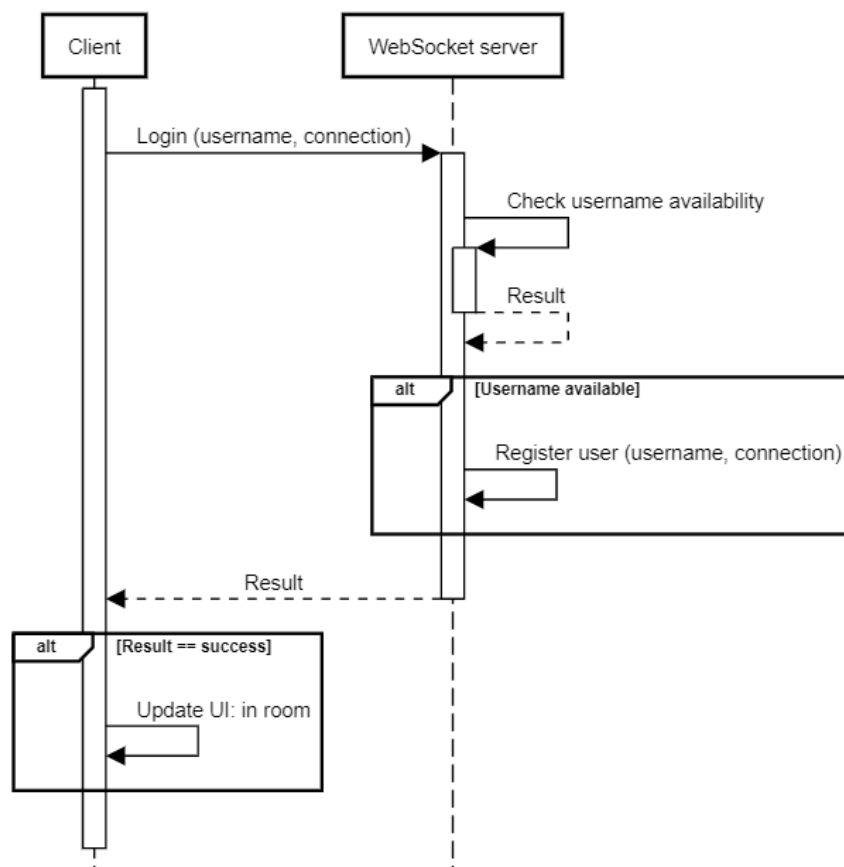


Figure 5.5: Sequence diagram: Login

Chat

The application includes a very simple chat system, allowing all users in a room to write and send text messages to one another. These messages are handled by the WebSocket Server, where it receives and forwards them accordingly. When the user inputs a message and sends it, the Server simply forwards that same message to every user in the room. When a Client receives a chat message, it is appended to the on-screen chat-window.

Canvas Update

Whenever the avatar canvas is altered (e.g. a user moves their avatar), a message of type "Canvas Update" is sent to the server, carrying information about the canvas change. When the Server receives such a message, it updates its internal state of the canvas to match the incoming information, and then forwards the message to all users in the room. Likewise, when a Client receives a Canvas Update, the Client's internal canvas state is updated to match. This way, the Server always has the final say of the state of the canvas, and all users will thus see the same, synchronized canvas of avatars.

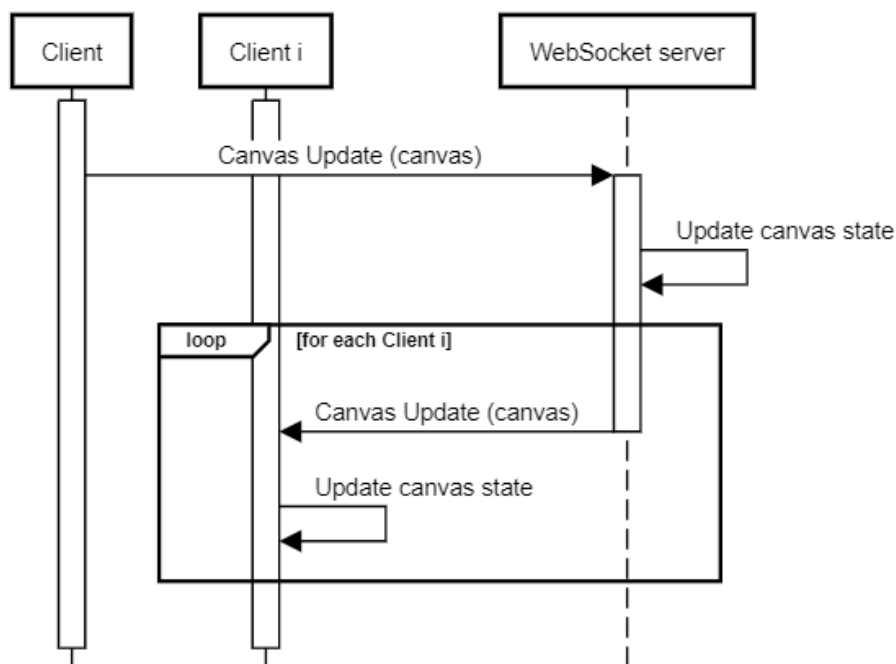


Figure 5.6: Sequence diagram: Canvas Update

Offer, Answer, Candidate

These messages all have to do with the process of establishing peer connections between Clients. When a Client has successfully "logged in" to a room, it will immediately send out "Offers" to every other user in the room. *Offers*, and their counterpart, *Answers*, are part of the WebRTC API, and are created from the context of a peer connection object (*RTCPeerConnection*). When sending Offers to users, a peer connection is created for each user.

After creation, an Offer is added to the peer connection which it is created from as its *Local Description*, after which it is sent to the WebSocket Server in a message of type "Offer" containing the following information: username of the offerer (sender), username of the answerer (recipient), and the Offer itself. When the Server receives an Offer, it simply forwards it to the corresponding answerer, the username of which contained in the message.

When a Client receives an Offer, it creates a new peer connection to reflect the connection

being established, and then adds the Offer as the *Remote Description* of the connection. It then creates an Answer from the newly made connection, and sends it back to the Server in a message of type "Answer" containing the same offerer and answerer usernames as the incoming Offer message, albeit with the Answer instead.

When a Client receives an Answer, it simply adds the Answer as the *Remote Description* of the corresponding peer connection, listed internally by the username of the user an Offer was initially sent to.

Lastly, the Server is responsible for distributing ICE Candidates to other users when it receives messages containing them. ICE Candidates are "found" as part of the Trickle ICE process which WebRTC utilizes to chart a network path between peers. As soon as a peer connection has had its Local- or Remote Description set, ICE Candidates will start "appearing" asynchronously, at which point the peer connection's candidate handler is responsible for handling them. This handler is set to send any ICE Candidate to the Server as a message of type "Candidate", along with the username of the sender. When the Server receives such a message, it simply forwards it to all other users in the room, excluding the sender. Finally, when a Client receives a Candidate, they are added to their corresponding peer connections. ICE candidates are "found" and provided by what are called STUN and TURN servers. While it is generally recommended to build your own STUN or TURN servers, such that you can rely on their availability, we opted to use one that was publicly available, the address of which being `stun:stun.l.google.com:19302`.

5.3 Frontend

In this section, the thoughts and intentions behind the development of the frontend user interface is described, along with documentation of any significant parts of the accompanying client-side software.

5.3.1 User interface

When developing the initial prototype for CoffeeBreak, its interface was deprioritized for the sake of functionality. This meant that necessary UI components were simply added to the working prototype without any further thought as to how they appeared or where they were placed. This allowed us to focus on getting a *functional* prototype in place, rather than a *pretty* one. Besides, we would have likely had to rework the UI every time a significant functionality changed during development, so this was ultimately the optimal approach.

When we felt that CoffeeBreak had reached a stable and functional enough state that it warranted a proper UI, all existing UI components which were tied to system functionality were compiled in a single model. This allowed us to envision how to construct the UI from them.

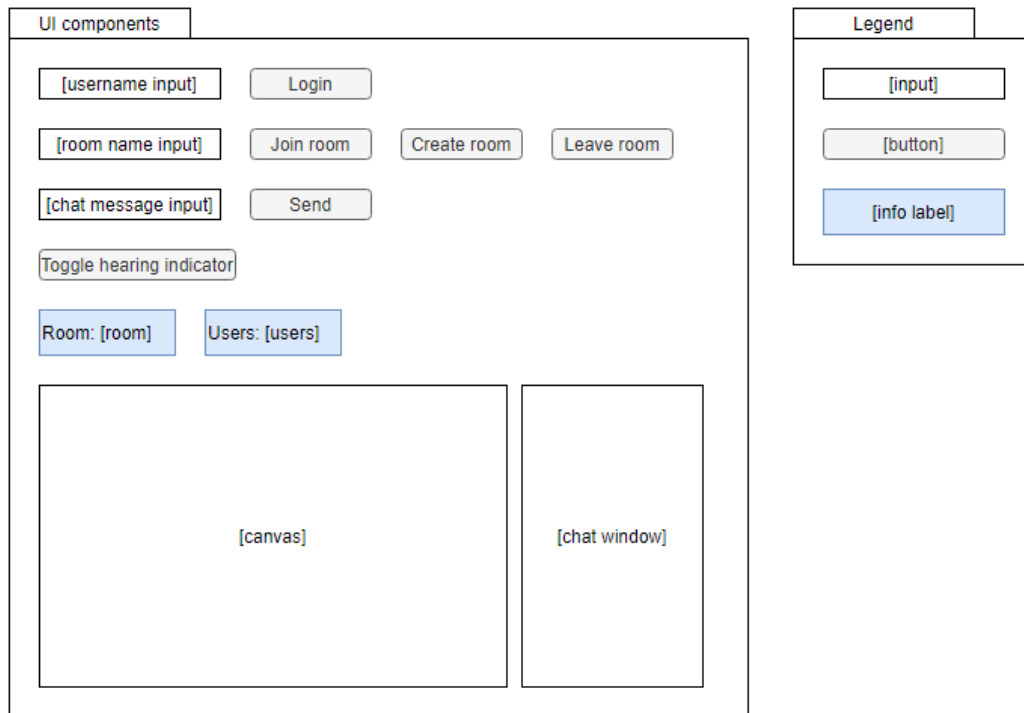


Figure 5.7: User interface component breakdown

The result consists of dividing the UI into three "views", these being

- A Login view
- A Join/Create room view
- A Room view

The intention behind this was to not overwhelm the user with all of the inputs at once, but rather introduce them as they are needed. This way, the user is linearly guided through logging in, joining/creating a room, and then being in the room. The division of the Login and Join/Create room views also ties into the structure of the Kubernetes Cluster, since each of the two types of Pods serves the user a different view, ultimately forcing it.

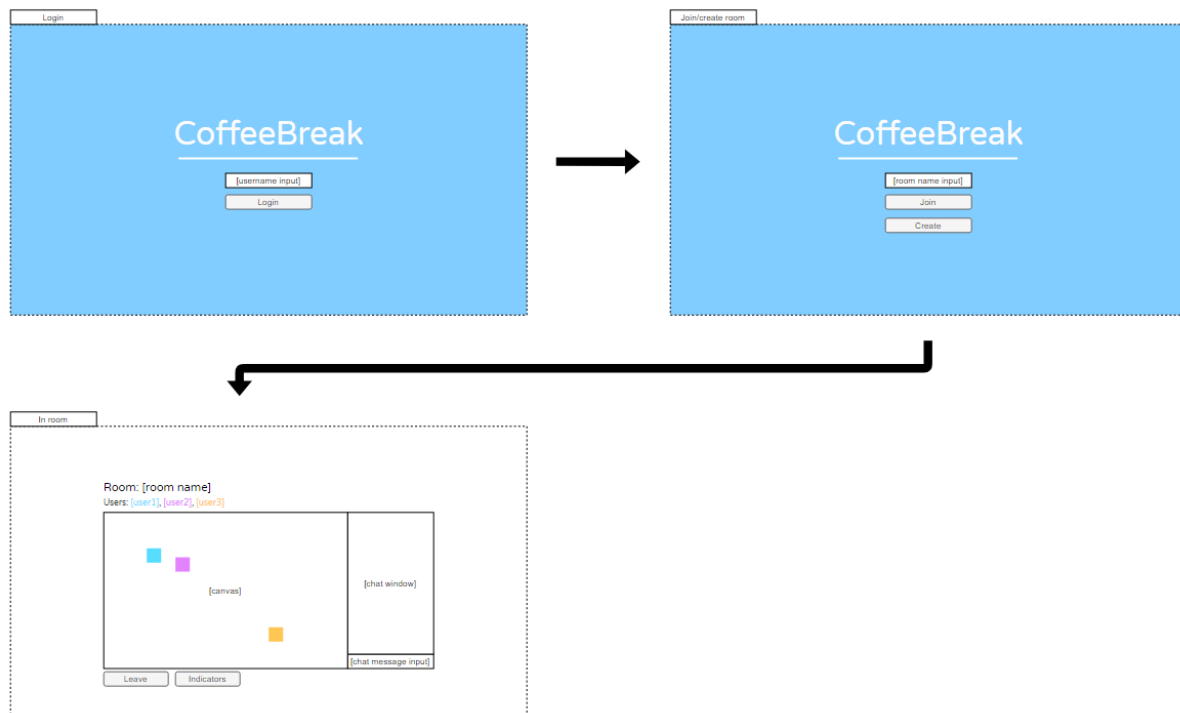


Figure 5.8: User interface: Mock-ups and flow

These are of course conceptual sketches, and not representations of the actual implementation. CoffeeBreak is a web application, and so the user interface is naturally built in HTML and CSS, with interaction between user and system handled by JavaScript. Images of the final interface implementation can be found in appendix A.6.

5.3.2 Client-side software

As we know, CoffeeBreak is a web application, and thus client-side code must be present in some shape or form. With JavaScript, any code which is to run on the client is served by the Web Server, along with HTML and CSS, and this means that the Client in theory has direct access to the source code of the site. This sparks considerations between what code is okay and safe to give to the Client, and what should be kept on the external servers. This does not apply to *all* of the source code, of course, since some is simply required to exist on the client-side. Ultimately, we decided to limit the client's access to managing rooms and such to predetermined "commands" which are sent to the Server, as described in section 5.2. In its current state, however, CoffeeBreak has a major security flaw, in that any user, in theory, can send custom messages to the Server without any form of authorization. This is obviously quite severe, and should be addressed in the immediate next iteration.

Canvas, avatars and soundscape

One major requirement called for the implementation of something resembling a 2D "canvas" where users could arrange their avatars, and the incoming sound from other users was based

hereof (described in 3.2). In CoffeeBreak, this functionality is achieved through the use of an HTML *canvas* element. Essentially, this element acts as a window where one can render whatever they like, and this serves the premise perfectly. Like with a generic "game-loop", the canvas renders the entire picture constantly, each time iterating through a local list of avatars, reading their x-y coordinates, colors and associated usernames, and using these for the render. The Client's local list of every avatar in the canvas is updated by the Server every time a change is made to it (see 5.2.2), keeping every user in sync.

With a system for handling and managing user avatars in place, it was fairly straight-forward to implement the functionality of varying sound levels of other users based on distance/orientation. For the sake of time, we opted to aim for a simple solution of volume scaling up and down, without inclusion of the previously proposed stereo mixing. To accomplish this, we needed a way to represent the distance between the Client's avatar, and every other avatar. Since the avatars exist in a 2D plane, and all have x-y coordinates associated to them, getting their distances was as simple as calculating the 2D vector between them, and using Pythagoras theorem on said vectors to get their lengths. For example, consider the figure below.

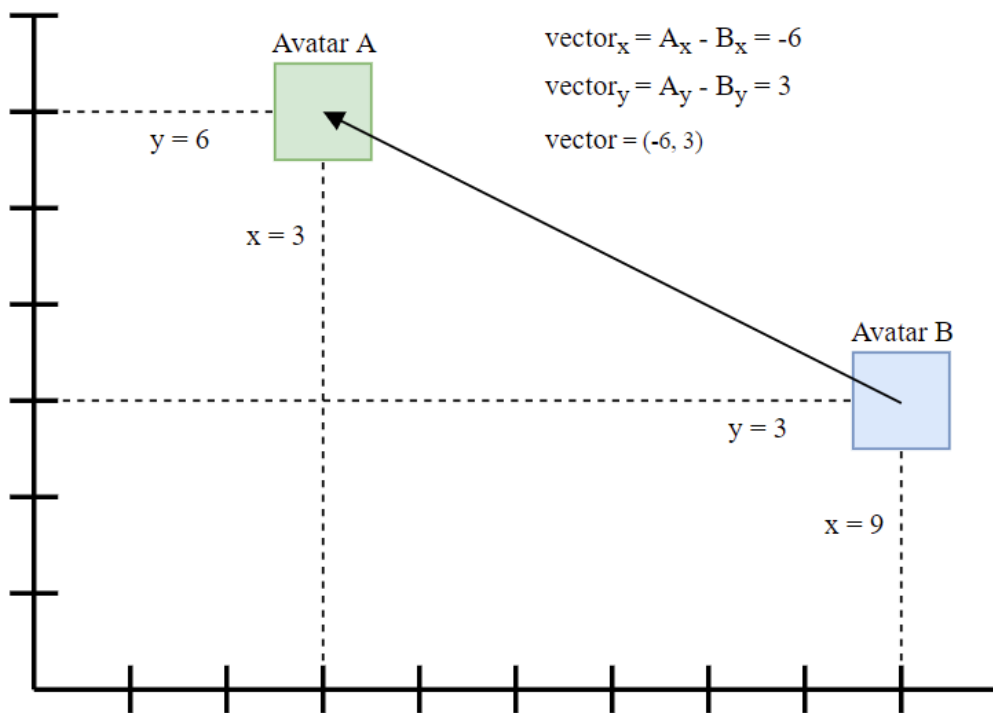


Figure 5.9: Calculating distance vector between two avatars

In figure 5.9, the vector from avatar B to avatar A is calculated by simply subtracting the coordinates of B from the coordinates of A. This results in the vector, illustrated as pointing from B to A. The length of this vector is then calculated using Pythagoras theorem, as such:

$$\text{Distance} = \sqrt{x^2 + y^2} = \sqrt{-6^2 + 3^2} = 6.708$$

In the code, this happens as such:

Listing 5.1 Function `getDistance()`

```
1 function getDistance(avatar1, avatar2) {
2     var vector = {
3         x: avatar1.x - avatar2.x,
4         y: avatar1.y - avatar2.y
5     };
6     // Pythagoras
7     var distance = Math.sqrt(Math.pow(vector.x, 2) + Math.pow(vector.y, 2));
8     return distance;
9 }
```

Once the distance between a Client's avatar and another has been acquired, its value must of course be used to change the volume level of the audio stream of the specific avatar's user. One approach might be to just gradually scale the volume from 100 to 0 within one specific range, but we opted to do this according to multiple predetermined "hearing ranges". These are:

- "Close": distance = 0 - 75, volume = 100%
- "Medium": distance = 200 - 400, volume = 10%
- "Far": distance = 400+, volume = 0%

If an avatar resides within any of the ranges mentioned above, their volume levels are set to the static value associated with the range. For instance, an avatar 279 units away will lie within the range of *Medium*, and such will have a volume level of 10%. However, avatars in the range of 75 to 200, or "in-between" will have their volume levels adjusted gradually between the 100% and 10%, as these are the upper and lower values of the surrounding ranges *Close* and *Medium*. In the application UI, the user has the functionality of toggling a visual indicator of these ranges, as seen in figure 5.10. This way, the user can position their avatar according to who they wish to listen to.

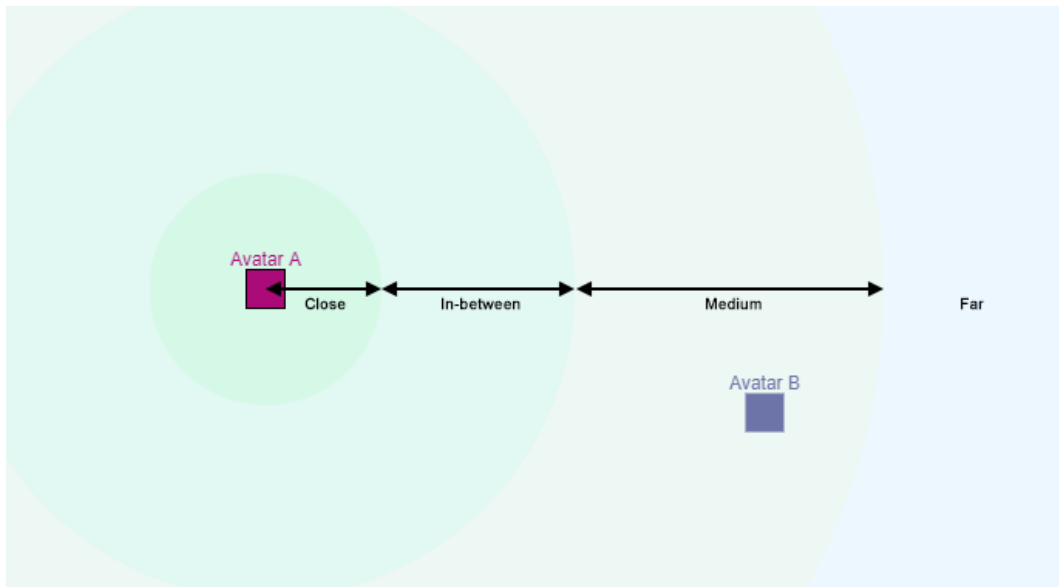


Figure 5.10: Visual representation of hearing ranges

To calculate the right gradual volume levels within the predefined range, the value must be mapped onto it, as such with the following function:

Listing 5.2 Function `convertRange()`

```

1 function convertRange(value, oldRange, newRange) {
2     return ((value - oldRange.min) * (newRange.max - newRange.min)) / (
        oldRange.max - oldRange.min) + newRange.min;
3 }

```

This function is used when an avatar requires its volume gradually adjusted, as such:

Listing 5.3 Function `updateVolumes()`

```

1 var volume;
2 if (distance >= far.threshold) { // "far" away
3     volume = far.volume;
4 } else if (distance >= medium.threshold) { // "medium" away
5     volume = medium.volume;
6 } else if (distance <= close.threshold) { // "close"
7     volume = close.volume;
8 } else { // in between
9     volume = convertRange(distance, {
10         min: close.threshold,
11         max: medium.threshold
12     }, {
13         min: close.volume,
14         max: medium.volume
15     });

```

```
16 }  
17 audioElement.volume = volume;
```

As evident in listing 5.3, once the appropriate volume level has been calculated, it is assigned to its appropriate audio element, effectively changing its volume for the Client.

5.4 Deployment

This section aims to briefly describe the different steps associated with deploying the Coffee-Break project's codebase. Given that the project is designed to be launched on a Kubernetes Cluster, setting it up is not as simple as running it on a single server. Therefore, a sequence of steps must be followed in order to set up the project successfully.

Do note that updating the Docker Images is optional for deployment, as the latest versions already exist on DockerHub. However, the corresponding Image must be republished to DockerHub, in order to reflect changes made to it in the codebase.

5.4.1 Updating the Container Images with Docker

Essentially, all of the project's code is designed to run in Containers, which are then eligible to be deployed to a Cluster. To build the project's two microservices *lobby* and *room* for CoffeeBreak, the following commands are executed from the root directory of the Node project:

Building a Docker Image for the **room** microservice and pushing it to DockerHub:

```
$ docker build ./room/. -t benjaminhck/coffeebreak-room  
...  
$ docker push benjaminhck/coffeebreak-room  
...
```

Building a Docker Image for the **lobby** microservice and pushing it to DockerHub:

```
$ docker build ./lobby/. -t benjaminhck/coffeebreak-lobby  
...  
$ docker push benjaminhck/coffeebreak-lobby  
...
```

Once these commands have been run, both Images will be publicly available from the corresponding DockerHub repositories. Pushing the Images to these specific repositories, under user *benjaminhck*, does require collaborator privileges.

5.4.2 Setting up the kubectl configuration

To be able to deploy and using any of the created Images using *kubectl*, a Kubernetes command-line tool, it must be configured to access the specific Cluster. Kubectl is a simple binary for

manipulating Kubernetes Clusters using the Kubernetes API. For this specific project, a Cluster residing on the University of Southern Denmark's own network was used to host the prototype. An example configuration to the Namespace *group2* on the SDU-Cluster, albeit with hidden certificates, is shown below:

Listing 5.4 Kubectl config example

```
1 apiVersion: v1
2 kind: Config
3 preferences: {}

5 clusters:
6 - cluster:
7     certificate-authority-data: OMITTED
8     server: https://10.123.252.228:8443
9     name: my-cluster

11 users:
12 - name: group2-user
13   user:
14     client-key-data: OMITTED
15     token: OMITTED

17 contexts:
18 - context:
19     cluster: my-cluster
20     namespace: group2
21     user: group2-user
22     name: group2
23 current-context: group2
```

Using the above configuration, kubectl will automatically and by default access the Cluster residing on SDU's network. However, it is possible to setup a kubectl configuration that can access multiple Clusters from the command line. This is very handy when working with multiple environments, e.g. accessing a local Minikube/Docker Desktop Cluster, and a Cluster hosted on a cloud service at the same time.

5.4.3 Deploying the project's code to a cluster

Once the kubectl tool is configured to access the Cluster, the only remaining step is to deploy the project's YAML-configuration file. This is done in the command-line from the root directory of the Node project, as shown below:

Updating the Kubernetes Cluster setup with the YAML-file:


```
$ kubectl apply -f deployment/deployment.yaml
...
ingress.networking.k8s.io/ing-coffeebreak created
pod/lobby created
service/svc-lobby created
```

The above command will spin up the Ingress, Service and Pod with the specifications referenced at pages 22, 23 and 24 respectively. After this, the status of each deployed Kubernetes resource can be checked by using the aforementioned tool *kubectl*.

An example of using *kubectl* to get Pod status::

```
$ kubectl -n group2 get pods
NAME      READY   STATUS    RESTARTS   AGE
lobby     1/1     Running   0           1m
```

The CoffeeBreak project should now be deployed to the Cluster and be ready for interaction.

6 Verification

This chapter aims to verify to which degree the initial requirements defined in section 3 were met through implementation and development. This is accomplished in a variety of ways. Where applicable, by simple design inspection, meaning that the code/system/implementation is closely inspected, to evaluate whether it achieves its intended purpose, or perhaps meets one or more requirements. Others, by measuring parameters of the system, to assess whether these are within their intended range.

6.1 Functional requirements

To verify the functional requirements of the system, as described in section 3, they are systematically evaluated, based on their suggested methods of evaluation. Ultimately, we seek to either concretely **confirm** or **deny** the fulfillment of each existing functional requirement, so that a new list of requirements can be set for a potential future iteration of CoffeeBreak.

6.1.1 Room requirements

| Room Requirements Overview | | | |
|----------------------------|--|---|---|
| FR1 | The system has rooms which encapsulate communication sessions and their participant users | M | ✓ |
| FR1.1 | Users can create rooms | M | ✓ |
| FR1.2 | Users can invite other users to their room | S | ✓ |
| FR1.2.1 | A room has a unique ID which can be given to invitees | M | ✓ |
| FR1.3 | Users can join rooms they are invited to | M | ✓ |
| FR1.3.1 | A room can be joined using its unique ID | M | ✓ |

Table 6.1: Room requirements

FR1: Room

The requirement about the software system having encapsulated rooms where users can communicate with each other has been the backbone of the project. Essentially, this is based on the physical definition of a room, where people can come together and interact. The virtual representation of a room is implemented as software deployed to a Kubernetes Pod, having a one-to-one ratio between rooms and Pods. This relationship is evident on figure 5.1. Based on this figure, and documentation of the room implementation in section 5.2.2, it is clear that FR1 is met.

FR1.1: Create Rooms

As seen in the previous section, the room implementation is done per the premise of Kubernetes Pods, which means that creating new rooms in the system involves setting up a new Kubernetes Pod. However, a Pod in the Kubernetes infrastructure will not work as a standalone resource, and must be supported by other resources to fully expose it to the outside of the Cluster. How the system dynamically creates these resources using the Kubernetes API is demonstrated in section 5.2.1. Based on this, along with manual testing from the perspective of a user, it is clear that FR1.1 is met.

FR1.2, FR1.2.1: Inviting to Rooms

The requirements of users being able to invite others to a room is fulfilled by the innate way room identities and routing is managed. As described in section 5.2.1, the Ingress object is patched to open a path to the room Pod matching a user-entered value, this being the name of the room. For example, if a user inputs '*bachelor*' and clicks the Create Room button, the necessary Kubernetes resources will be set up within the Cluster, after which the room will be available on the '/*bachelor*/' endpoint. In the case of CoffeeBreak's current deployment, this would make the invitation URL for room *bachelor* as follows: <https://group2.sempro0.uvm.sdu.dk/bachelor/>.

While there isn't a specific frontend implementation for inviting users besides either manually copying and distributing the URL, or simply the room name, documentation of the implementation, along with manual testing from the perspective of a user, makes it clear that FR1.2 and FR1.2.1 are met.

FR1.3, FR1.3.1: Joining Rooms

Based on the verification of FR1.2 and FR1.2.1, FR1.3 and FR1.3.1 are nearly confirmed as met by default, since we have already established that a room can be joined by a user given that the user knows its name. As described in section 5.2.1, a user can simply input the name of the room into the input field of the user interface, and then click "Join Room" to be redirected to the room's available endpoint. Based on this, along with manual testing from the perspective of a user, it is clear that FR1.3 and FR1.3.1 are met.

6.1.2 Personal configuration requirements

| Personal Configuration Requirements Overview | | | |
|--|--|---|---|
| FR2 | Users can manage their identity | C | ✓ |
| FR2.1 | Users can assign and change their nickname | S | ✓ |
| FR3 | Users can manage their sound settings | C | ✗ |
| FR3.1 | Users can mute and unmute their microphone | C | ✗ |
| FR3.2 | Users can adjust incoming sound | C | ✗ |
| FR3.2.1 | Users can adjust level of all incoming sound gradually | C | ✗ |
| FR3.2.2 | Users can mute individual participants in the room | C | ✗ |

Table 6.2: Personal configuration requirements

FR2, FR2.1: Managing identity

As described in section 5.2.2, a user is forced to enter a chosen username upon entering a room. This value is then used, both by other users and the room itself, to identify the specific user and distinguish it from others. This supports that users can assign their nicknames, however not necessarily that they can change it. While it might not be entirely clear to the user, an argument can be made that a user's nickname *can* be changed, by the user simply re-joining the room and inputting a different username. Based on this, along with manual testing from the perspective of a user, it is clear that FR2 and FR2.1 are met.

FR3, FR3.1, FR3.2, FR3.2.1, FR3.2.2: Managing sound

These requirements all have to do with adjusting volume levels for incoming sound. While CoffeeBreak does not contain in-app functionality for accomplishing any sound-management in its current state, an argument can be made that "blanket-adjusting" all incoming volume can be achieved by the user by simply adjusting the sound settings of their operating system or audio peripherals, however this negates the purpose of including these requirements in the first place. Based on this, it is clear that FR3, FR3.1, FR3.2, FR3.2.1 and FR3.2.2 are not met.

6.1.3 Soundscape requirements

| Soundscape Requirements Overview | | | |
|----------------------------------|---|---|---|
| FR4 | Users can hear a simulated soundscape based on location of self and other users | M | ✓ |
| FR4.1 | Volume of speakers varies based on distance from user (farther = lower) | M | ✓ |
| FR4.2 | Sound of speakers play for user in their relative direction via stereo | C | ✗ |
| FR5 | User has an avatar in a 2D representation of the room and can manage it | M | ✓ |
| FR5.1 | User can place and move their avatar within the 2D plane | M | ✓ |

Table 6.3: Soundscape requirements

6.1.4 FR4, FR4.1, FR4.2: Simulated soundscape

The major selling point and gimmick of CoffeeBreak is to present the users with a simulated soundscape. For the sake of managing time, it was decided to place focus on implementing FR4.1, as it was the minimal viable solution to satisfy the aforementioned selling point, and then attempt an implementation of FR4.2 if time allowed it. In the current state of CoffeeBreak, there is no stereo-mixing in relation to the simulated soundscape, as suggested by FR4.2. Instead, sound levels between users are simulated by the two-dimensional distance between their avatars, as described in section 5.3.2. Based on this, along with manual testing from the perspective of a user, it is clear that FR4 and FR4.1 are met, while FR4.2 is not.

6.1.5 FR5, FR5.1: Avatars

As described in 5.3.2, every user is automatically assigned a visual avatar in a two-dimensional canvas within the user interface. Furthermore, a user can drag-and-drop their avatar to reposition it, actively altering their simulated soundscape. This enables users to seamlessly "move around" in a room, and even split up into multiple discussion groups where people can speak to a select few, which has been one of the primary goals of the project. Based on this, along with manual testing from the perspective of a user, it is clear that FR5 and FR5.1 are met.

6.2 Nonfunctional requirements

As defined in section 3, CoffeeBreak was developed with three nonfunctional requirements in mind, these being:

| Nonfunctional Requirements Overview | |
|-------------------------------------|--|
| NFR1 | Capable of 50 concurrent users in a room |
| NFR2 | Scalable up 100 rooms |
| NFR3 | Low latency for voice communication |

Table 6.4: Non-functional requirements

While we will attempt to assess the degree of fulfillment to which these requirements were met, it is worth to note that these requirements were initially set without our newfound knowledge in subjects such as JavaScript, WebRTC or Kubernetes.

6.2.1 NFR1: User capacity

To verify this requirement, the simplest method would be to gather 50 people, and have them connect to a single room at the same time. However, due to limitations of the project, this has not been possible. Instead, we will attempt to analyze the current iteration of CoffeeBreak and its implemented technologies, in order to find potential limitations in regards to user capacity.

Since CoffeeBreak uses a very simple WebRTC implementation, creating peer connections between every Client in any given room, the amount of network bandwidth required rises with the amount of Clients. This is because every Client has to stream their media to every other Client, while also receiving their streams at the same time. This essentially means that any given Client has to stream and receive $N - 1$ times, where N is the number of current Clients. This will eventually lead to one of two issues: 1. the Client runs out of available bandwidth, or 2. the Client runs out of available processing power to process the incoming and outgoing streams.

To assess the severity of this situation, we have attempted to measure network activity when connecting with a high number of Clients. The results are as follows:

| N | Network activity |
|---|------------------|
| 2 | 10 Kbps |
| 3 | 52 Kbps |
| 4 | 95 Kbps |
| 5 | 173 Kbps |
| 6 | 220 Kbps |
| 7 | 358 Kbps |

Table 6.5: Network test results

Plotted on a graph, it is evident that the increase in network activity does not increase linearly

with the number of Clients in the room.

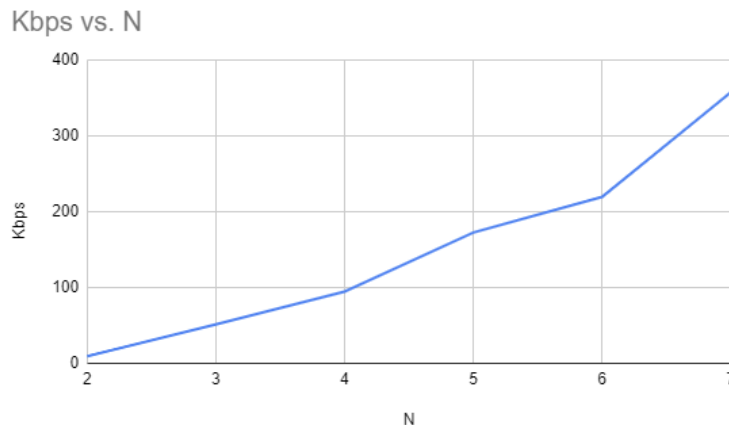


Figure 6.1: Network test results, graph

If we for curiosity's sake calculate the tendency line of the above data set, we get the function of:

$$f(x) = 66.34x - 147.21$$

This function represents the ratio between the number of Clients and required network bandwidth. If we input the desired number of 50 Clients into the function we get:

$$f(50) = 66.34 * 50 - 147.21 = 3169.79$$

This suggests that roughly 3 Mbps of bandwidth is required for each Client to be able to use CoffeeBreak in a room of 50 participants. This is a rough estimation, and its trustworthiness is debatable, considering the conditions of the test. Furthermore, the aforementioned risk of lacking processing power would most likely become an evident issue at this point. Essentially, CoffeeBreak should have been developed around a WebRTC technology designed to handle many Clients at once, such as what's known as an SFU (Selective Forwarding Unit), where Clients only have to stream their media once.

So, to conclude on NFR1, CoffeeBreak, in its current iteration, does not contain the capabilities of effectively supporting upwards of 50 Clients at once. However, testing has shown that smaller meetings of sizes 10 and lower run with no issues whatsoever.

6.2.2 NFR2: Scalability

In theory, CoffeeBreak is able to have 100 rooms at one time, this of course meaning that 100 Pods are running simultaneously. As the Kubernetes documentation states, Kubernetes is designed to be able to handle up to 100 Pods on one Node, and up to 5000 Nodes per Cluster [24], meaning that CoffeeBreak could potentially scale to an incredibly large number of rooms. This, however, does not consider CoffeeBreak's innate measure of scalability. As [23] states, simply adding more instances of a given application never results in linearly growing throughput.

In order to get this measure, the Universal Scalability Law comes into play, as it seeks to measure just how scalable a system really is, by depicting with which rate the throughput of the system grows (and potentially falls) as it is scaled. As described in section 2, the USL requires two main parameters. To reiterate, these are:

- σ : Unparallelizable workload of the system
- κ : Crosstalk penalty of the system

In the case of CoffeeBreak, κ is equal to 0, since each instance of a room has zero interaction with anything but itself, and so does not rely on a central server for data or similar. However, σ is a measure which must be gathered through load testing of the system, and will represent the fraction of which is unparallelizable, e.g. 0.1 for 10%. Unfortunately, due to time limitations, no such tests were conducted on CoffeeBreak, so this value was never obtained. If we, for the sake of illustrating the tool, imagine that σ was measured to be 0.1, this would then simply be input into the formula, resulting in the following function and graph:

$$C(p) = \frac{p}{1 + 0.1(p - 1) + 0p(p - 1)}$$

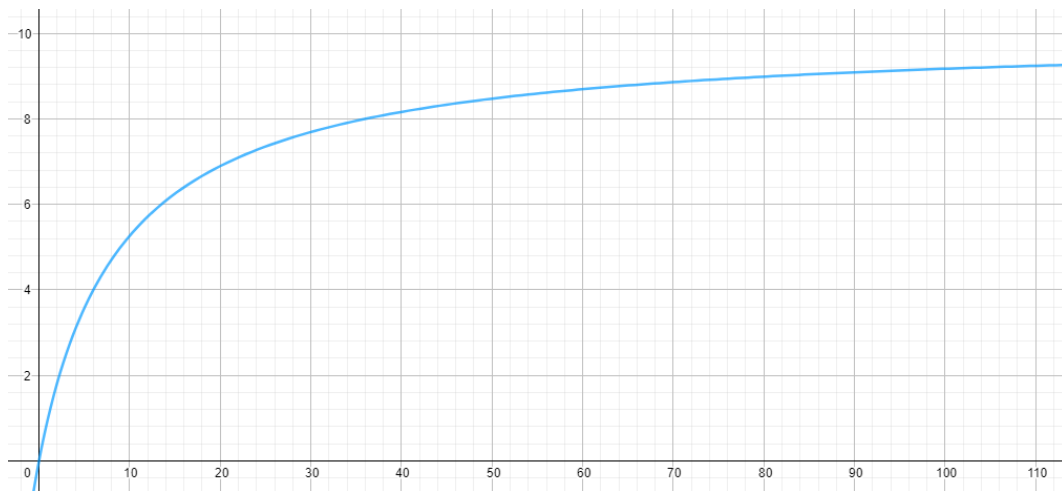


Figure 6.2: Simulated Universal Scalability Law, graph

If we then input 100, representing 100 room Pods, as p into the function, we get a throughput value of 9.17, as such:

$$C(100) = \frac{100}{1 + 0.1(100 - 1) + 0 * 100(100 - 1)} = 9.17$$

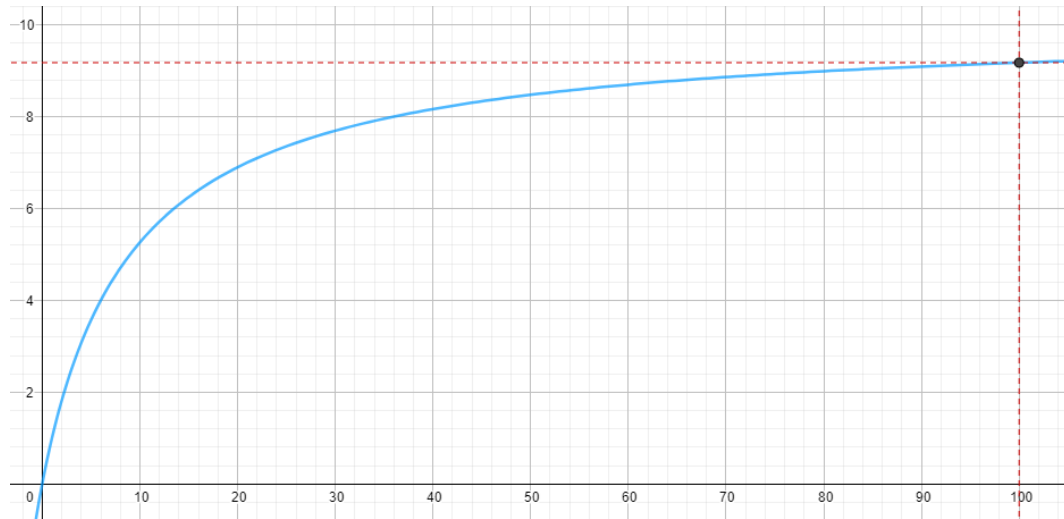


Figure 6.3: Simulated Universal Scalability Law, graph, 100 rooms

The value of 9.17 essentially has no value in this context, as it relates to a non-existent data set of throughput measurements, however the placement of the resulting point does provide some information. The fact that it exists slightly before the curve enters a close to horizontal orientation, tells us that, in a case where σ equals 0.1, CoffeeBreak could scale to 100 rooms before throughput would fully stagnate. Once again, this is based on pure speculation, and to some extent, guesswork, as no actual data has been used for this estimation.

To conclude on NFR2, fulfillment of the requirement is deemed inconclusive, however verified to the extent allowed by the project deadline.

6.2.3 NFR3: Latency

This requirement was initially set before the RTC approach was chosen, and so does not fully apply to the current iteration of CoffeeBreak. This is because, as previously stated, WebRTC is implemented through a peer-to-peer approach, meaning networking between Clients is not handled by any deployed unit of software, but rather the Clients themselves. This means that latency between a Client speaking into a microphone and other Clients hearing the message depends entirely on the quality and speed of the involved Clients' network connections. However, this does not mean that we can simply rely on Clients to have "good enough" connections. Rather, we should verify whether the chosen implementation of peer-to-peer is viable given the user-base and their internet connections, and if the resulting latency lies within an acceptable range.

Obtaining latency values between peers is fairly straight forward, as the WebRTC API provides the tools necessary out of the box. The tool in mention is the method of *getStats()*, which will return various statistics surrounding any given peer connection (*RTCPeerConnection*) object. These include a pre-calculated field *roundTripTime*, which represents the average amount of time in seconds it takes from when RTP packets are sent, when they are received by the re-

ipient, to when responses are received by the sender - a "round trip" is made. Since this time measurement is both "to" and "from", if we assume that both trips take an equal amount of time, we can divide it by two to get the true end-to-end latency. By accessing CoffeeBreak from different geographical locations using a VPN, the following data was obtained:

| Peer 1 | Peer 2 | roundTripTime | Latency |
|------------|---------------|---------------|---------|
| DK, Odense | SE, Stockholm | 17ms | 9ms |
| DK, Odense | DE, Frankfurt | 43ms | 22ms |
| DK, Odense | NL, Amsterdam | 38ms | 19ms |
| DK, Odense | US, Atlanta | ~305ms | ~153ms |
| DK, Odense | CA, Montreal | ~300ms | ~150ms |

Table 6.6: Latency test results

Data in table 6.6 preceded by a ~ symbol was gathered using a fairly unstable VPN (latency would not stabilize), and so these are rough averages. The most important note to make here, is that the latencies do not exceed what one would usually expect when connecting across such geographical distances. This is vital, since latency as a result of physical distance is an unavoidable constraint, which no networking approach can counteract, and so we are able to conclude that the approach of CoffeeBreak is optimal in this context.

With this data now gathered, we need to consider how to evaluate "low latency" for real-time voice communication. A 2003 report by the Telecommunication Standardization Sector of ITU recommends that latency does not exceed 400ms, and that anything lower is sufficient for what they describe as "general network planning" [25]. Furthermore, they present a tool called the *E-model*, which can be used to estimate the effects of latency on conversational speech with latencies of 500ms and lower.

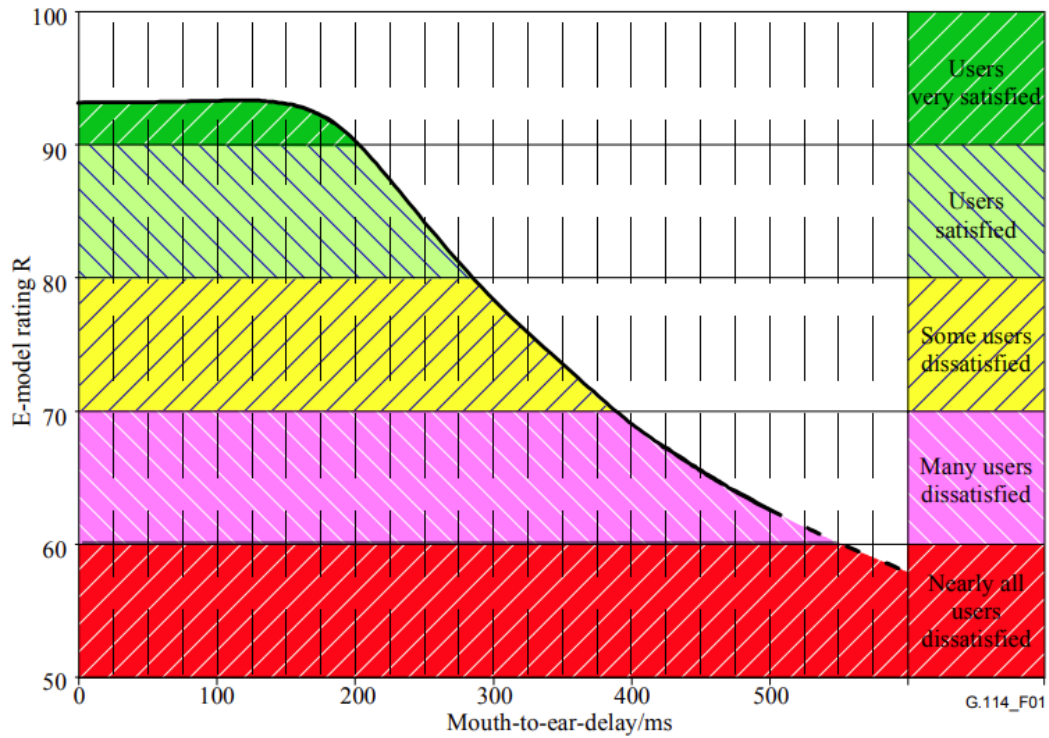


Figure 6.4: E-model, Telecommunication Standardization Sector of ITU [25]

If we take the row from table 6.6 with the highest measured latency, 153ms, and plot it into the E-model, we get a rating of around 93, which belongs to the interval of "Users very satisfied".

To conclude on NFR3, we feel that the requirement is very well met and represented by the gathered data. The conducted latency test, along with general latency guidelines provided from external sources, show that CoffeeBreak has optimal end-to-end latency for voice communication.

7 Conclusion

In this final chapter, we aim to conclude on the project, considering its results and findings, and comparing these to the goals initially set for the project.

7.1 System requirements

In the early stages of the project, a thorough requirements specification for how we imagined CoffeeBreak to operate was set, and this has been the cornerstone of development, both in terms of implementation, but also for researching of subjects. In section 6, we verified to which degree our efforts have served the requirements specification, and it was found that:

- **All** functional requirements prioritized as "must-have" were met (9/9)
- **All** functional requirements prioritized as "should have" were met (2/2)
- **Very few** functional requirements prioritized as "could have" were met (1/7)

Above metrics tells us two things: Firstly, that the method of prioritization used, and the actual prioritization done, were very effective for focusing work on important tasks, while setting less important tasks aside for the time being. Secondly, that CoffeeBreak can easily be considered to be in a presentable state of "minimal viable product" or "proof of concept".

7.1.1 Summary of features of CoffeeBreak

In this report, we have documented and described the following features:

- Peer-to-peer voice communication in real-time
- Simulated soundscape in real-time
- Room management system
 - Unique rooms isolating users within it
 - Dynamic on-demand scaling of room services
- Two-dimensional room canvas
 - User-interactive avatar system

Considering all of the above, we can conclude that the most vital and important functionalities of the imagined CoffeeBreak system have been implemented, providing an excellent foundation for further iterations of development.

7.2 Project goals

The goals of the project were to learn about the technologies and networking approaches required to develop a system capable of emulating how multiple conversations can be had in the real world ,at the same time, within the same room, so that we would later be able to create a prototype of such system.

The group sought to explore and acquire knowledge in new areas, to later be able to apply them. These areas included methods of transmitting voice in well-established applications, web-application development, and scalable technologies, as these seem to be increasingly more common and thus an important part of the industry.

Now having dealt with most of the intricacies of CoffeeBreak we, as a group, think we have met the goals we initially placed for the project.

8 Reflections

In this chapter, we consider CoffeeBreak in its current state, with the goal of reflecting on its flaws and shortcomings, while attempting to suggest future steps to correct them, should development of a next iteration ever take place.

8.1 Product

8.1.1 Deployment

As the system is currently configured, in order for the deployment to work as intended, a Kubernetes Namespace and Service Account must be present on the Cluster which CoffeeBreak is to be deployed on. In short, CoffeeBreak is currently configured to specifically deploy to the Kubernetes Cluster made available for the project. A way of future-proofing this requirement could be to simply add the creation of said Namespace and Service Account to the deployment configuration file, so that these mirror the aforementioned circumstances.

8.1.2 Room management

Since the Kubernetes API is only accessible from a Lobby Pod, we thought of maybe creating a new type of Pod and Service within the Cluster, this being one that is not exposed through the Ingress, but rather connected to, internally from the other Pods. This Pod would then contain all necessary functionality for interacting with the Kubernetes API. This way, when a Lobby Pod needs to create a new room, the information is instead sent to the API Pod, e.g. via HTTP request, which in turn would then create the new room and return a status response. This would also ensure that users are not directly connected to a Pod with access to the API, which of course is a security concern.

Due to time constraints, functionality for automatically removing room Pods was not implemented, however possible solutions for this were considered. Since room Pods utilize Web-Sockets, they would be able to tell how many Clients, if any, are currently connected. Having a period of time, e.g. five minutes, where no active connections are present, could then indicate that the room should no longer exist. In combination with aforementioned proposal of implementing an API Pod, each room would have the ability to send a request to it, asking to have themselves removed.

8.1.3 Networking approach

With the current implementation of the system, Clients connect directly to each other (peer-to-peer) for voice communication, and the pros and cons of this approach are described in section 6.2.1.

An alternative approach could have been to make the Clients transmit their audio streams to a central media server, and have it manage and forward streams to intended recipients, with the intent of reducing Client network and audio processing load. This concept is already broadly explored and implemented through existing solutions, and two major categorizations exist: **SFU** (Selective Forwarding Unit) and **MCU** (Multipoint Control Unit). These concepts both revolve around the idea of a media server handling and forwarding streams, however they differ slightly in approach.

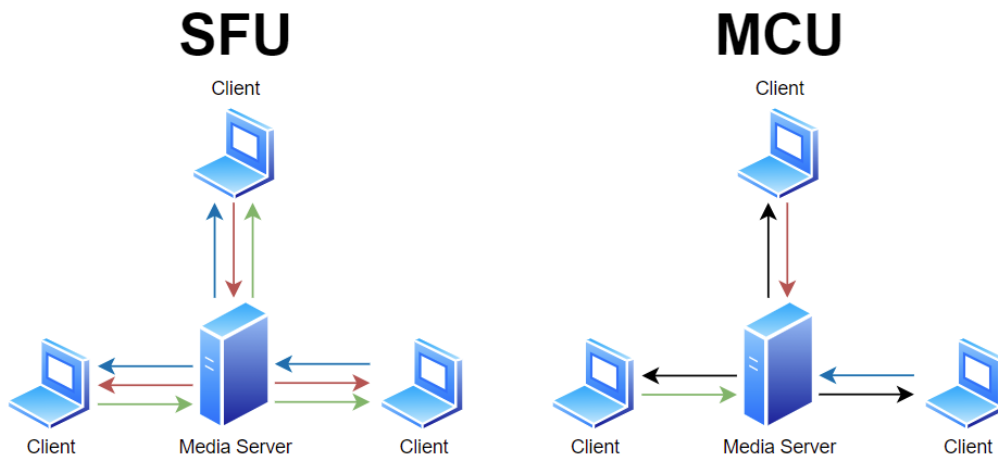


Figure 8.1: SFU vs. MCU

As figure 8.1 illustrates, an SFU receives all outgoing streams, and forwards them once per every receiving Client. The benefit of this approach is that Clients only have to send their outgoing stream once, instead of once per peer, as with the peer-to-peer approach, reducing Client load. An MCU on the other hand, while also receiving all outgoing streams, mixes them into a single stream before forwarding it to every Client, further reducing Client load. Either of these approaches puts more load on the server while reducing load for the Client, which was found to be necessary in section 6.2.1 in order to meet the requirement of large calls.

During development, an idea was discussed, which was to incorporate both peer-to-peer and media server approaches, potentially dependent on number of Clients in a room. When Client count is below a certain threshold, connect them through peer-to-peer, however transition their connections to the server-based approach if the threshold is surpassed. This way, the server would not experience much increase in load from having many smaller rooms when compared to having fewer larger rooms, as the networking in small rooms would be entirely Client-side.

A future iteration of CoffeeBreak would require an implementation of a media server based networking approach, for it to ever be capable of supporting upwards of 50 Clients in a call. The most likely choice is that of an SFU, as it allows mixing of individual audio streams to happen on the Client, which ties into the set requirements regarding this functionality.

8.2 Process

As described in section 1.4, we managed the work load of the project using Scrum, and perhaps more specifically through several Scrum-boards, providing an overview of every sprint. While this approach served its purpose, and we are satisfied with its outcome, there are some improvements to be made in retrospect, which we believe could have improved the resulting project quality.

The importance of time management was partly neglected, as no official schedule was constructed against the aforementioned work load, meaning time had a habit of "slipping" during the work on the project. Ultimately, this lead to serious crunch-time towards the end of the project. As a group of three, internal communication is high, and so we deemed that a schedule was not a necessary time investment. However, with three other semester courses with numerous deadlines of their own, on top of Covid-19 restrictions undoubtedly having a strong negative effect, the project would have definitely benefited from being managed by a time schedule, such as the well-known Gantt-diagram, preferably one per each conducted sprint.

Literature

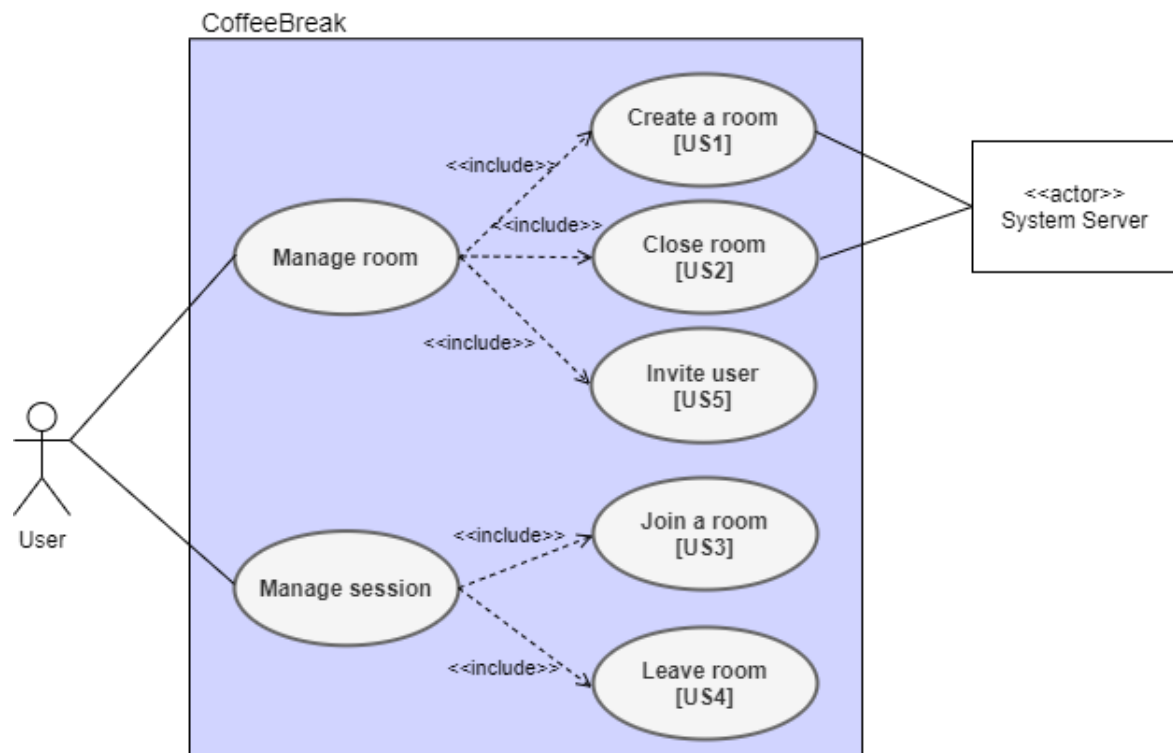
- [1] Wikipedia contributors. *Docker (software)* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 27-May-2021]. 2021. URL: [https://en.wikipedia.org/w/index.php?title=Docker_\(software\)&oldid=1019840030](https://en.wikipedia.org/w/index.php?title=Docker_(software)&oldid=1019840030).
- [2] Wikipedia contributors. *Kubernetes* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 27-May-2021]. 2021. URL: <https://en.wikipedia.org/w/index.php?title=Kubernetes&oldid=1024839217>.
- [3] Wikipedia contributors. *Node.js* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 1-June-2021]. 2021. URL: <https://en.wikipedia.org/w/index.php?title=Node.js&oldid=1025480714>.
- [4] Wikipedia contributors. *HTML* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 27-May-2021]. 2021. URL: <https://en.wikipedia.org/w/index.php?title=HTML&oldid=1016919858>.
- [5] Wikipedia contributors. *CSS* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 27-May-2021]. 2021. URL: <https://en.wikipedia.org/w/index.php?title=CSS&oldid=1024661292>.
- [6] Wikipedia contributors. *JavaScript* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 27-May-2021]. 2021. URL: <https://en.wikipedia.org/w/index.php?title=JavaScript&oldid=1024972090>.
- [7] Wikipedia contributors. *WebRTC* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 30-May-2021]. 2021. URL: <https://en.wikipedia.org/w/index.php?title=WebRTC&oldid=1024323804>.
- [8] Mansoor Qbal. "Zoom Revenue and Usage Statistics (2021)". In: (2021). URL: <https://www.businessofapps.com/data/zoom-statistics/>.
- [9] Manish Jain Konstant Infosolutions. *Zoom Video Conferencing App: Features, Tech Stack, Monetization, Cost*. 2021. URL: <https://www.konstantinfo.com/blog/zoom-video-conferencing-app/>.
- [10] *Guild Resource - Discord*. [Online; accessed 30-May-2021]. Discord. URL: <https://discord.com/developers/docs/resources/guild>.
- [11] Jozsef Vass. *How Discord Handles Two and Half Million Concurrent Voice Users using WebRTC*. Discord. 2018. URL: <https://blog.discord.com/how-discord-handles-two-and-half-million-concurrent-voice-users-using-webrtc-ce01c3187429>.

- [12] The Sherweb Team. "7 things you should know about Microsoft Teams". In: (2021). URL: <https://www.sherweb.com/blog/office-365/microsoft-teams/>.
- [13] Alex Tsukernik. *It's Here! Monitor Microsoft Teams Audio Video Conferencing*. Exoprise, 2019. URL: <https://www.exoprise.com/2019/09/10/monitor-microsoft-teams-audio-video-conferencing/>.
- [14] Tom Morgan. *Under the Hood of the Microsoft Teams Desktop Application*. 2017. URL: <https://blog.thoughtstuff.co.uk/2017/04/under-the-hood-of-the-microsoft-teams-desktop-application/>.
- [15] Salvatore Loreto and Simon Pietro Romano. *Real-Time Communication with WebRTC*. English. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly, 2014.
- [16] Tsahi Levent-Levi. *10 Massive Applications Using WebRTC*. 2017. URL: <https://bloggeek.me/massive-applications-using-webrtc>. (accessed: 12.03.2021).
- [17] Chandra Rajasekharaiah. *Cloud-Based Microservices: Techniques, Challenges, and Solutions*. English. 1. 2021. Berkeley, CA: Apress, 2021;2020;
- [18] IBM Cloud Education. *Containerization Explained*. IBM. 2019. URL: <https://www.ibm.com/cloud/learn/containerization>.
- [19] Roman Bessonov. *Microservices and containerisation: four things every IT manager needs to know*. 2018. URL: <https://nordcloud.com/microservices-and-containerisation-four-things-to-know>. (accessed: 12.03.2021).
- [20] Romana Gnatyk. "Microservices vs Monolith: which architecture is the best choice for your business?" In: (2018). URL: <https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/>.
- [21] Martin Kleppmann. *Designing data-intensive applications: the big ideas behind reliable, scalable, and maintainable systems*. First edition, second printing. O'Reilly Media, 2018. ISBN: 9781449373320,1449373321.
- [22] Baron Schwartz. *Practical Scalability Analysis With The Universal Scalability Law*. Tech. rep. VividCortex, 2015.
- [23] USENIX. *LISA17 - Scalability Is Quantifiable: The Universal Scalability Law*. YouTube. 2017. URL: <https://www.youtube.com/watch?v=IZU6RK0oazM>.
- [24] The Kubernetes Authors. *Considerations for large clusters*. [Online; accessed 29-May-2021]. 2021. URL: <https://kubernetes.io/docs/setup/best-practices/cluster-large/>.
- [25] Telecommunication Standardization Sector of ITU. *One-way transmission time*. Tech. rep. ITU-T, 2003.

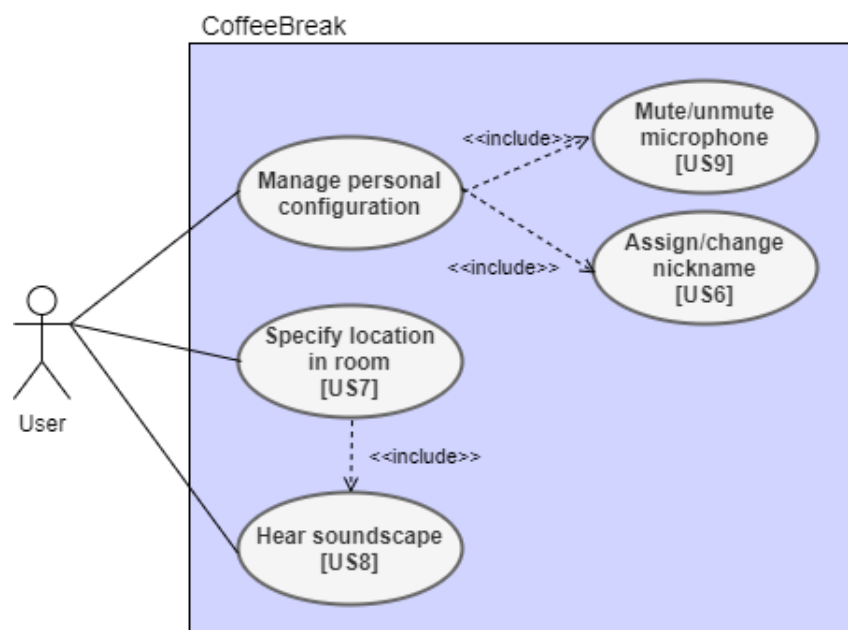
A Appendix

A.1 Use Case diagrams

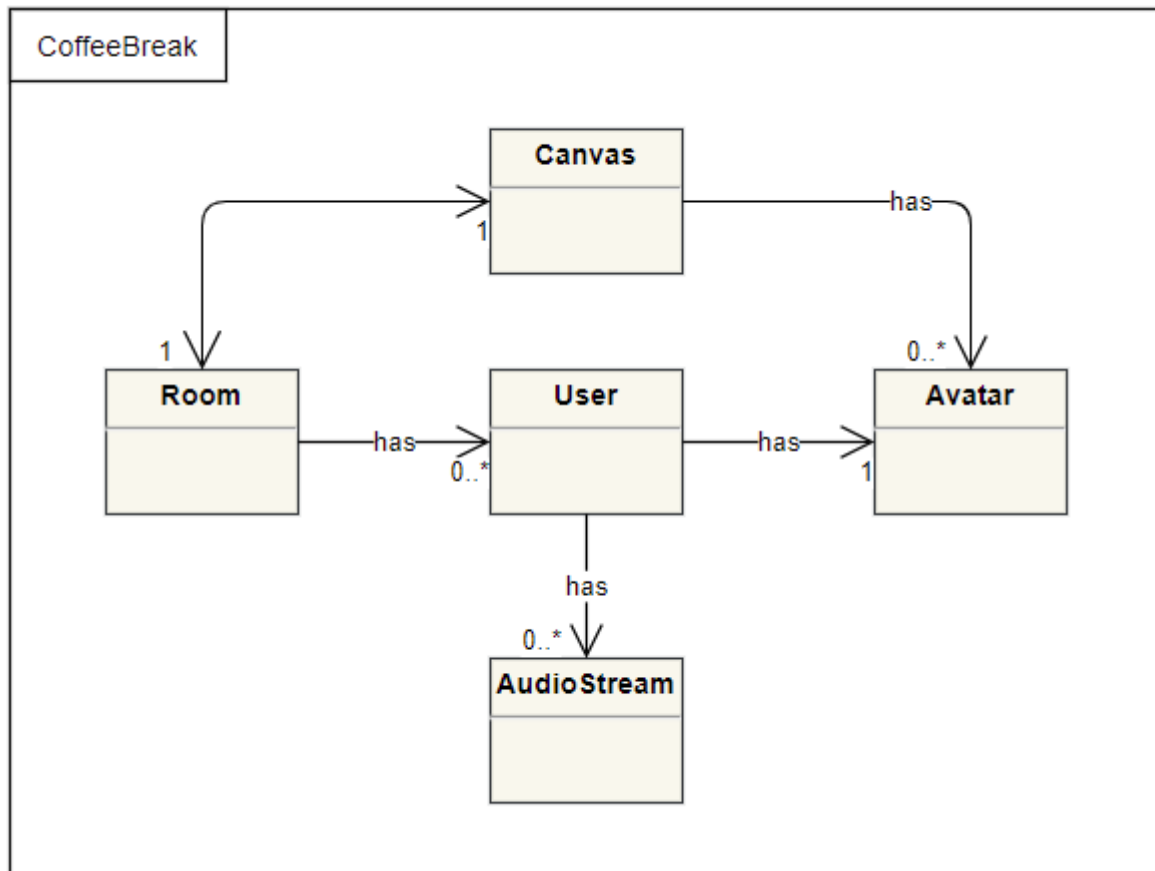
A.1.1 Use Case diagram 1



A.1.2 Use Case diagram 2

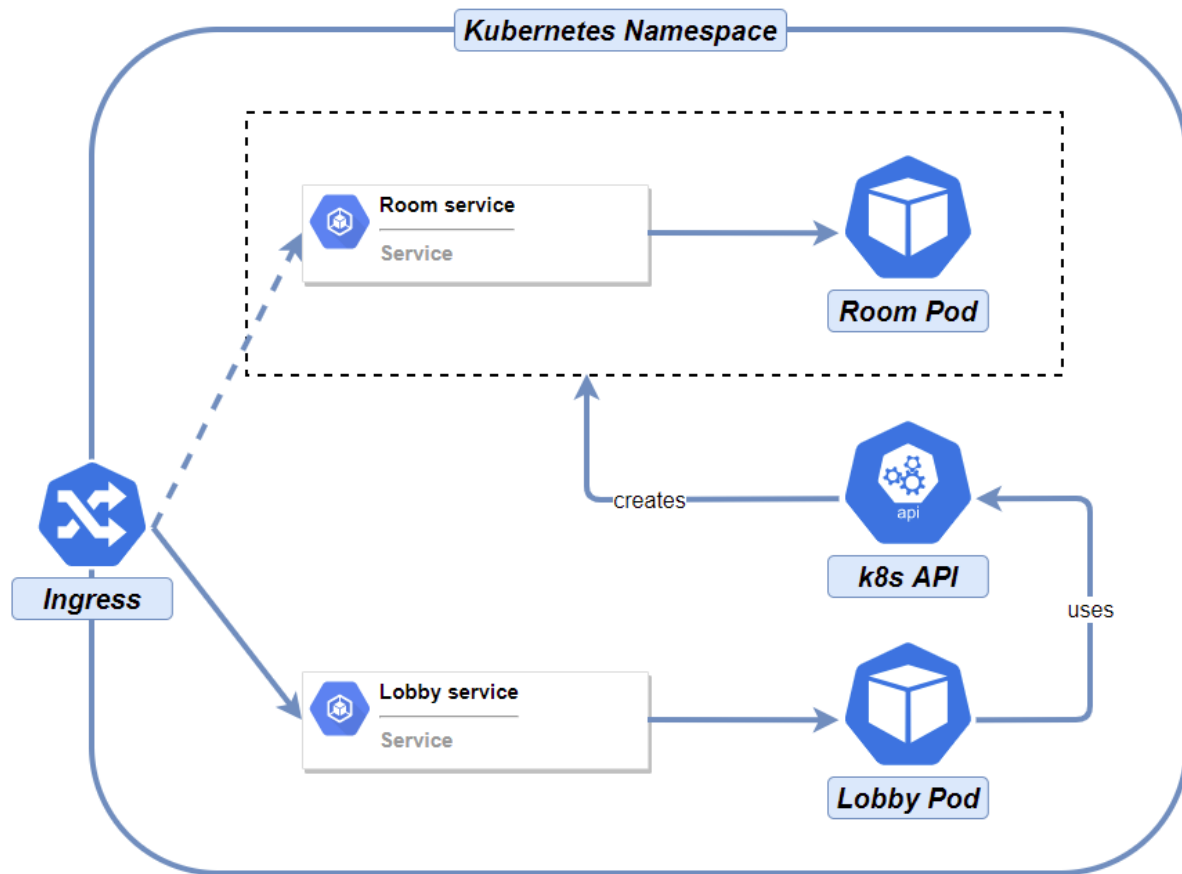


A.2 Domain model

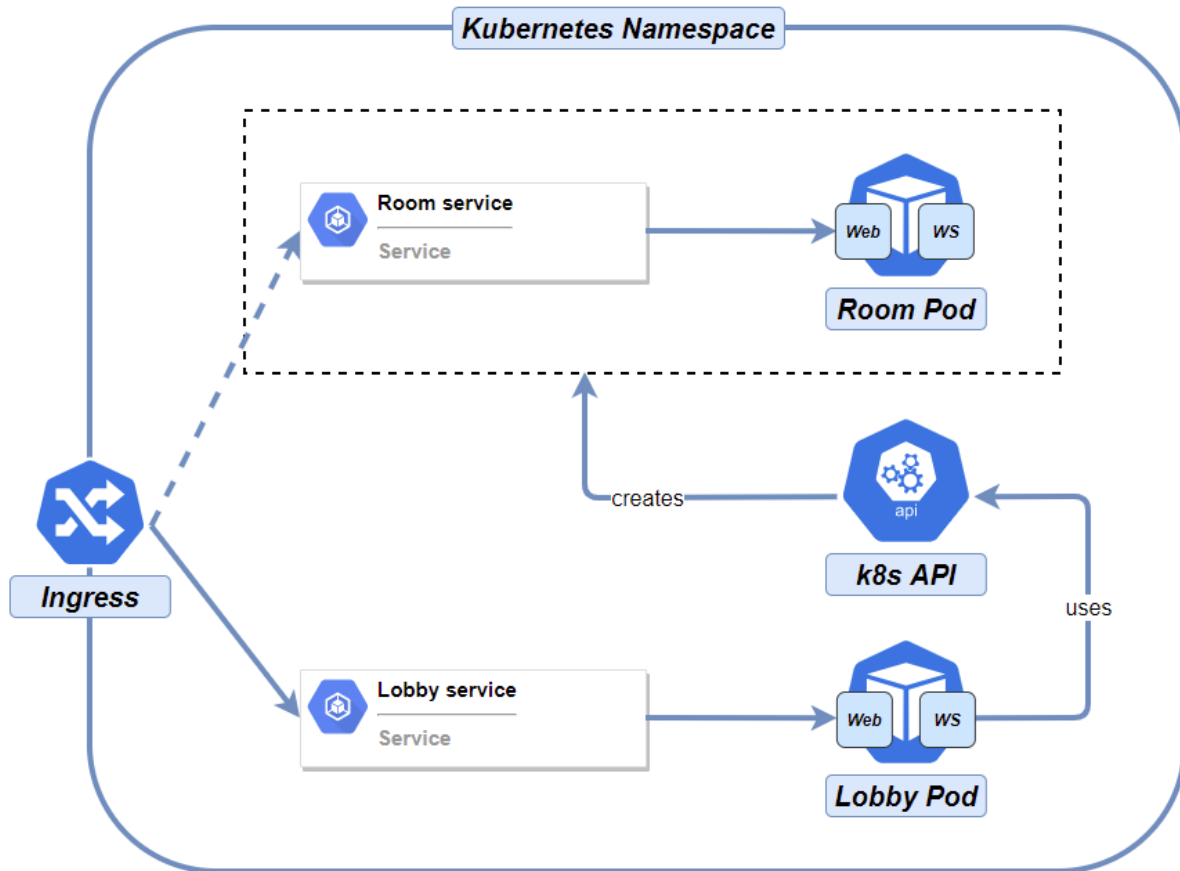


A.3 Containerization models

A.3.1 Containerization model 1



A.3.2 Containerization model 2



A.4 Sprint backlogs

A.4.1 Backlog, Sprint 1

| Title | Description |
|----------|--|
| FR1.1 | Users can create rooms |
| FR1.2 | Users can invite other users to their room |
| FR1.2.1 | Users can invite other users to their room |
| FR1.3 | Users can join rooms they are invited to |
| FR1.3.1 | A room can be joined using its unique ID |
| FR2.1 | Users can assign and change their nickname |
| FR4.1 | Volume of speakers varies based on distance from user (farther = lower) |
| FR5.1 | User can place and move their avatar within the 2D plane |
| Basic UI | Build a basic UI which incorporates the essential features of the system |

Table A.1: Backlog of Sprint 1

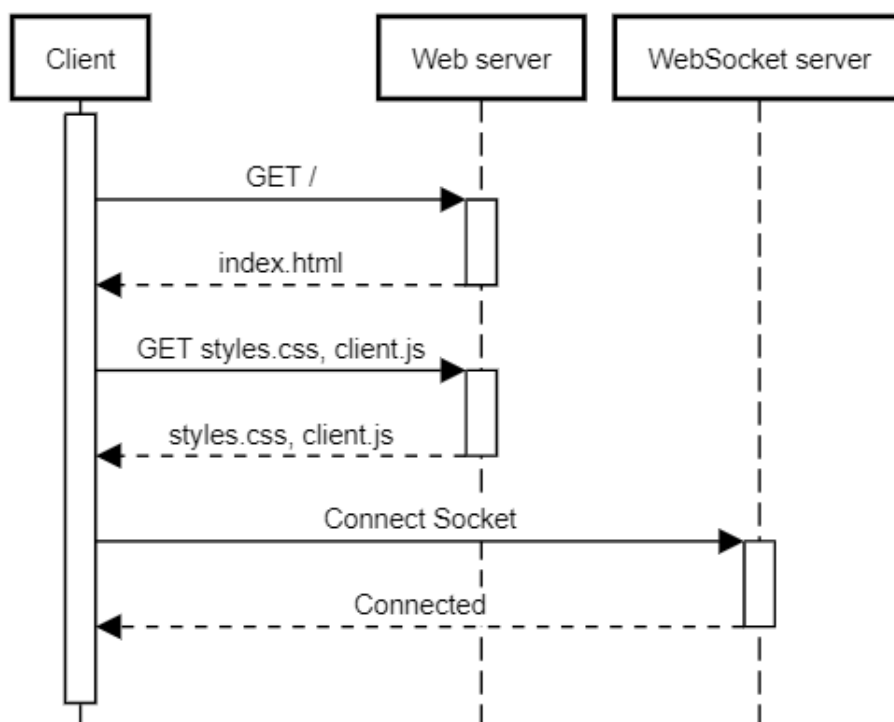
A.4.2 Backlog, Sprint 2

| Title | Description |
|---------------------------|--|
| Room system k8s rework | Rework the existing room system to work in the context of containerizing rooms, such that each room is a Pod |
| Dynamic Room Pod creation | Implement dynamic creation of Room Pods as a response to the user clicking "Create Room" |
| Dynamic Room Pod exposure | Make Room Pods created dynamically get exposed to users by creating the necessary Kubernetes resources |

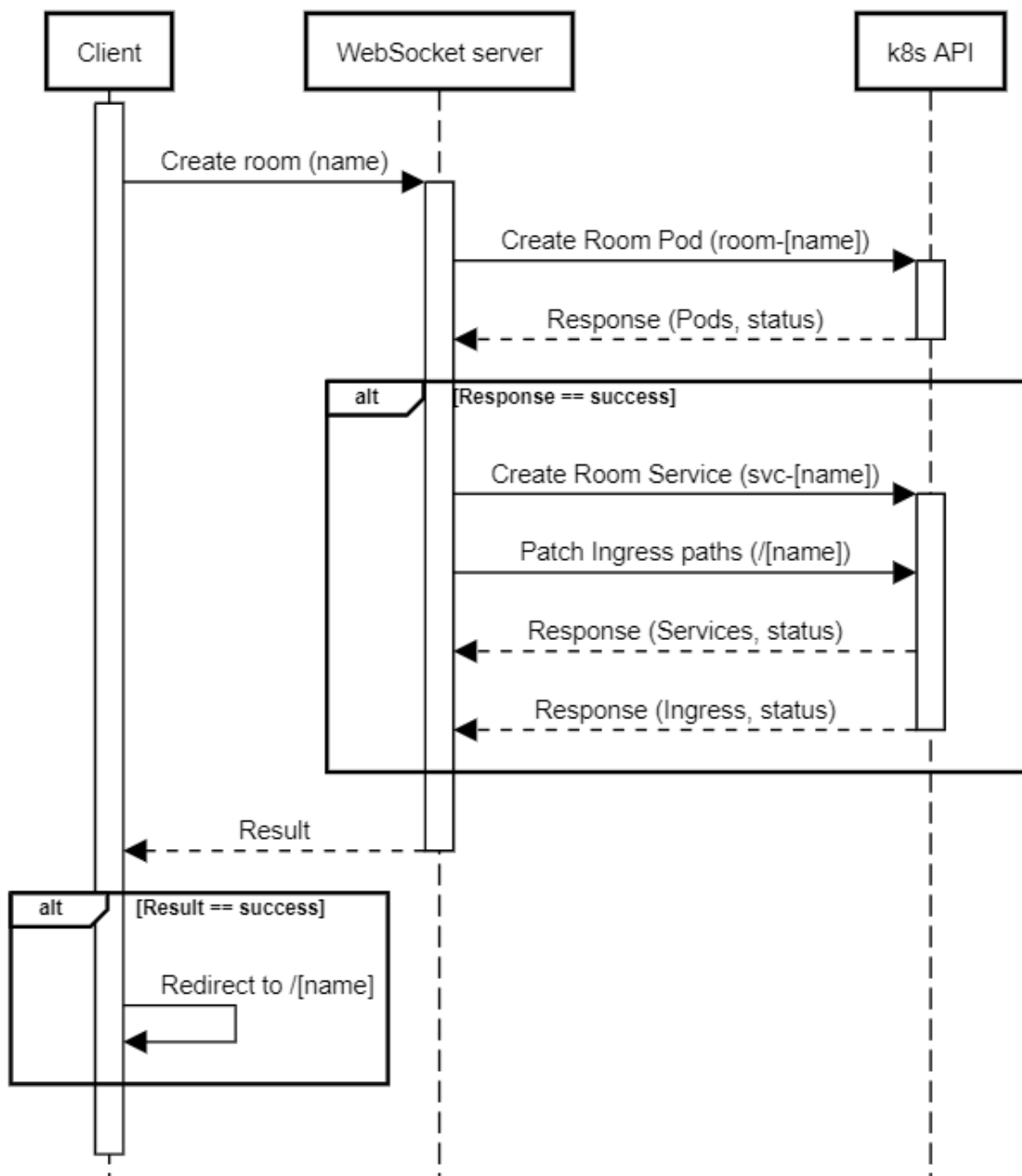
Table A.2: Backlog of Sprint 2

A.5 Sequence diagrams

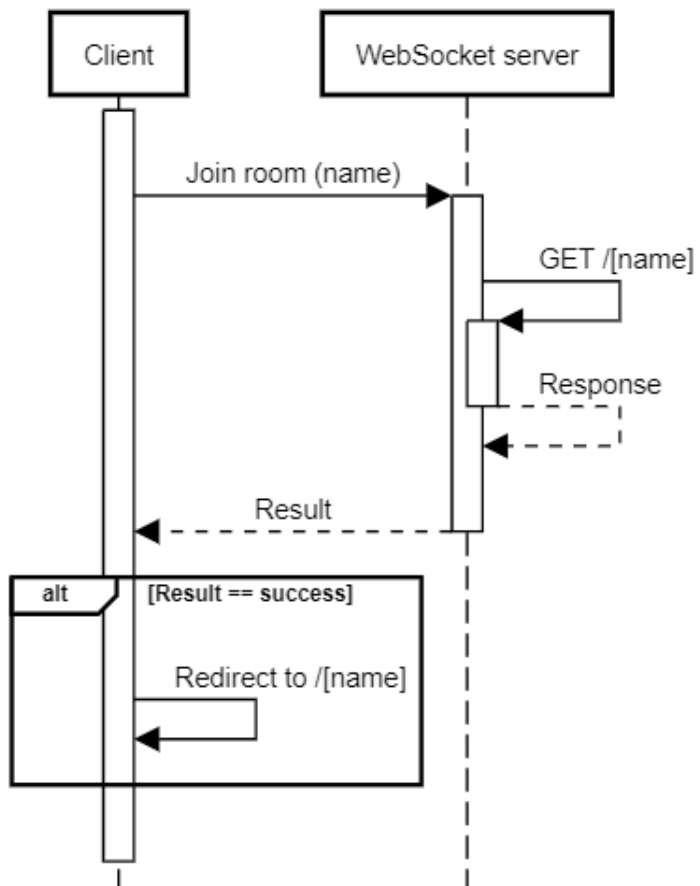
A.5.1 Lobby connect



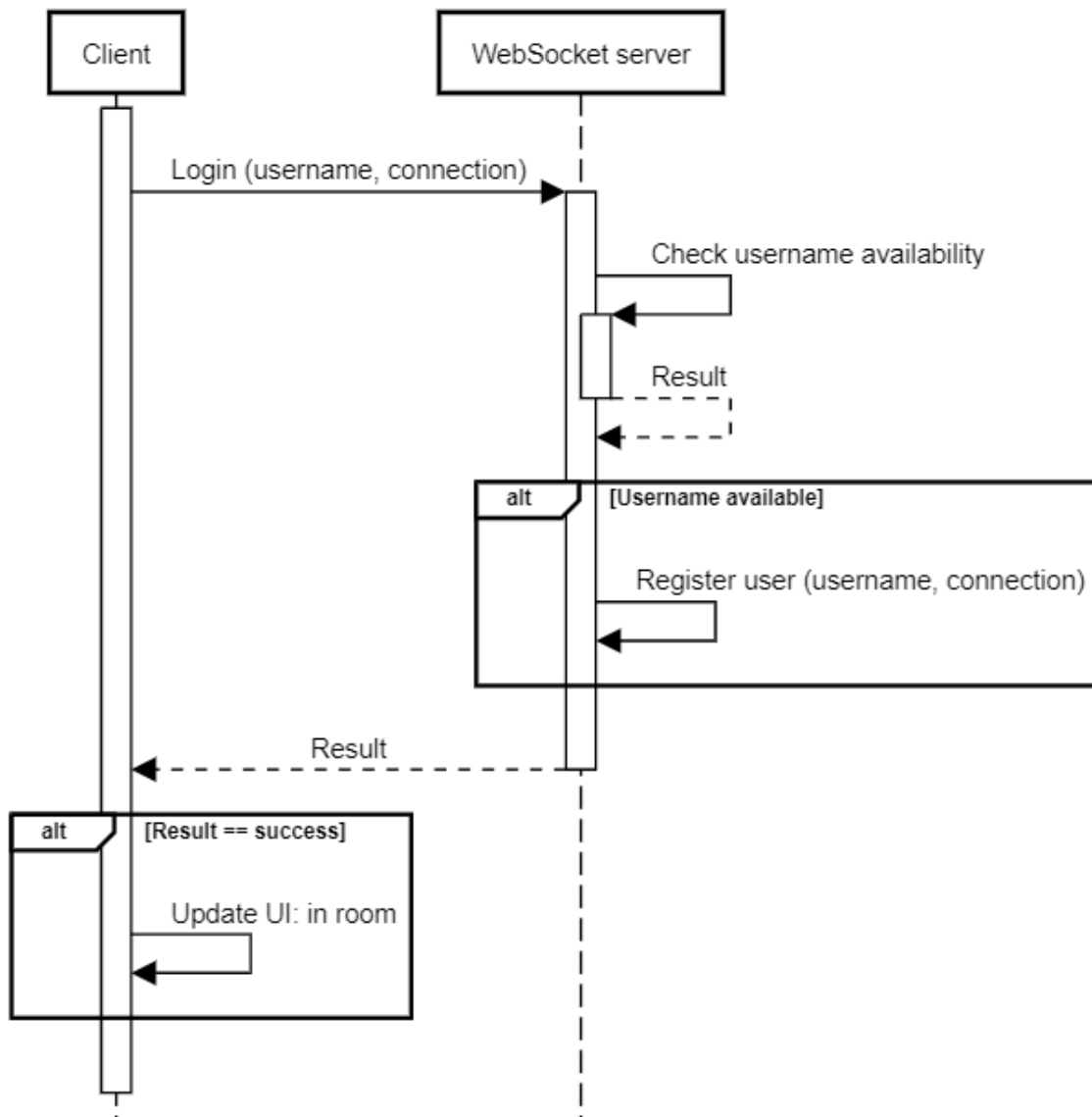
A.5.2 Create room



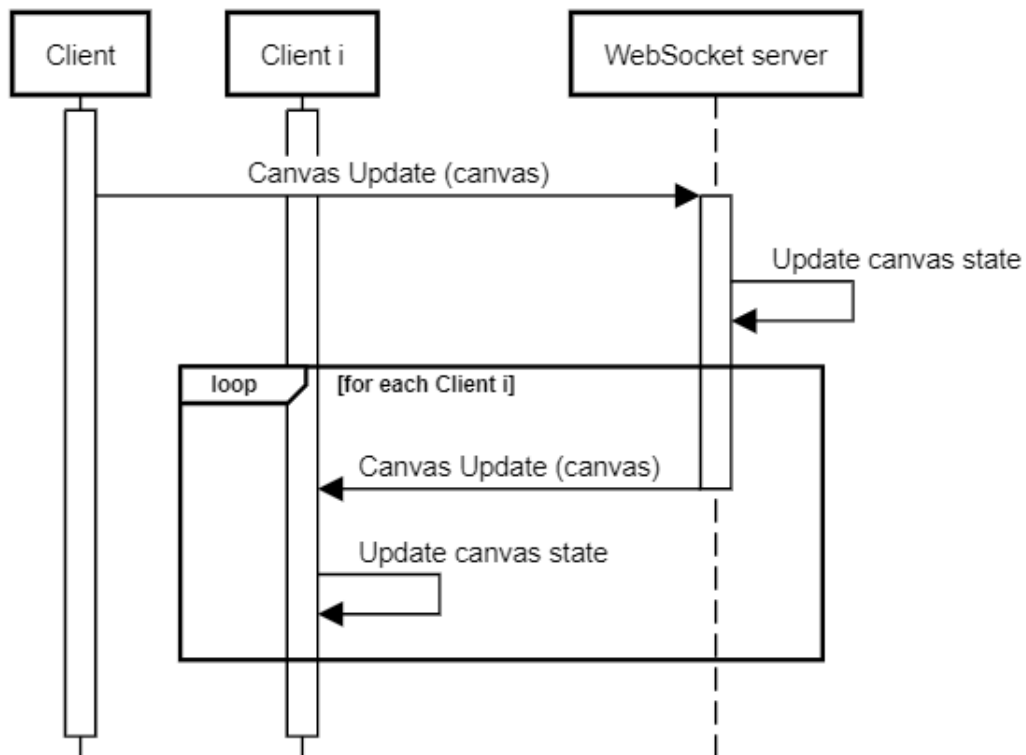
A.5.3 Join room



A.5.4 Login

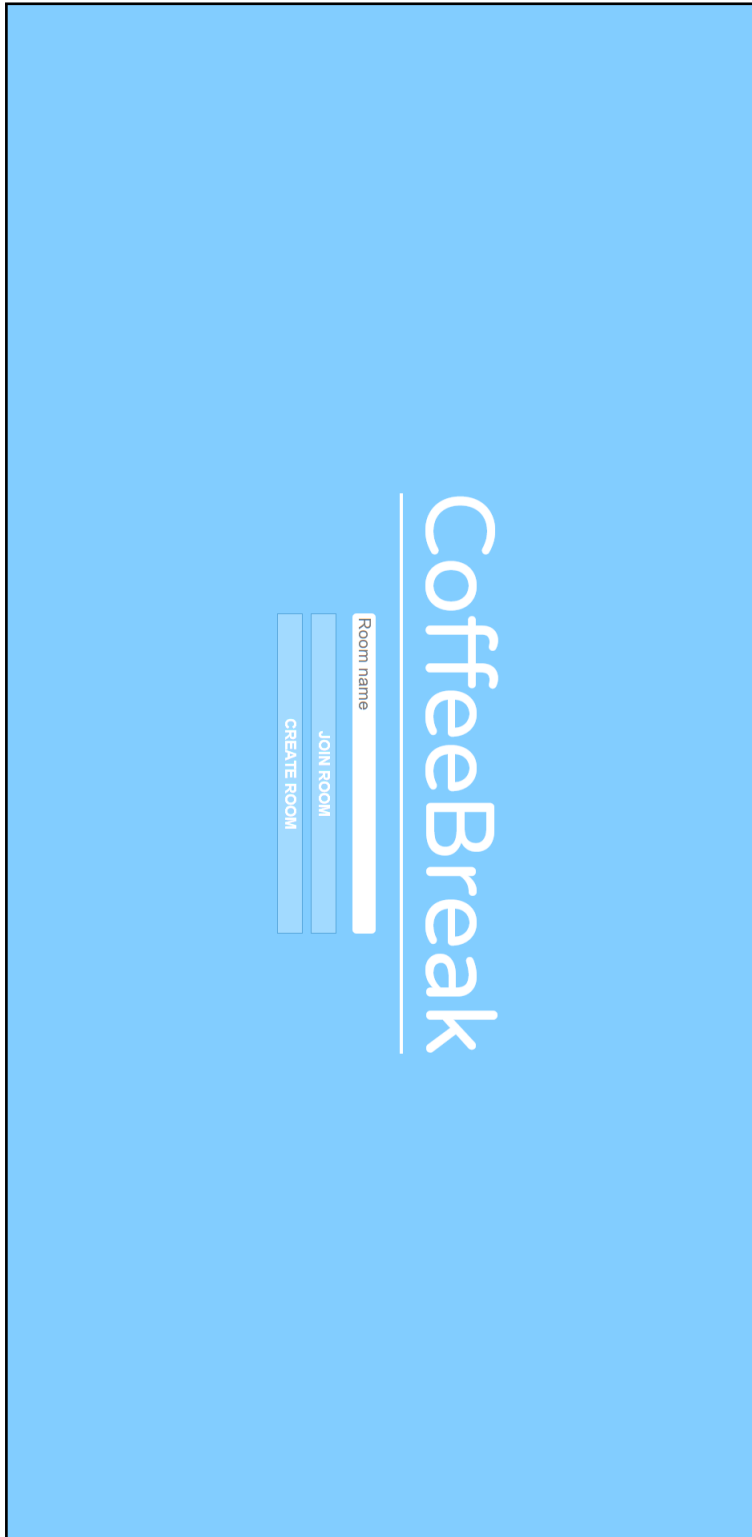


A.5.5 Canvas Update

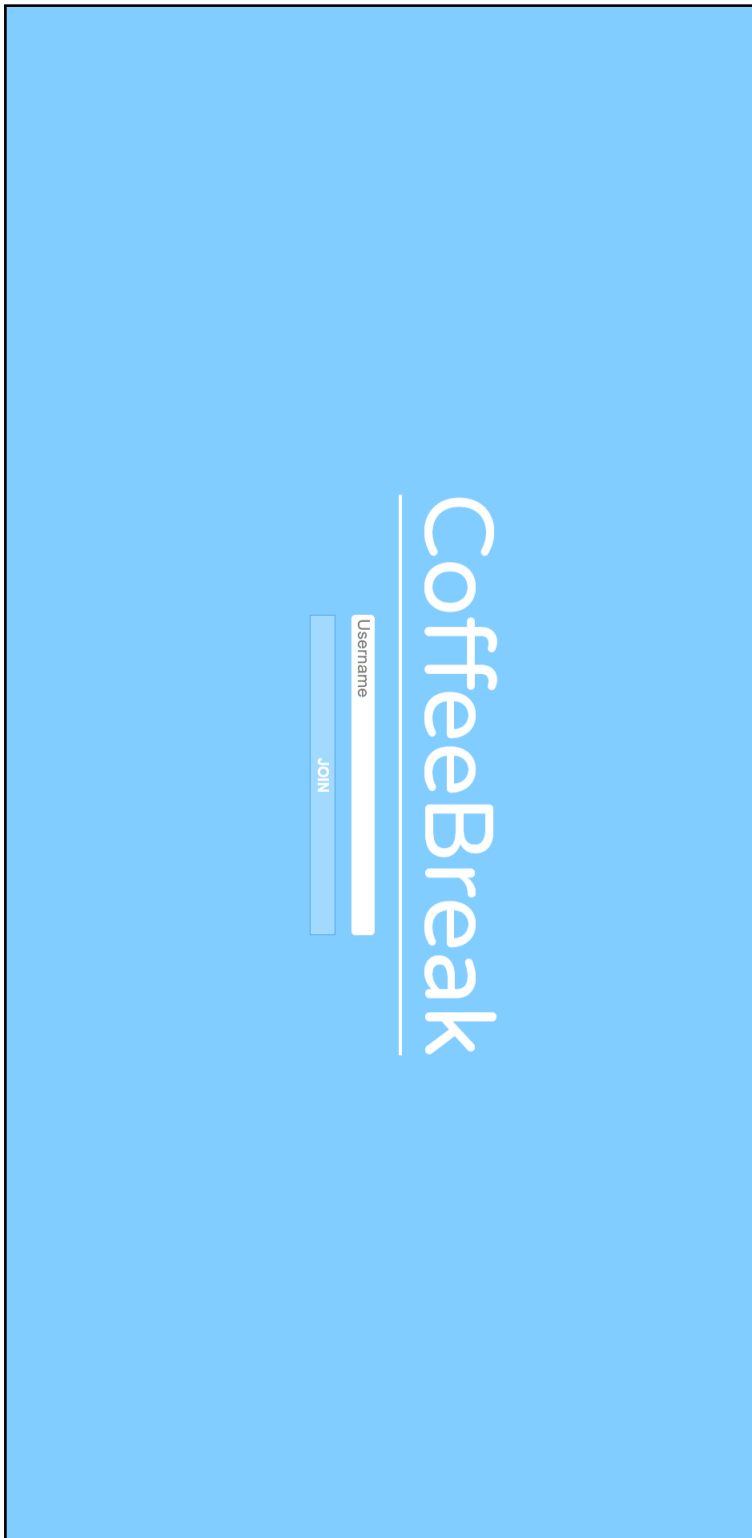


A.6 User interface

A.6.1 Join/create room view



A.6.2 Login view

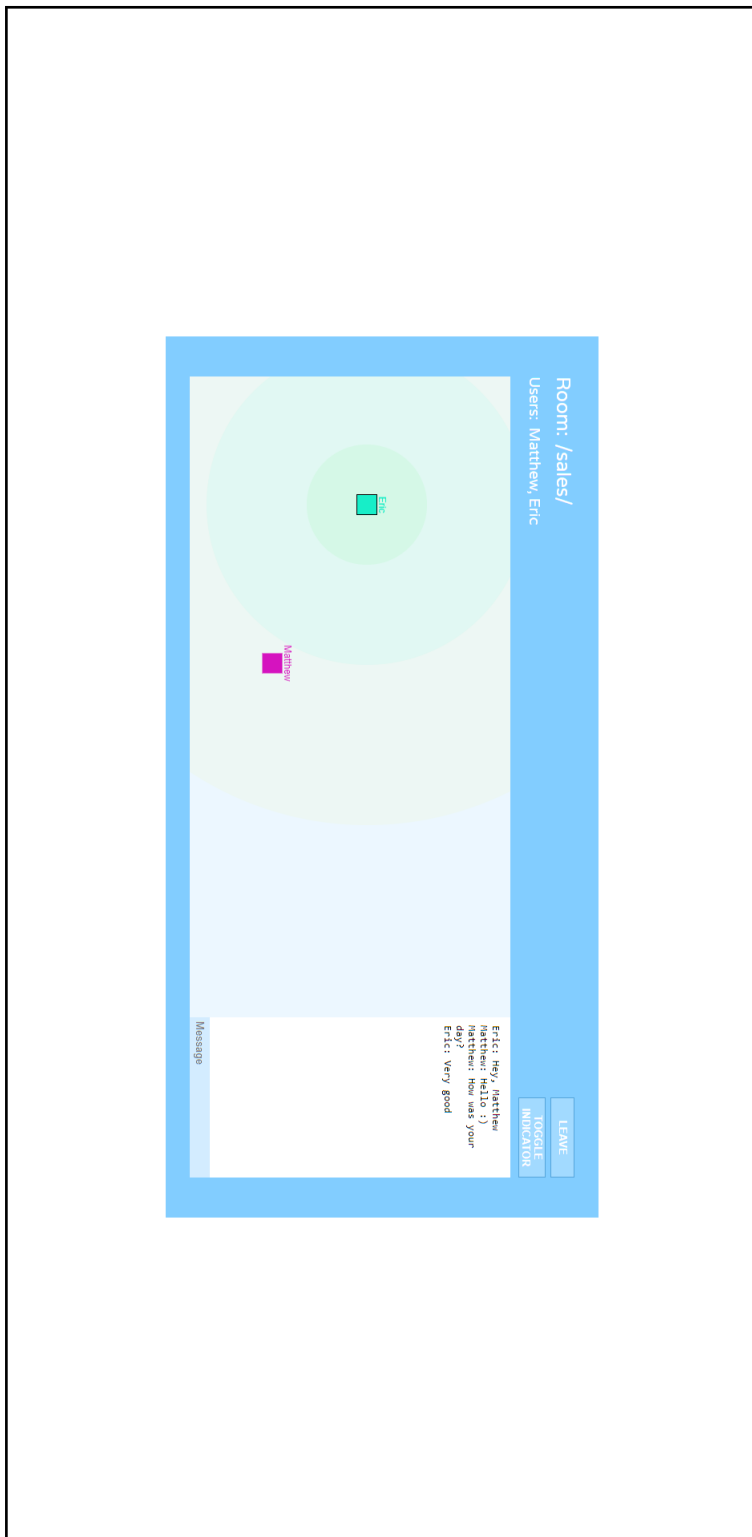
The image shows a login view for an application named "CoffeeBreak". The background is a solid light blue. The text "CoffeeBreak" is written in a large, white, sans-serif font, centered horizontally. Below the text, there is a white rectangular input field with the placeholder text "Username" in a small, dark font. To the right of the input field is a light blue rectangular button with the word "JOIN" in white, uppercase letters. The entire login form is centered on the page.

CoffeeBreak

Username

JOIN

A.6.3 Room view



A.7 Lobby Dockerfile

Listing A.1 Lobby Dockerfile

```
1 FROM node:16-alpine3.11
2 WORKDIR /work/

4 COPY package.json /work/package.json
5 RUN npm install

7 COPY . /work/

9 EXPOSE 80
10 EXPOSE 8082

12 CMD node server
```

A.8 Room Dockerfile

Listing A.2 Room Dockerfile

```
1 FROM node:16-alpine3.11
2 WORKDIR /work/

4 COPY package.json /work/package.json
5 RUN npm install

7 COPY . /work/

9 EXPOSE 80
10 EXPOSE 8082

12 CMD node server
```

A.9 Kubernetes deployment file

Listing A.3 Kubernetes deployment file

```
1 apiVersion: networking.k8s.io/v1
2 kind: Ingress
3 metadata:
4   name: ing-coffeebreak
5   namespace: group2
6   annotations:
7     nginx.ingress.kubernetes.io/rewrite-target: /$2
8     nginx.ingress.kubernetes.io/force-ssl-redirect: "true"

10 spec:
11   rules:
12     - host: group2.sempo0.uvm.sdu.dk
```

```

13     http:
14       paths:
15         - path: /()(.*)
16           pathType: Prefix
17           backend:
18             service:
19               name: svc-lobby
20             port:
21               number: 8888

23         - path: /ws()(.*)
24           pathType: Prefix
25           backend:
26             service:
27               name: svc-lobby
28             port:
29               number: 8082

31 ---
32 apiVersion: v1
33 kind: Pod
34 metadata:
35   name: lobby
36   labels:
37     app: lobby
38 spec:
39   serviceAccountName: group2-user
40   containers:
41     - name: con-lobby
42       image: benjaminhck/coffeebreak-lobby:latest
43       ports:
44         - containerPort: 80
45         - containerPort: 8082

47 ---
48 apiVersion: v1
49 kind: Service
50 metadata:
51   name: svc-lobby
52 spec:
53   type: LoadBalancer
54   selector:
55     app: lobby
56   ports:
57     - protocol: TCP
58       name: web
59       port: 8888
60       targetPort: 80

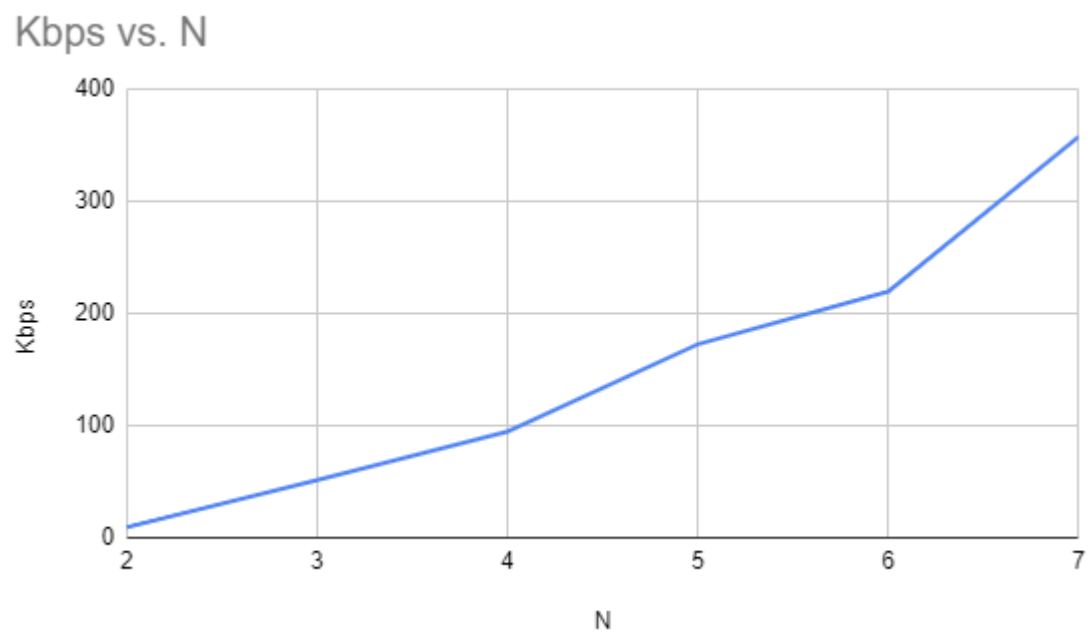
```



```
62 - protocol: TCP
63   name: socket
64   port: 8082
65   targetPort: 8082
```

A.10 Network test

A.10.1 Network test graph



Technical
University of
Denmark

Campusvej 55
5230 Odense M
Tlf. 4525 1700

www.sdu.dk/mmmi