A Multi-View Video Conferencing System in Links

Caitlin McDougall

MInf Project (Part 2) Report
Master of Informatics
School of Informatics
University of Edinburgh

2023

Abstract

Links is a functional programming language which solves the impedance mismatch problem arising in classic web development technologies by offering a single source language which compiles to JavaScript and SQL. However, it is important to explore whether the approaches used in Links are capable of providing as many features as those of classic languages. Specifically, this thesis demonstrates Links' ability to provide live peer-to-peer video conferencing functionality through the use of the WebRTC API. Additionally, this thesis presents an application which offers a selection of interfaces to users for interacting with this underlying video conferencing system. One such interface aims to provide a simple user experience with the minimum information supplied to the user to allow them to interact with other users. The other interface aims to offer a user experience which more closely resembles real-life interactions by introducing a spatial environment in which users move more naturally from conversation to conversation. In order to develop these interfaces with the same underlying system state and allow users on both interfaces to interact smoothly, a model-driven approach was used. This proved particularly effective in conjunction with the Model-View-Update pattern supplied by Links. After outlining the design and implementation of the system, we go on to discuss its achievements and offer several possible future directions for the project.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Caitlin McDougall*)

Acknowledgements

I would first like to thank my supervisor Sam Lindley for his continuous support and encouragement throughout this project.

I would also like to give a huge thanks to my family and all the friends I have made along the way, without whom I would not have come this far.

Table of Contents

Chapter 1

Introduction

1.1 Motivations

Since its inception, the internet has become integral to the way we live our lives as it allows us to find information instantly, share our interests with peers, buy groceries, and much more. In order to enable the internet to function as smoothly as it does, a variety of web technologies are utilised.

Classic web development involves learning multiple programming languages such as JavaScript, PHP, and SQL to ensure the different layers of a web application function properly. Unfortunately, using multiple languages causes friction when interfacing between their various distinct archetypes [24].

It is this impedance mismatch problem that prompted the creation of the Links programming language [2]. Links provides a single functional source language which compiles to JavaScript and SQL, allowing it to offer the functionality of all the classic web development languages. To prove the effectiveness of the approaches used in Links, it needs to be able to provide as much functionality as the classic web development languages. One such capability integral to modern society is live video conferencing as you find on Zoom [28] and Teams [18].

These traditional conferencing systems are static, meaning users do not move around to interact with one another and if they would like to enter a separate conversation, they would have to create or join a new breakout room through a menu of options. This interface is in contrast with real-life interactions in which we have the freedom to move around in a social space.

With the aim of offering a more natural environment for virtual conversations, a variety of alternative systems have arisen. One such application called Gather.town [12] supplies a gamified environment in which participants move characters around and when they are within close proximity of another character, they begin communicating. Thus, users navigate between conversations in a manner more akin to real-life interactions.

FluidMeet [11] is another such system which instead offers users a choice of how much spatial information they receive from the system. This is important as having

an elaborate interface could prove overly complex to those with visual impairments or other disabilities who are found to prefer a clear and simple design [8].

1.2 Approach

Consequently, this thesis proposes the creation of a Links-based video conferencing system (VCS) which supports the choice of either a gamified or traditional experience, allowing participants on both systems to interact with one another.

From the perspective of clients accessing the application, we provide the option of a gamified spatial interface or a simple static interface for interacting with the VCS. In our spatial interface, the user sees their selected avatar located on a visual map of available rooms through which the user can navigate between different conversations using their keyboard. In our static interface, the user navigates between conversations by simply selecting their desired room from a list, without any knowledge of the underlying map.

To enable live communication between clients, the system successfully makes use of the WebRTC framework which provides a JavaScript API for creating and maintaining peer-to-peer connections [16]. In order for Links to connect to this API, it must make use of a Foreign Function Interface (FFI) which allows it to call JavaScript functions directly from Links code [6]. These JavaScript functions take care of media capturing, setting up RTC connections, and generating offer or answer messages during call initialisation. These offers are then sent via the Links code to the server's mailbox which forwards them on to the connecting client. Although connecting to the WebRTC API through the FFI proved successful, we also describe how this highlighted a limitation in Links' ability to handle the asynchronicity which sometimes arises in JavaScript functions.

During the development of the frontend interfaces, a model-driven approach was used such that each view exposed to the client is abstracted from the underlying implementation of the system's state [15]. Links makes this particularly easy to achieve by providing functionality for a Model-View-Update architecture [7]. This approach consists of a model to maintain the state of the application, a view function to render the model, and an update function which handles messages produced by these models and produces new models. With this architecture, we show that offering two interface options can be achieved with the same underlying model state by simply changing the view function and adding an additional update message to handle movement.

1.3 Thesis Outline

This thesis first goes through the background research which motivates and underpins the system in Chapter 2. Then, Chapter 3 describes and justifies the design choices made during the development of the system and is followed by Chapter 4 which outlines the technical implementation used to realise this design. Thereafter, Chapter 5 evaluates and discusses the various achievements and limitations of both the system implementation and interfaces. Finally, Chapter 6 concludes the paper, summarising its successes and suggesting directions for future work.

Chapter 2

Background

2.1 Web Technologies

The internet plays many important roles in modern life, allowing us to find information instantly, share our interests with peers, buy groceries, and much more. In order to enable the internet to function as smoothly as it does, a variety of web technologies are utilised.

2.1.1 Classic Approaches

Commonly, the client-side consists of markup languages such as HTML and XML to specify the basic structure and content of the web-page, a styling language such as CSS, and a language such a JavaScript to provide dynamic functionality [2]. On the server-side, languages such as Python, Java and PHP can be used for performing more intensive computations and functionality. Finally, languages such as SQL and XQuery allow connections to databases which store the huge quantities of data which may be requested on a website. These different layers, or tiers, allow modulation of websites and allows for languages which are tailored to specific purposes.

Unfortunately, having this variety of programming languages to choose from at each level creates friction known as the impedance mismatch problem. Impedance mismatch problems are issues which arise as a result of combining technologies which use different archetypes [14].

2.1.2 Links

For this reason, the Links programming language has been created, offering an alternative approach to web technologies which replaces this three-tiered web model with a single language. Links is a strict, statically-typed, functional language which aims to replace the three-tiered web system with a single-source language [2]. It does so by providing a translator from the Links code to JavaScript and SQL, with the functional aspect of the language providing additional benefits such as database query optimisation, continuations for web interaction, and concurrency with message passing.

The pattern classically used when programming in Links to achieve concurrency is similar to that of the Actor Model [5]. In this model, an Actor is responsible for managing its own state and performing computations [9, 10]. In this way, there is no global state shared across actors or processes which allows for concurrency since computations are safe from attempting to modify the same memory location. Messages are sent between different actors and when an actor receives a message, it can perform any of 3 concurrent actions: send messages to other actors, create additional actors, or prepare for how subsequent messages will be handled [9]. However, the imperative nature of the Actor model is not well-suited to the implementation of graphical user interfaces (GUI) which makes it more challenging to create webpages in Links.

For this reason, an adaptation of the Model-View-Update (MVU) architecture introduced by Elm has been added to Links [7]. Elm, like Links, is a functional language and, as such, the architecture it presents is particularly well-suited to functional programming. In this architecture, a model contains the state of the application, a view function renders the model, and an update function handles messages produced by the rendered model and produces new models. This offers a declarative approach to developing user interfaces which is not available with a purely Actor-based model. Figure 2.1 shows the components and interactions involved in the classic MVU architecture.

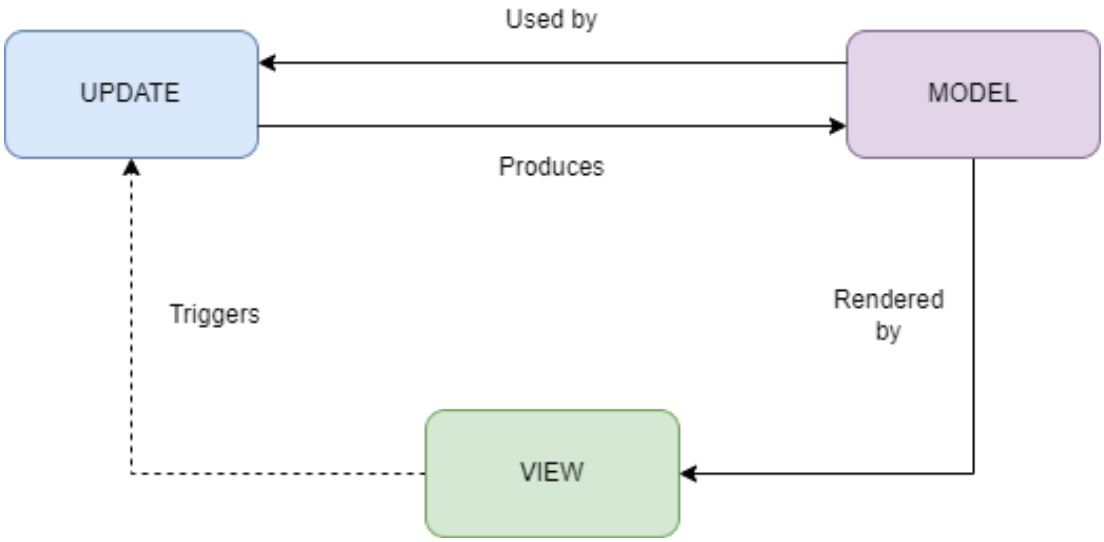


Figure 2.1: The standard Model-View-Update pattern.

Since the Links language is still in active development, it still lacks some functionality which is provided by JavaScript. Fortunately, Links offers a Foreign Function Interface (FFI) which allows custom JavaScript functions to be written and called from a Links application such that any missing JavaScript functionality can still be included by the programmer [6]. In particular, this paper will use FFI to access the WebRTC framework to achieve real-time communication between clients.

2.2 WebRTC

WebRTC [3] is a set of standards making use of peer-to-peer connections to enable real-time applications such as text-based chat, audio sharing, and live video conferencing. This framework allows peers to send and receive information directly, without first sending the data through a server which can create delay [16]. The WebRTC API will therefore be required to allow our video conferencing application to facilitate the exchange of live video and audio between clients.

2.2.1 Client-Server vs P2P

In the traditional Client-Server network architecture, devices in a network act as either a client or a server. Clients make requests for services and content which are received and actioned by the server, a higher-performance entity which is usually connected to a large number of clients [24]. In this way, clients are not connected directly to one another and in order to share content must instead send content first to the server which can in turn forward this to the destination client.

On the other hand, in the peer-to-peer (P2P) architecture, devices in the network can act as either a client or service provider at any point. In this way, devices in a P2P network share their combined resources in order to send content directly to one another without passing through a central entity [24]. WebRTC chooses to make use of the P2P network architecture due to the fact that these direct connections between clients give less delay for real-time applications while also giving more privacy to those communicating [16].

Although WebRTC uses P2P connections for file transfers during a session, it does make use of servers to properly manage each P2P connection. For example, when two clients access a video conferencing website and want to communicate with one another, a signalling server will be set up to allow initial communication between the two clients.

A diagram of the exchanges necessary to set up a WebRTC connection between two peers is shown in Figure 2.2 and these steps are described in detail below.

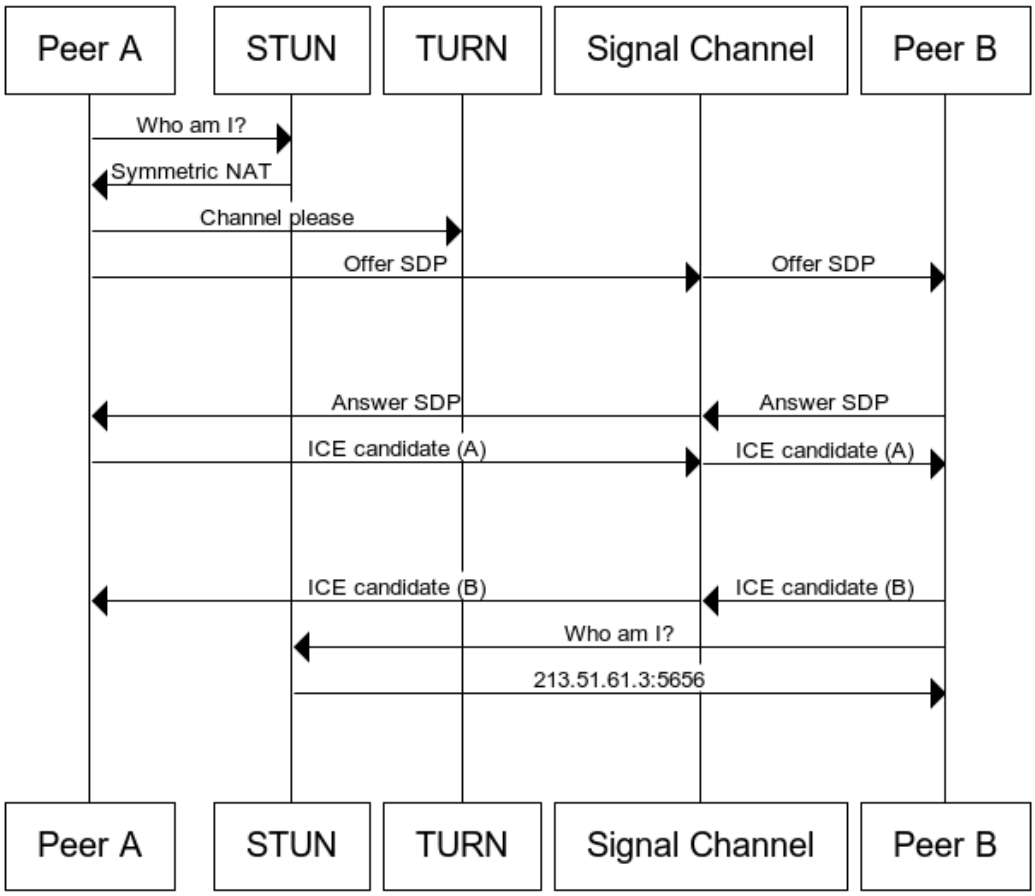


Figure 2.2: Exchanges involved in the initialisation of a WebRTC connection. Sourced from: [19].

2.2.2 ICE

In order for two clients to communicate with one another directly, they first need to know where within the network the other client is located, or in other words, their IP address. However, it is likely that both clients are connected to a router which performs Network Address Translation (NAT) [3]. This means that the local IP address of the client is hidden from the public network and replaced with a public IP address which is the address external hosts should use to communicate with the client. Since this translation happens at the router, the client does not know its own public IP address and so can't automatically tell other clients which address to use to connect to it.

Fortunately, Session Traversal Utilities for NAT (STUN) servers give clients the ability to ask for their own public IP address such that they can distribute this information to clients or servers they want to be able to connect to them [23]. If, however, the router uses symmetric NAT, which means that the NAT mappings are dependent on both the source and the destination IP addresses, this no longer works as other servers would still be unable to connect since their IP address is not trusted. In this situation, Traversal Using Relays for NAT (TURN) servers are instead utilised to route messages

to and from the clients, therefore creating a client-server connection rather than a P2P connection [20].

Interactive Connectivity Establishment (ICE) is used to describe the framework of collecting the different candidates which may be used to connect to the client such as their local IP address, public IP addresses given by STUN servers, or TURN server IP addresses [22].

2.2.3 Signalling

The ICE candidates are bundled up along with information about available media types and formats according to the Session Description Protocol (SDP) [22]. Clients can then exchange these SDP formatted messages as an offer to the other client and the other client responds with an answer until an agreement on how to communicate is achieved. This process is called Signalling and is facilitated by the Signalling server. Once the direct connection is agreed upon, the clients can then exchange media using a P2P connection.

2.3 Video Conferencing Systems

Technology has long been used to communicate with one another from afar, but mostly in the form of telephone calls or text-based messaging. However, the ability to send and receive video streams has revolutionised the way businesses, schools, and social circles function. The ability to share video with multiple users at the same time has become particularly relevant in the wake of the Covid-19 pandemic which saw the use of video conferencing systems (VCS) boom as this became the closest to face-to-face meetings we could get.

A variety of VCS are available to use, with popular systems including Zoom [28], Microsoft Teams[18], and Cisco Webex [1] all sharing a similar interface. The interface consists of live video feeds for a subset of participants, a larger area displaying the current speaker or currently shared screen, a small area showing the user's own live feed, and an area for text-based conversation. These systems also come with an increasing number of additional features to mimic the way we interact in real meetings such as raising hands, white-boarding and breakout rooms.

However, even with the inclusion of breakout rooms, these systems fail to adequately represent the spatiality of real meetings and social interactions. For example, a teacher may walk around the edge of a classroom to get a sense of who is struggling, while in these virtual environments they need to enter every breakout room, potentially interrupting the conversation. This limitation of traditional VCS has motivated new approaches to video conferencing which take into account this spatial awareness and enable more natural virtual interactions.

2.3.1 Gather.town

One such VCS offering proximity-based interactions is Gather.town [12], an online platform which allows companies or individuals to create a gamified virtual meeting space in which customised avatars can move around and interact with other avatars. An example of a group call in a Gather.town space is shown in Figure 2.3 [25]. When two avatars are within close proximity they will be connected and begin sharing video, whereas when they are far apart the connection stops and they are unable to see or hear one another. In addition to this, spaces can be set up for broadcasting to a large room of people as would be possible during a presentation and tables are available in which everyone can communicate with one another similarly to the traditional Zoom and Teams approaches.

Figure 2.3: A group call between participants in Gather.town. Sourced from: [25].

Additionally, Gather.town is increasingly incorporating many of the features offered by traditional VCS such as white-boarding, screen-sharing and file-sharing. This ensures that users do not miss out on any functionality by switching to Gather.town. Gather.town also provides features not offered by static VCS as it has the unique ability to provide interactive games which users can play. This allows for a much more natural experience in environments such as classrooms or icebreaker sessions. Games offered include Chess, Codenames, Tetris, and more [13].

Some limitations of Gather.town which have been highlighted are its limit of 25 participants in the free version, and the performance of the software when multiple users are connected on a poor internet connection [27]. It has also been suggested that this visual interface could be inaccessible for those who find too much visual stimulus overwhelming or those with visual impairments who may struggle to interact easily with the virtual environment.

Overall, Gather.town provides a much more interactive experience for users and has been shown to reduce fatigue associated with virtual meetings. In addition, a study by [17] determined that both educators and students preferred using Gather.Town in comparison with using other static VCS such as Zoom and Teams. However, due to the potential accessibility issues mentioned previously, there is a need for a video

conferencing system which provides the benefits of Gather.Town's spatial and gamified interactions while maintaining an appropriate level of accessibility for those with visual impairments.

2.3.2 FluidMeet

Another alternative approach to live video conferencing is proposed by FluidMeet [11]. In this system, the authors have focused on creating flexible boundaries between conversations (unlike the rigid break-out room boundaries offered by alternatives such as Teams). In doing so, they aim to offer more natural transitions between conversations and provide a more visual representation of different conversations.

FluidMeet achieves more natural transitions by offering the option for breakout rooms to be fully or partially open to others [11]. When fully open, users who are not in the group are able to hear and see what is going on in that group before joining. When partially open, non-group members instead have access to keywords being used in the conversation in the form of a word cloud and they can view audio visualisations in order to gain a sense of the atmosphere of a group. Figure 2.4 shows the Lound and Breakout Room interfaces provided by FluidMeet [11].

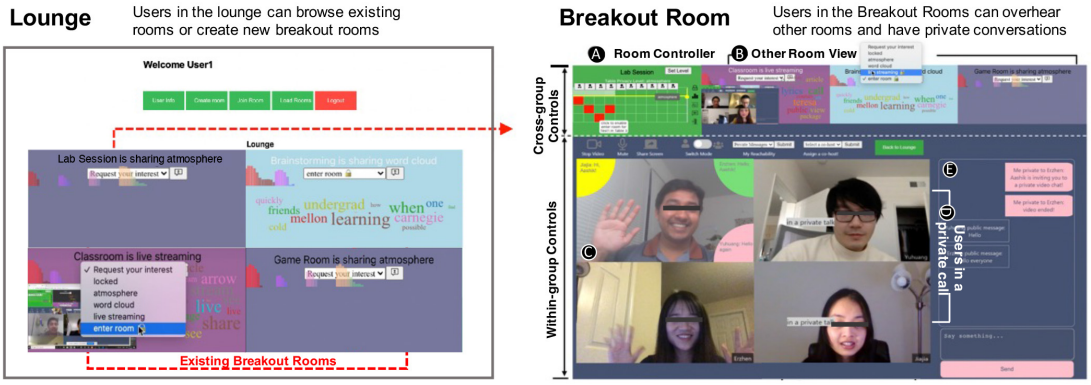


Figure 2.4: The FluidMeet user interface with relevant sections in the Lounge and Breakout Room labelled. Sourced from: [11].

Having access to all this information could be overwhelming for users, but FluidMeet offers the ability to choose which of these features are displayed, reducing the visual overload which may occur. This is an important feature to note as allowing users to customise how much information they receive enhances the accessibility of FluidMeet to those with visual impairments or those who just prefer a simpler interface.

2.3.3 Accessibility through Multiple Interfaces

A study by Gappa and Nordbrock (2004) explored ease of use in internet portals (search engines) which included individuals with hearing impairments, visual impairments, learning difficulties, and the elderly [8]. Their findings showed that all participants valued a clear and simple design. This shows that adding many layers of complexity which may be preferable for some, is not fit for purpose in all situations.

In the context of video conferencing, it is extremely important that disabled students, employees, and teachers have the same access to these systems and can communicate effectively with peers and colleagues. Therefore, if companies and schools want to use these interactive and spatial VCS, there need to be features included to allow disabled or elderly individuals to participate equally.

The ideal system is one which caters to both the needs of those who prefer simpler interfaces and those who prefer a more interactive experience. Therefore, this dissertation explores offering a choice between a simplified interface offered by static VCS and a gamified experience as offered by Gather.town. However, this produces a challenge in terms of allowing those using one interface to interact with those using the alternative interface.

2.4 Model-Driven Architecture

The Object Management Group (OMG) presents a Model-Driven Architecture (MDA) as an approach which allows us to abstract this problem of having multiple views from the underlying technical details [26]. MDA uses an incremental approach to the system design process in which models are kept at the forefront of thinking. These models represent different levels of abstraction of the underlying system and thus make the overall system easier to maintain and adapt over time. Once these models have been designed from the highest to the lowest level, and the interactions between them have been defined, they can then be converted to code.

Thus, MDA is beneficial to the development of our system in which the two visual interfaces offered to the user can be seen as two different models representing a high-level abstraction of the same underlying system model.

Chapter 3

Design

3.1 System Overview

Links aims to provide a frontend language which provides programmers with the tools to create any application which can be created using traditional web-development technologies. In order to further test that this is possible, we outline the design of a video conferencing web application written in Links. This application not only facilitates live video and audio communication between users but also takes advantage of the Model-View-Update architecture provided by Links to offer multiple interface options to the user.

A diagram of the components and their relationships is shown in Figure 3.1. The application consists of a Server which maintains the overall state of the application. When a user enters the web address at which the application is hosted, a Client process is generated that hosts a variety of sub-components: a mailbox which allows message exchanges with the server; a JavaScript-based Foreign Function Interface (FFI) which handles WebRTC interactions and media capture; a model to maintain the client state; an update function which updates the model in response to actions; and a view function which uses the model to generate HTML to be displayed to the user. The clients are initially only able to exchange messages with the server but once an RTC connection is made between two clients, they are able to exchange messages directly.

As described in Subsection 2.3.1, allowing more natural, spatial interactions between users in a video conferencing system can be beneficial for enhancing engagement with online meetings and reducing fatigue. However, it is also noted in Subsection 2.3.3 that it is important to offer a more traditional, simplified interface for those who may be overwhelmed by too much visual information at once. Further, if both of these interfaces are to be offered, it is vital that the individuals utilising different interfaces are not segregated from one another. Therefore, our application will make use of Links' built-in MVU architecture to allow different view functions of the same underlying model state such that users with each interface can interact with one another. From the perspective of the user, when they access the application they will first be prompted to choose their preferred video conferencing style, static or spatial, and this choice will influence how the underlying model state is displayed to the user by the View function.

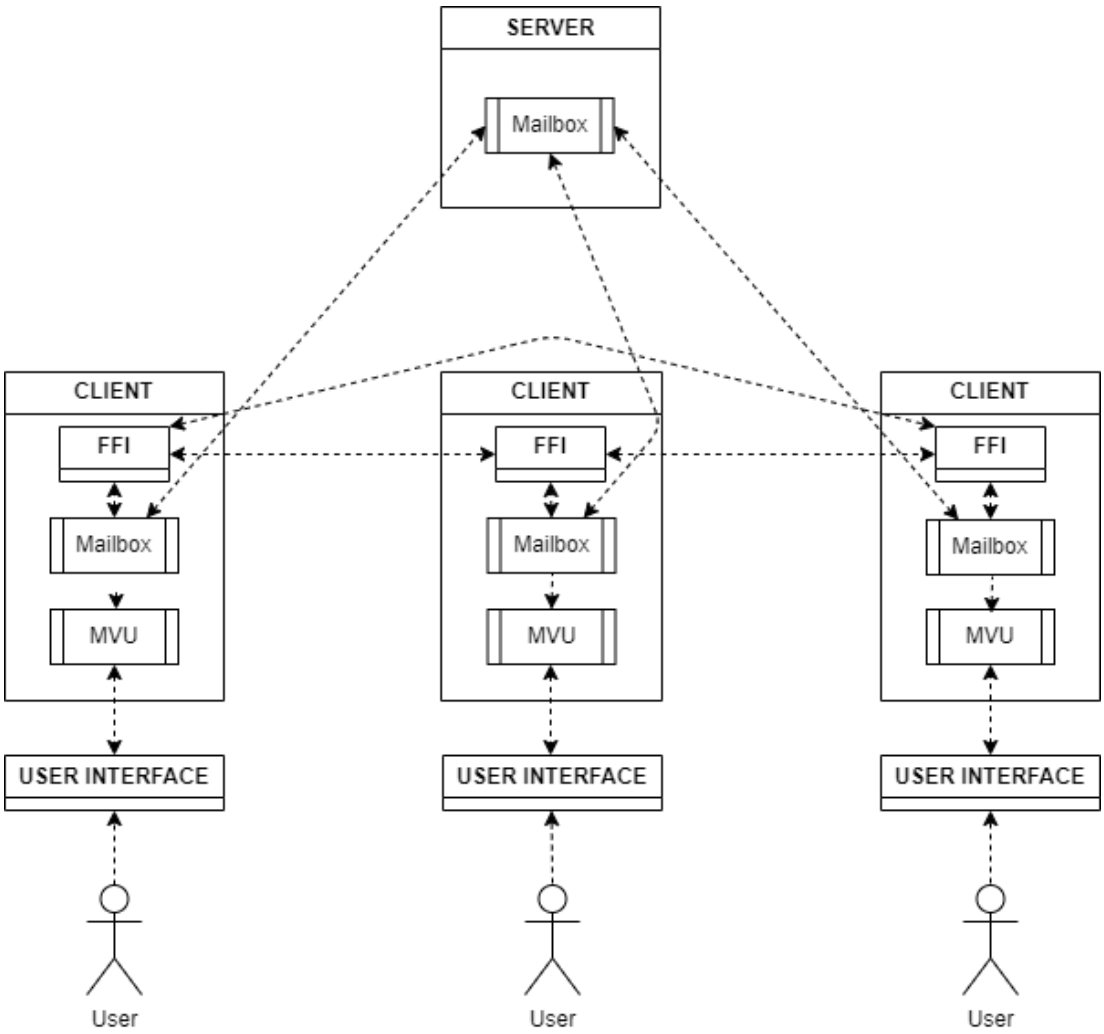


Figure 3.1: System diagram of VCS components and interactions.

Full descriptions of each component of the system are given in the following sections.

3.2 User Interface

This project offers a choice between interface designs to improve accessibility. Specifically, it provides 3 views depending on which page the user is on: the landing page before entering the VCS, the spatial map-based view of the VCS, and the static view of the VCS.

3.2.1 Landing Page

The landing page of the application is shown in Figure 3.2 and is the same for every client when they enter the application. It prompts the user to enter information to be displayed once they enter the room such as their name, and their character icon. Once the user is happy with the information they have entered, they will select which view they would like to be displayed to them: spatial or static.

Figure 3.2: The landing page of the application.

3.2.2 Spatial View

The spatial view of the application is designed to give the user a gamified experience of interacting with other users which more closely resembles the real-life experience than classic VCS systems such as Zoom and Microsoft Teams. In this view, the rooms available for entry are displayed to the user as the map displayed in Figure 3.3 with the passages in between the labelled rooms representing the 'Lobby' which is the default room upon entry.

The client and other users of the VCS are displayed as pixel art icons along with their names and will appear in the position/room in which they are currently situated. In order to navigate between rooms, users click the arrow keys to move their character from grid square to grid square. When a client enters a new room, the video and audio feeds of other clients in the new room will appear while those from the previous room will disappear. If other clients are instead using the static view of the application (without a map), they will still be displayed in the correct room on the map view in an unoccupied grid square such that those using the spatial view are still aware of which room these users are in and can join their room if desired.

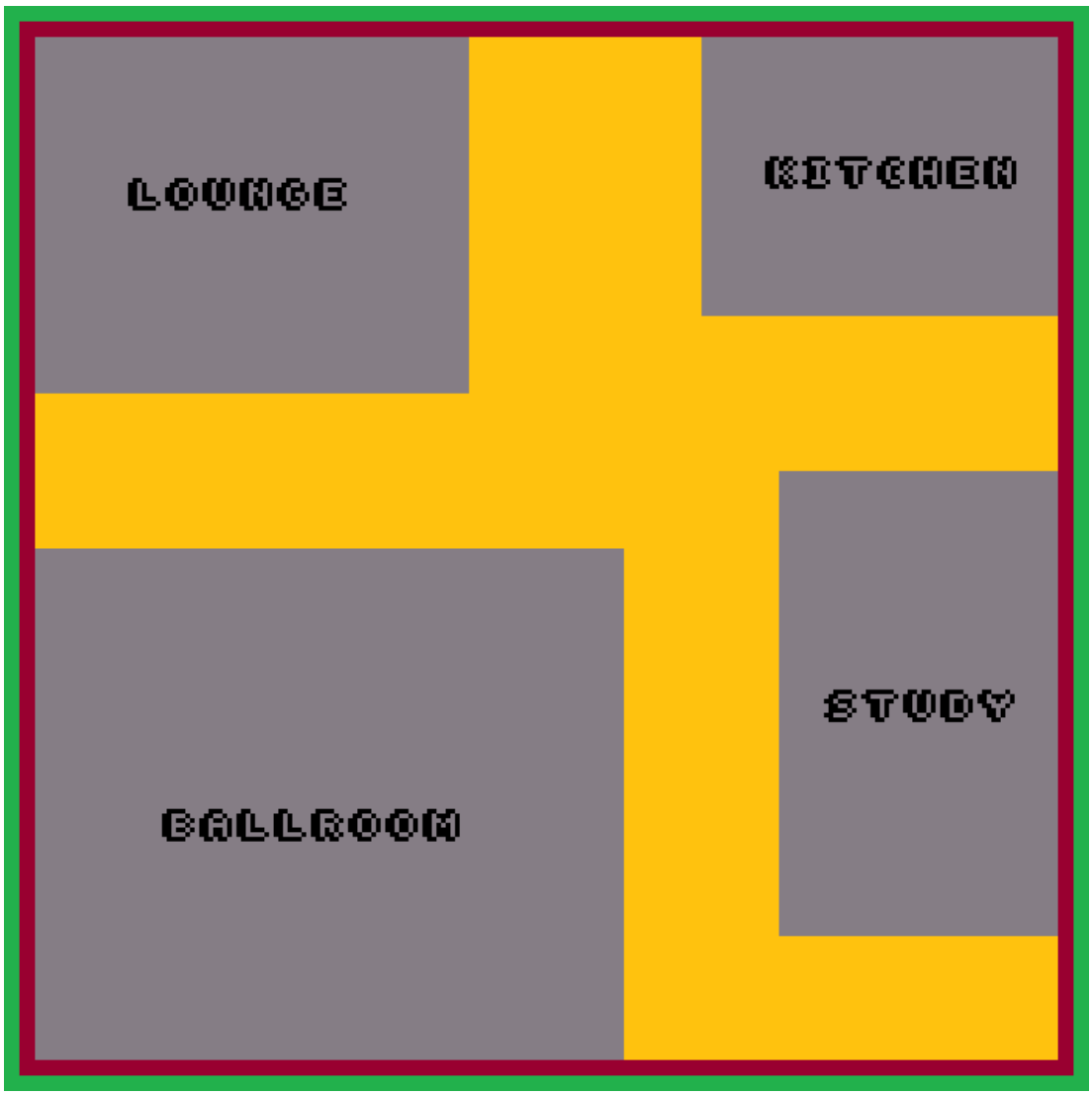


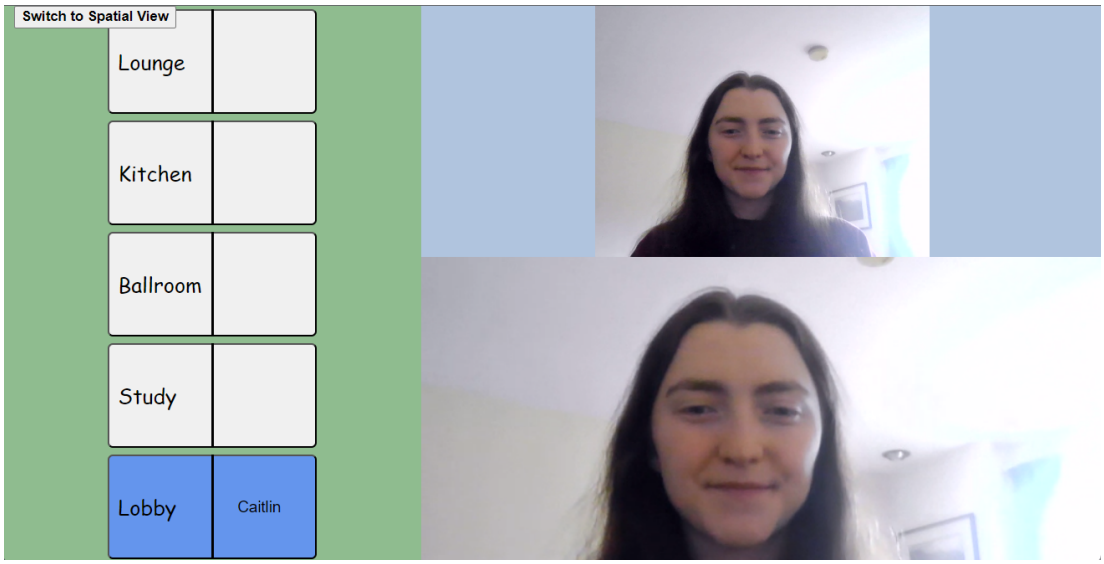
Figure 3.3: Map of available rooms displayed in the spatial view.

3.2.3 Static View

This previously described spatial view works well for engaging users and providing a more natural environment for conversations. However, to those with visual impairments or learning disabilities, this visual-heavy interface may prove overwhelming and discourage them from engaging in social gatherings taking place in a VCS environment. For this reason, the static view of the underlying application instead provides the user with the minimum necessary information to allow them to join calls with other users. This view resembles the classic VCS application such as Zoom and Microsoft Teams, displaying the client's own video along with those of connected clients. Instead of seeing rooms spaced out on a map, the client will simply see the list of available rooms to join on the left of the screen and the video streams on the right. Each possible room selection consists of the room name as well as a list of clients currently occupying that room and the corresponding room selection will be highlighted to demonstrate the user's current room.

For example, if two users enter the application with each selecting a different interface, their respective views of this initial underlying state are as shown in Figure 3.4. In the spatial view, this is demonstrated by displaying the two users' icons and names in the orange lobby area. The user's own video stream is displayed in the top right of the screen with the video stream of the other user displayed below. In the static view, the 'Lobby' room on the left is highlighted as their current room and the other user's name is listed in the 'Lobby' room with the video streams of each user displayed on the right. These static-view users have no concept of which icons other users have selected, how close rooms are to one another or any other map details because these details are unnecessary for them to interact with others and navigate to other rooms.

(a) Spatial view of initial state with two users

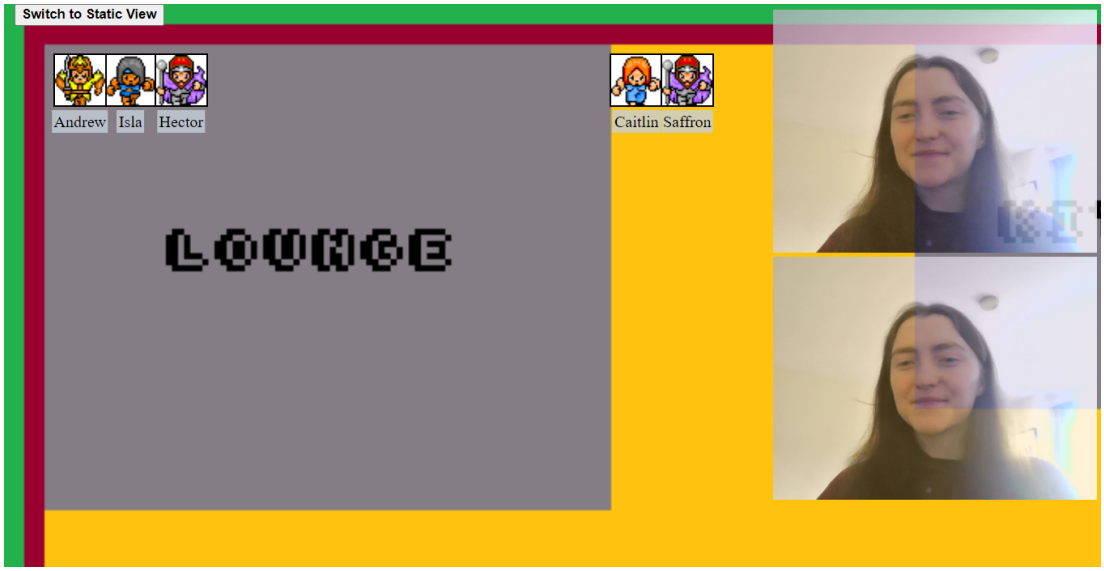


(b) Static view of initial state with two users

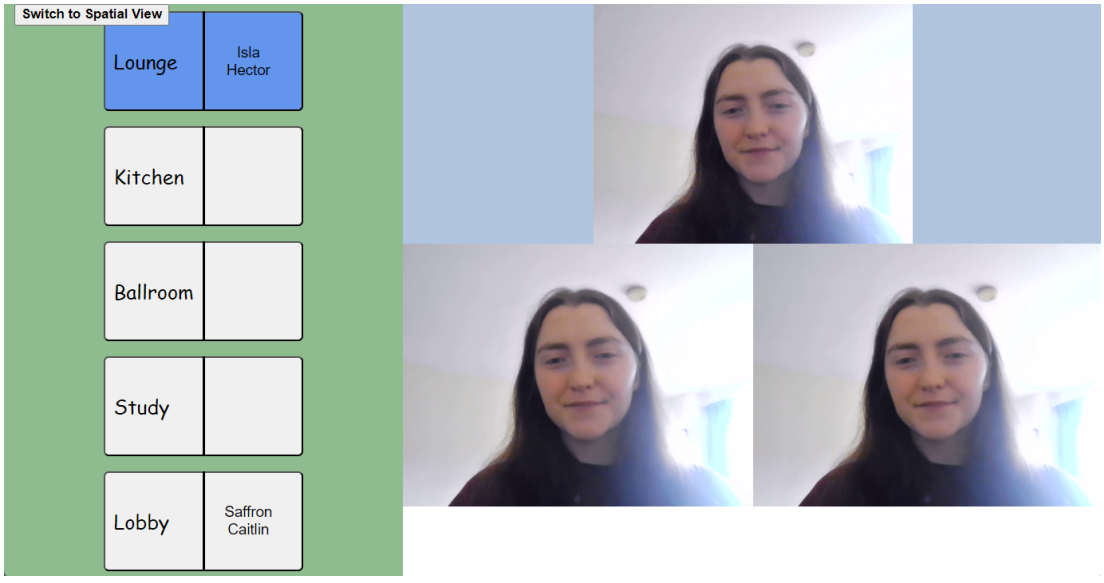
Figure 3.4: Comparison of (a) the spatial view and (b) the static view in the same state with two users having just joined the VCS

.

Figure 3.5 then shows how each interface is displayed upon the static user entering the ‘Lounge’ with other users present on the system. In this state, three users are present in the ‘Lounge’ and two users have remained in the ‘Lobby’. In the static view, the ‘Lounge’ selection is now highlighted to demonstrate that this is the user’s current room along with displaying the names of other users present in their respective rooms. The spatial view also captures the new room associated with the static user as their icon is in the area representing the ‘Lounge’ room. We see here that in both interfaces the users only see their own media and that of the other users in their respective rooms with the static user seeing two other users and the spatial user seeing only one other user.



(a) Spatial view of a user in the ‘Lounge’ connected to two other users



(b) Static view of a user in the ‘Kitchen’ connected to two other users

Figure 3.5: Comparison of (a) the spatial view and (b) the static view, given the same system state with the users residing in different rooms.

3.3 Server

The server is an integral component within the application that maintains the system's overall state. This includes knowledge of which clients are currently accessing the application, the overall grid of squares in which clients may be located on the spatial map, and the location of different rooms on this map. Additionally, although the WebRTC process will primarily depend on direct connections between clients to allow for minimum communication latency, a server is still necessary to act as an intermediary between these clients when they initially set up these direct Peer-to-Peer connections.

3.3.1 Grid Creation

In order to ensure easy interaction between clients accessing the Spatial view and those accessing the Static view, a grid system is used to represent the map of rooms seen in the Spatial view. This grid consists of entries representing a square area of the map with each square holding a pixel location, which room it is a part of, and how many clients are currently on that square. In this way, when a client in the Spatial view moves to a new square in the grid, it is easy to find both the pixel location for displaying the character in the correct position and the room they are in which will be displayed within the Static view. Additionally, when a client in the Static view selects a new room to enter, they can be placed in an unoccupied square in this room if such a square exists. The server is also responsible for updating the grid when clients move in and out of squares to ensure each square holds the correct number of occupants.

3.3.2 Broadcast Client Movements

Clients using the Spatial view need to be aware of where other clients are on the map to allow them to communicate. Instead, those in the Static view need to know which room each client is in. For both of these to work, the server must broadcast any changes in the client's position or room to every other client.

3.3.3 Trigger Client Calls

The server is aware of each room's occupants and is therefore in charge of alerting relevant clients of any changes they should make to their connections when a client enters a new room. In particular, it must signal each client in the new room to begin a call with the entering client as well as alert every client in the previous room to close their connection with this client.

3.3.4 Forward WebRTC Messages

The server must then forward any SDP messages between the connecting clients which will inform them of which addresses they can use to communicate directly with the other client. Once this P2P connection is made, the clients can communicate live data directly but the server must still handle the forwarding of any changes in the ICE candidates available to a client to ensure they can continue communicating smoothly.

3.4 JavaScript Foreign Functions

In order for clients to send and receive live video and audio data, our video conferencing application first requires the ability to identify and access available media devices of the clients. Additionally, the WebRTC framework API is integral to the live communication of this video and audio between clients. However, these functionalities cannot currently be accessed directly from Links.

Fortunately, as discussed in Subsection 2.1.2, we are able to make use of Links' Foreign Function Interface (FFI) which allows custom JavaScript functions to be called directly from the Links code. This allows us to access JavaScript functionality which has not yet been added to the Links library such as media capture and WebRTC API calls.

The primary functionality that our JavaScript library provides to the Links application is to maintain and manage RTC connections between peers. In order to achieve this, the library must make several functions available: capturing local streams from client media devices, setting up new RTC connections and adding any local or remote video streams, creating and receiving offer SDP messages, creating and receiving answer SDP messages, checking for new local ICE candidates, adding new remote ICE candidates, and closing existing RTC connections.

In addition to providing these functions, the FFI also stores relevant information for each open connection the client has with other peers.

3.5 Client

3.5.1 Client Mailbox

When a user accesses the application webpage, they will be assigned a client process which includes a mailbox for receiving and handling messages sent to it by the server process. All functionality provided by the client process will be executed within the client's browser rather than on a separate server system and thus each client maintains only their own state without direct access to any other client's state. In order to share state information with the server and other clients, the client process must exchange messages with the server which will then determine whether the information should be propagated to other clients.

Messages received by the client's mailbox can trigger a variety of functionalities to be executed. These include opening and closing RTC connections to maintain the state of client calls, initiating calls by generating and sending SDP messages to the server which will propagate this to the desired client, handling SDP messages propagated by the server from other clients, and sending newly available ICE candidates to connected clients through the server.

3.5.2 Model-View-Update

So far, the client follows Links' standard Actor-based model. However, as described in Subsection 2.1.2, this model's imperative nature is not ideal for the creation of GUIs

and so our client also makes use of the Model-View-Update architecture shown in Figure 3.6

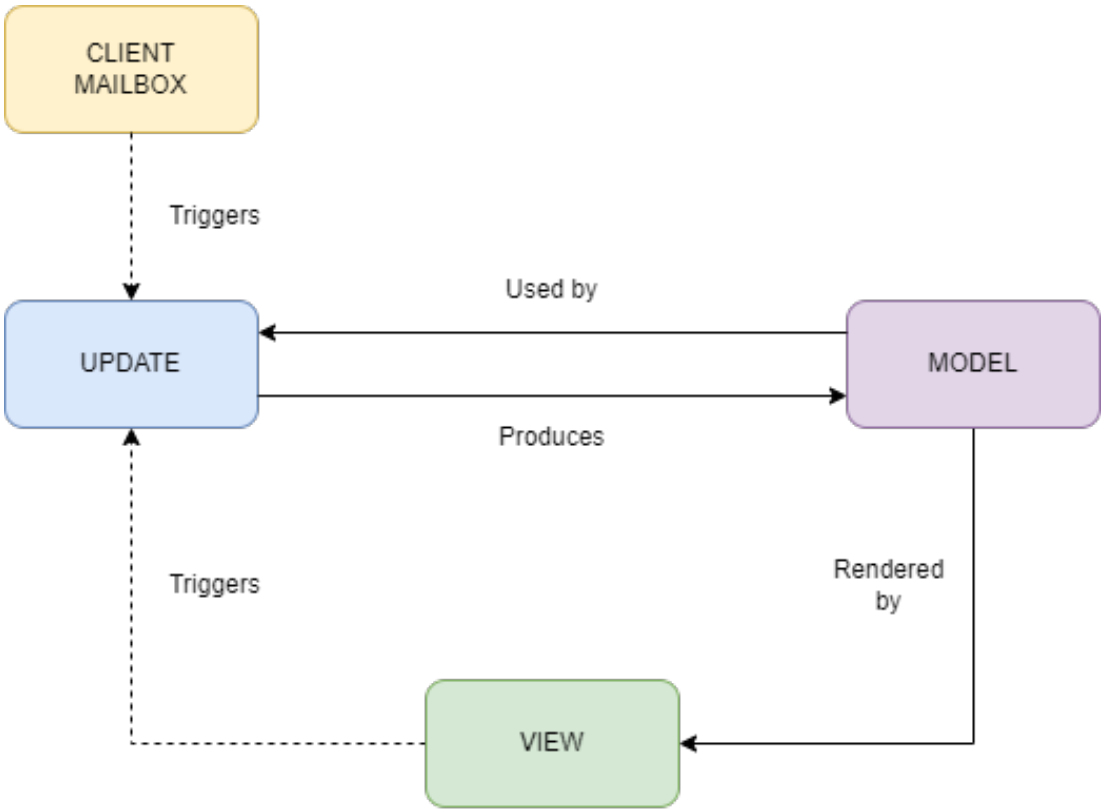


Figure 3.6: Overview of the Model-View-Update pattern along with its interaction with the client mailbox.

This architecture instead allows us to utilise side-effects as exist within JavaScript such that the client's state can be maintained and smooth transitions between models can be made. In our application, the client's model holds its own information and information about other clients received from the server. This information includes the ID, name, grid position, pixel position, and room of each client. The model also tracks the current view we should be displaying to this client (i.e, the landing page, the static view, or the spatial view), as well as a list of rooms and their current members for displaying in the static view.

The Update function accepts messages and updates the state according to the message type and parameters. Our update function has multiple purposes: setting the unique ID of the client as assigned by the server, setting the name and icon of the user once selected, updating the view upon selection and alerting the server that the client has entered this view, handling subscribed events such as pressing the arrow keys to move, updating the rooms and positions of clients when the server signals a change, and responding to requests for its own current position which will then be broadcast to other clients.

However, since the Update function is only accessible by the client through user actions and interactions with the user interface, we need a method of changing the model's state

when it receives messages from the server instead. In order to achieve this, we use the Dispatch functionality offered by Links for MVU-based applications. In this manner, the server sends messages to the client's Actor-based mailbox as usual, and the client itself can then dispatch this message to its model's update function to be processed and update the model accordingly.

The View function of the client will then generate and display an HTML representation of the current model state. Since the application allows different choices of view, the layout of the HTML entities depends on the current view state. This allows us to maintain almost exactly the same underlying model and update functionality for each system, with different high-level representations of this state observed by users.

Chapter 4

Implementation

4.1 Initial States

4.1.1 Server

Upon the initial deployment of the application, a process is spawned in the server to maintain the system's overall state and exchange messages with any connected clients. This server is initialised with an empty grid of squares, an empty list of connected clients, and a complete list of available rooms. This list of rooms includes information about which squares they will occupy and an initially empty list of members who are currently within them.

Before the interface is exposed to users, the Server is first instructed to begin creating the underlying grid based on the defined rooms and their positions. The underlying map is of size 1400 x 1400 pixels and so the underlying grid consists of 784 squares each of size 50 x 50 pixels. To create the grid, a square is created at each column and row index with the following type:

```
typename GridSquare = (
    tpos: Float,
    lpos: Float,
    room: String,
    numOccupants: Integer
)
```

Each square holds its placement in terms of distance from the top and left of the webpage, its designated room, and the number of occupants on this square (initialised to 0). The room associated with the square is found by checking whether the current index falls between the indices associated with the edges of any defined rooms. If not, the square is said to belong in the 'Lobby' which is the space on the map which connects other rooms.

Once the grid creation is completed, a route to a function which generates the main page of the application is added to the server. Static routes to other sources such as the style sheets, JavaScript files, and supporting media are then also added. The server

must then initialise web sockets which will allow two-way communication between the server's mailbox and any connected clients. Finally, the server will begin serving the pages from the added routes, allowing users to access the application.

Overall, the initialisation of the server is achieved as follows:

```
fun main() {
  serverPid ! CreateGrid;
  addRoute("/", fun(_) {mainPage()});
  addStaticRoute("/images", "images", [("png", "text/plain")]);
  addStaticRoute("/images", "images", [("gif", "text/plain")]);
  addStaticRoute("/css", "css", [("css", "text/css")]);
  addStaticRoute("/js", "js", [("js", "text/javascript")]);
  serveWebsockets();
  servePages()
}
```

4.1.2 Client

Each time a user enters the application, a new client process is spawned with a mailbox for exchanging messages with the server. This process starts by creating a new MVU handler with an initial model, an update function, a view function, the ID of the HTML element in which the result of the view function will be placed, and a subscription function which will listen for specified events such as onKeyDown events in our case.

The initial model state is set to the following:

```
var initialState = (
    currentView="Options",
    myPosition=(
        id="None",
        name="Nameless",
        xind=0,
        yind=0,
        room="Lobby",
        xpos=0.0,
        ypos=0.0,
        icon="None"
    ),
    rooms = [
        (name="Lounge", members=[]),
        (name="Kitchen", members=[]),
        (name="Study", members=[]),
        (name="Ballroom", members=[]),
        (name="Lobby", members=[])
    ]
)
```

Here, the client's id and positions are originally unset as these are later set by the server.

The client's name and icon are temporarily set to default values as the user will enter these on the first page. The list of rooms contains each room's name along with an initially empty list of members.

The current view of the model is initially set to 'Options' and this property is used by the view function to determine that the initial options menu should be displayed to the user. The 'Options page' allows the user to enter their name and select an icon to be displayed as their character in the spatial view. This is also the stage at which they have the option to choose whether they wish to enter the static or spatial view when they join the video conference.

Once the MVU handler is set up, the client can then be registered with the server by sending a 'Register' message to the server along with the client's process identifier (PID) such that the server can identify it. Here, the server will assign to the client a unique numerical ID converted to a string. This ID will be used by clients to identify one another and the server uses the associated PID to determine which process messages should be forwarded to when only the string ID is specified. The server will then add the mapping between the PID and its corresponding ID to its list of connected clients. The server will also send a 'SetMyId' message to the client PID with the generated ID as an argument which the client will dispatch as a 'SetId' message to its update function which will in turn set the model's own client ID to this value.

Upon entry to the application, the JavaScript Foreign Function Interface of the client is also initialised with an empty dictionary ready to store data about any open connections with peers. The property representing whether there is a local video stream associated with the client is initially set to 'False' such that the client can wait for it to become available before registering to the VCS. This registration will be described further in Section 4.2

4.2 Entering a VCS View

4.2.1 Client Updates

Once a user selects a view preference, this will send an 'EnterView' message to the client's update function. This message causes the update function to first ask the FFI to get the user's local media stream and, before proceeding, the client repeatedly queries the FFI until it discovers that the media stream has been fetched successfully as follows:

```
fun awaitVideoLoop(){
    if (WebRTC.isLocalVideo()){
    }
    else{
        awaitVideoLoop()
    }
}
```

Meanwhile, the FFI will make requests to any user devices for any media which captures audio or video like so:

```
navigator
    .mediaDevices
        .getUserMedia(constraints)
            .then(function success(stream) {
                setLocalVideo(stream);
            })
```

Upon success, it will add the video track to the HTML node containing the user's local video, and it will add both the video and audio tracks to a media stream which will be sent to any connected clients. Finally, it will set the local media flag queried by the Links client so that the client may continue its entry to the VCS.

Now that the user's local media has been captured, the client needs to signal to the server that it would like to be added to the VCS. Thus, the client sends a 'RegisterInitialClientPosition' message to the server along with the client's own ID and the name and icon chosen on the options page. This step is also independent of the user's chosen view as the server's actions and the client's underlying MVU model will be the same.

Finally, while the server is processing this message, the update function will output a new model which consists of the previous client model with the view state changed to the option selected by the user. This state change will cause the view function to emit the newly selected interface to the user.

4.2.2 Server Registration Process

Upon receipt of this 'RegisterInitialClientPosition' message, the server will register the client to the VCS as follows:

```
case RegisterInitialClientPosition(id, name, icon)->
    var newPos = getInitialGridPos(grid, id, name, icon);
    var newGrid = updateGridSquareOccupants(
        grid, -1, -1, newPos.xind, newPos.yind, 0);
    var newPid = getPid(id, pids);
    var currRoom = "None";
    manageCalls(newPid, id, name, pids, currRoom, newPos.room);
    broadcastCharacterMovement(
        pids, newPos, id, name, currRoom, newPos.room);
    connectionServer(newGrid, pids, rooms)
```

The server will start by finding the initial position and room information for the client. It does this by iterating through all the squares in the grid until it finds an unoccupied square whose room is set to 'Lobby'. Once such a square is found, its pixel position and room are extracted along with the indices in the grid at which it was found. The server temporarily stores the client's information along with its new grid indices, pixel positions, and room name.

In the case that the server has other clients already connected, it then initiates a call between the new client and those clients also residing in the 'Lobby'. A full description of how calls between clients are managed by the server is given in Section 4.4. Similarly,

the server sends the new client's position and information to all connected clients and asks for their information which it can then forward to the incoming client. The process of communicating changes in a client's position is described in more detail in Section 4.3.

The server will lastly update the state of the grid to reflect the addition of this new client by iterating through the grid and replacing the square with a new square which has its number of occupants incremented. This completes the server registration of the client.

4.2.3 Resulting Interface

Now that the client is registered to the VCS and the view state has been changed, a new interface is shown to the user representing the current state of the VCS. Common to both views is a button which allows the user to switch which view of the system they currently see. When a user selects the button, a 'StateChange' message is sent to the client's update function along with which view the user would like to switch to ('static' if they are currently in the spatial view and vice versa). The update function will then output a new model with the view state updated and the view function will begin outputting its HTML as defined by the new view state instead of the old state.

When the view state is set to 'Spatial', the view function will start by setting the 'className' property of the video container to 'spatial'. This allows the CSS rules for the video nodes in a spatial environment to take effect. The CSS rules for the spatial view video elements are shown in Figure 4.1a.

In order to display the character icons and names in the correct map positions, the 'getCharacterHTML' function is used on each character. In the case of the client's own character, this function is called with the model's 'myPosition' information as an argument. In the case of other clients, it is called on each member of the model's 'othersPositions' list. In both cases, the 'getCharacterHTML' function can then use the xpos, ypos, icon, and name properties of the member to create nodes with their left, right, background-image, and text properties set respectively.

These character nodes are added on top of a background image representing the map along with the 'Switch to static view' button whose onClick property sends the 'StateChange' message described above.

If the view state is set to 'Static', the view function will instead set the 'className' of the video container to 'static' such that the correct CSS rules for video elements are utilised. The static view CSS rules for videos can be seen in Figure 4.1b.

The static view does not need to generate a background image or characters as seen in the spatial view. Instead, it iterates through each room stored in the model's 'rooms' state creating buttons for each with their onClick property set to send a 'ChangeRoom' message to the update function. Within each button is stored the room's name and a list of nodes representing the name of each client in that room which is generated by iterating through the 'members' property of the room. Finally, the 'Switch to spatial view' button is created as in the spatial view function and this is output to the user's web page along with the previously created room nodes.

```css
.spatial #videoContainer{
    right: 1vmin;
    top: 1vmin;
    position: fixed;
    height: 99vh;
    overflow: hidden;
}

.spatial #remoteVideo{
    width: fit-content;
    max-height: calc(100% - 240px);
    overflow-y: auto;
    display: grid;
}

.spatial #remoteVideo video{
    width: 320px;
    height: 240px;
}

.spatial video{
    opacity: 0.8;
}
```

(a) CSS rules for the Spatial View

```css
.static #videoContainer{
    position: fixed;
    left: 400px;
    top: 0px;
    height: 100vh;
    width: calc(100vw - 400px);
}

.static #localVideoContainer{
    background-color: lightsteelblue;
}

.static #localVideoContainer #localVideo{
    margin-left: auto;
    margin-right: auto;
    display: block;
}

.static #remoteVideo{
    display: grid;
    grid-template-columns:
        repeat(
            auto-fit,
            minmax(350px, 1fr)
        );
}

.static #remoteVideo video{
    width: -webkit-fill-available;
    max-height: calc(100vh - 240px);
}
```

(b) CSS rules for the Static View

Figure 4.1: Side-by-side comparison of the video-related CSS rules for (a) the spatial view and (b) the static view

.

4.3 Client Movement

4.3.1 Movement in Spatial View

While accessing the spatial view of the VCS, users are able to navigate the map of rooms by using the arrow keys on their keyboard. This is possible due to the MVU handler's initialisation with a subscription module which is subscribed to the onKeyDown event emitted by the browser when it registers the user pressing down a key. When an event of this kind is triggered, the subscriptions function sends a 'MovementEvent' message to the MVU update function along with the code associated with the specific key which triggered the event.

Upon receipt of this message, the update function first checks that the view state of the model is 'Spatial' since keypresses in other views should not cause any changes to the system's state. The function also checks to ensure that the code associated with the pressed key is a valid code for causing movement: "ArrowDown", "ArrowUp",

"ArrowLeft", or "ArrowRight". If so, a 'CharacterMoved' event is sent to the server process such that it may update the system as described further in Subsection 4.3.3. This message is sent along with the keypress code and the user's current model information such as ID, name, x/y index, room, and icon.

4.3.2 Movement in Static View

While accessing the static view of the VCS, users instead navigate between rooms by selecting the room they would like to join from a list of available rooms. Each of these room buttons has its 'onClick' property set such that it will send a 'ChangeRoom' message to the client's MVU update function along with the associated room name. Unlike the spatial view, this will instead send a 'CharacterChangedRoom' message to the server process as described in Subsection 4.3.4. It includes similar information in the message except that it sends the desired room rather than the key code.

4.3.3 Server Response - Spatial

Once the server receives a 'CharacterMoved' message it follows the following steps:

```
case CharacterMoved(currRoom, ...) ->
    var newPos = getGridPos(...);
    var newGrid = updateGridSquareOccupants(...);
    broadcastCharacterMovement(...);
    if (newPos.room <> currRoom){
        manageCalls(...);
        connectionServer(newGrid, ...)
    }
    else{
        connectionServer(newGrid, ...)
    }
```

It will first increment or decrement the user's current x/y indices based on the given key code to get the new grid position after moving in the given direction. The information in the new grid square is used to find the new pixel position and room of the client and the new information is stored ready to be broadcast. The grid is then updated to reflect the change in the number of occupants in both the previous room and the new room.

With this new character information, the server will then begin to broadcast this to all of its connected clients, including the moving client. To the client with an ID matching that of the moving client, it will send a 'Moved' message to its PID along with the position information indicating that it should update its own position. To each other connected client process, it will send an 'OtherMoved' message with the new information to indicate that the client should update its information about the moving client.

Lastly, if the new room does not match the old room, the server will end any calls the moving client has with those clients in the previous room and begin calls with those in the new room. This process is handled by the 'manageCalls' function which will be described fully in Section 4.4.

4.3.4 Server Response - Static

When the server receives the 'CharacterChangedRoom' message from a client in the static view, it follows the following steps:

```
case CharacterChangedRoom(currRoom, newRoom, ...) ->
    if (currRoom <> newRoom){
        var newPos = getNewRoomPos(...);
        var newGrid = updateGridSquareOccupants(...);
        manageCalls(...);
        broadcastCharacterMovement(...);
        connectionServer(newGrid, ...)
    }
    else{
        connectionServer(newGrid, ...)
    }
```

It first checks that the new room selected is the same as the current room of the client and if so, no changes should be made to the system.

If the client has entered a different room, then the server will begin by trying to allocate the client an empty square in the new room. It does this by iterating through all the squares associated with the new room until it finds an unoccupied square. Once such a square is found, its pixel position and room are extracted along with the indices in the grid at which it was found. Otherwise, the search will restart and look for a square with a single occupant instead and this process repeats until a suitable square is found. The server temporarily stores the client's new information ready to be broadcast. The grid is then updated to reflect the change in the number of occupants in both the previous room and the new room.

The rest of the server's response is then the same as it gives upon a movement to a new room in the spatial view. First, the new client position and room changes are broadcast to all clients registered to the VCS including the moving client itself such that their model states may be updated. Then, the server will end any calls the moving client has with those clients in the previous room and begin calls with those in the new room as handled by the 'manageCalls' function which is described in Section 4.4.

4.3.5 Client Updates

The messages sent by the server to the client when the system state has changed are the same regardless of whether the client is in the spatial view or the static view. Thus the following functionality is common to all clients connected to the VCS.

When the client's mailbox receives a 'Moved' message from the server along with its new position, it needs to dispatch this message to its MVU handler such that its model can be updated. This is because only the MVU's update function can modify the model in an MVU architecture. Thus, the client reacts to this message by calling the 'dispatch' function provided by Links and passing it an 'IMoved' message with the new positions attached along with a reference to the client's MVU handler whose Update function

should receive the message. The ‘IMoved’ message will cause the update function to output a new model with its ‘myPosition’ state set to the newly received information which includes the new indices, pixel positions and room of the client.

Similarly, when a client mailbox receives an ‘OtherMoved’ message from the server, it will dispatch this as an ‘OtherMoved’ message to the MVU update function. Here, it will iterate through the list of rooms known by the client and if the room name matches the previous room of the moving client, then it will remove the given client from the member list of this room. Similarly, if the room name matches the client’s new room, then the client’s new information will be added to the member list for this room. A new model is then output with the ‘rooms’ state updated to reflect these changes.

4.4 Call Management

When the server determines that two clients should enter a call with one another, a variety of messages must be exchanged between the server and the two clients for them to establish a direct peer-to-peer WebRTC connection. A sequence diagram summarising the flow of these messages is shown in Figure 4.2

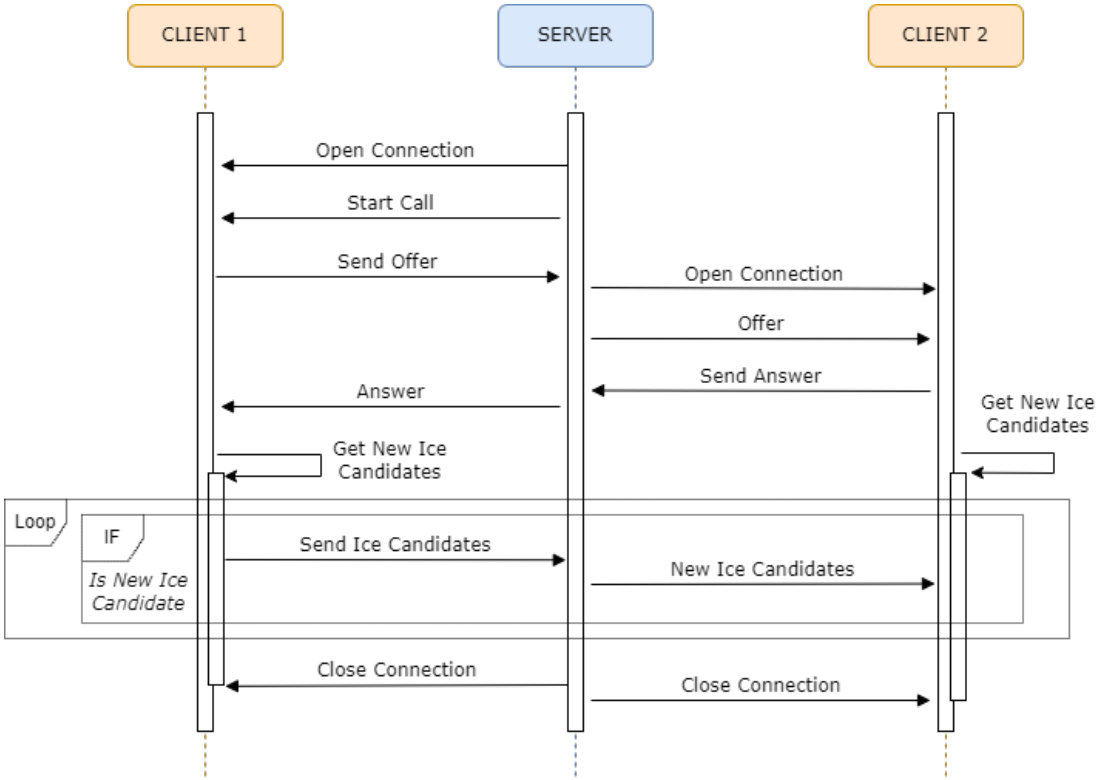


Figure 4.2: A sequence diagram of messages sent to establish a WebRTC connection.

The ‘manageCalls’ function is called by the server when a client initially enters the VCS or moves from one room to another. This function is responsible for broadcasting messages to clients alerting them of any changes they should make to their RTC connections as shown below.

```
fun manageCalls(pid, id, name, currentPids, oldroom, newroom){
    switch (currentPids) {
        case [] -> ()
        case p::ps ->
            var otherPid = first(p);
            if (second(p) <> id && third(p) == newroom){
                otherPid ! OpenConnection(pid);
                otherPid ! StartCall(pid)
            }
            else if (second(p) <> id && third(p) == oldroom){
                otherPid ! CloseConnection(pid);
                pid ! CloseConnection(otherPid)
            }
            else{};
            manageCalls(pid, id, name, ps, oldroom, newroom)
    }
}
```

To clients in the new room, the server will send an 'OpenConnection' and 'StartCall' message with the moving client's PID. For each client in the moving client's previous room, the server will send the moving client's PID in a 'CloseConnection' message to the other client as well as sending a 'CloseConnection' message to the moving client with the other client's PID as the connection must be closed on both ends.

4.4.1 Opening a Connection

When the client receives the 'OpenConnection' message from the server, it needs to set up a new RTC connection in preparation for a call with another client. Since this requires the use of the WebRTC API, the client will delegate this task to the JavaScript FFI by calling its 'setupRTCConnection' function which handles this request as follows:

```
function _setupRTCConnection(pid){
    connection = new RTCPeerConnection(peerConnectionConfig);
    connection.onicecandidate =
        event => newIceCandidate(event, pid);
    connection.ontrack =
        event => newRemoteTrack(event, pid);
    connection.addStream(localVideo);
    peerConn = {
        connection: connection,
        iceCandidates: [],
        isNewIceCandidate: false,
        remoteElement: null,
        hasRemoteAudio: false,
        hasRemoteVideo: false,
    };
    peerConnections[pid._clientPid] = peerConn;}
```

The ‘setupRTCConnection’ function will start by creating a new WebRTC connection ready to connect the local client to a remote client and maintain this connection. Relevant events that the connection listens for are each assigned a function which should be called when the respective event is triggered. These events include the discovery of a new ice candidate for the local client and the receipt of a new media track from the remote client. Finally, the local video stream stored during the client’s initial entry into the VCS is added to the connection ready to be sent to the remote client once the connection is complete.

This connection is then added to a dictionary of information to be stored about the current connection. Other properties added to this dictionary include an empty list of remote ice candidates, an initially null reference to the HTML node which will display the media for the remote client, boolean flags representing whether remote audio or video have been found, and a final boolean flag demonstrating whether the local client has any new ice candidates to send to the remote client. Finally, the dictionary of information about this connection is then added to a full dictionary of all connections the local client has with remote clients, with the key set to the remote client’s PID.

4.4.2 Call Initiation

When the client receives the ‘StartCall’ message, it will first task the JavaScript FFI to create an offer Session Description Protocol by calling its ‘createOfferSDP’ function. As the FFI function runs concurrently with the Links client, the Links client must repeatedly query the FFI to ensure that the local session description for the current connection has been set before proceeding. Meanwhile, in the ‘createOfferSDP’ function, the FFI will tell the RTC connection to create an offer which will include the ICE candidates available for the remote client to connect to the local client and a request for both audio and video media streams.

Once the local offer has been successfully created, the Links client will retrieve it from the FFI and send the offer to the server in a ‘SendOffer’ message along with both the local and remote clients’ PIDs so that the server can forward this to the desired client. Here, the server will first send an ‘OpenConnection’ message to the remote client as before, followed instead by an ‘Offer’ message which includes the offering client’s PID along with the SDP offer.

4.4.3 Receiving an Offer

The client’s response to receiving an ‘OpenConnection’ message has been described in Subsection 4.4.1 and after setting up this RTC connection, the receiving client will then process the ‘Offer’ message from the server.

First, the Links client will deliver the SDP offer it received from the initiating client to the FFI for processing by calling its ‘receiveOfferSDP’ function. The Links client will then await the completion of this function by repeatedly querying the FFI to ensure the remote description on the connection has been set. Meanwhile, the FFI will set the remote description of the previously created connection to the received SDP. As a result,

the connection now knows which addresses are available for direct communication with the peer and the type of media streams that have been requested.

Once the offer has been properly received, the Links client will continue by spawning a new process to check for and handle any new local ice candidates which become available. This process is described fully in Subsection 4.4.4. Then, the client will request the FFI to create an answer SDP and await its creation. The FFI will use the information stored in the connection about the remote offer to find suitable media and server addresses and if it is happy with the offer received, it will create an SDP with its return offer. The Links client will then fetch this SDP and send it in a 'SendAnswer' message to the server which will then forward the response to the initiating client.

When the initiating client receives the answer, it will follow a similar process. It will set the remote description of the connection to the received SDP and then spawn a process to handle any newly available ice candidates. Now, the connection is complete and the peers will begin communicating their available media directly over the WebRTC connection without going through the server.

4.4.4 Updating ICE Candidates

Upon a new ice candidate becoming available to a local client, the FFI function associated with the connection's 'onicecandidate' event is called. This function will place the new candidate in a list of newly available local ice candidates related to the connection and set the 'isNewIceCandidate' flag.

While two peers are connected, they both have a process running which will check for and send any new local ice candidates such that any changes do not prevent the peers from communicating smoothly. The function run by this process is as follows:

```
fun getNewIceCandidates(mypid, serverPid, pid){
    switch (WebRTC.getIsNewIceCandidate(pid)){
      case "True" ->
        var candidates = WebRTC.getNewIceCandidates(pid);
        serverPid ! SendIceCandidates(mypid, pid, candidates);
        getNewIceCandidates(mypid, serverPid, pid)
      case "False" ->
        getNewIceCandidates(mypid, serverPid, pid)
      case "End" ->
        ()
    }
}
```

This process will repeatedly query the FFI to check if any new candidates have been collected and whenever this is true it will then ask the FFI to respond with one of these new candidates. The FFI will then remove and return the first candidate in the list, and once the list of new candidates is empty, it will unset the 'isNewIceCandidate' flag so that the Links client will stop fetching candidates until more new candidates are added.

When the Links client receives a new candidate from the FFI, it sends it in a 'SendIce-

Candidates' message to the server along with its own PID and that of the destination client PID. Once forwarded by the server, the other client will instruct its FFI to add this to the current list of remote ice candidates associated with this connection.

4.4.5 Closing Connections

When clients receive a 'CloseConnection' message upon a client's exit from a room, they will relay this to their JavaScript FFI by calling its 'closeRTCConnection' function. This function will close the connection and remove the HTML node associated with the remote client's media stream. It will then delete the connection dictionary such that this connection no longer exists.

To clean up the spawned processes for collecting new ICE candidates in the Links client, when the FFI is queried to find whether there are new ice candidates and the connection queried does not exist, it will return an "End" message causing the process to complete.

Chapter 5

Evaluation and Discussion

5.1 User Experience

The presented video conferencing system allows users to communicate live video and audio with other clients in a variety of rooms and, as designed, they can choose whether they wish to enter a spatial or static view of the system.

5.1.1 Spatial View

For the benefit of users who prefer a gamified experience when using a VCS, we have successfully delivered a spatial view of our underlying system. This view offers a similar interface as that of Gather.town [12] with rooms displayed on a visual map. In our case, each room on the map represents a call between all the individuals in that room and as such, each user is able to hear and see the video feeds of any users in their current room. Users can easily navigate from room to room using the ArrowKeys on their keyboard to move up, down, left, and right as they would do in most character-based games they may be familiar with. This interface closely resembles how interactions and meetings in real-life social spaces work in which people move to separate rooms to have meetings or private conversations with a subset of people.

An unfortunate drawback of using only arrow keys to move around is that users accessing the application on a mobile device cannot use the spatial view functionality since they do not have access to arrow keys. This could be included through the addition of buttons representing the arrow keys or a virtual joystick as is used in mobile games.

Unlike Gather.town, our application does not currently allow for interactions between users based on the distance between their characters. This is because we prioritised the support of interactions between users on both views and therefore used the simplest underlying technical implementation to support this. Therefore, even if two users are far away from each other on the map, they will still have a call between them if they are in the same room. This is particularly evident in the ‘Lobby’ which represents any space between all the other defined rooms and therefore encompasses a variety of areas across the map.

5.1.2 Static View

For those users who prefer a minimalistic VCS interface, we have successfully offered a static view of the underlying system. This is more similar to the traditional VCSs which most people are familiar with and therefore should be easy to get used to. Contrary to traditional systems, the user still has access to all the rooms available to those in the spatial view but has no concept of an underlying map or characters' positions on this map. The rooms are instead displayed in a list along with their current members from which the user can select their desired room.

5.1.3 Cross-View Interactions

From the perspective of spatial view users, when other clients on the spatial view move around, their characters will move from square to square accordingly and their media will appear and disappear as the character enters or exits the user's current room. However, when other clients in the static view move from room to room, their character will appear in the new room in one move and so they will appear to 'teleport'. Although this makes the movement seem less natural, it is necessary to ensure that the understanding of the state of the system is consistent between all users on either interface. If each static view user were placed on the same square, it would become impossible to see most of the users in the room. Thus, when this 'teleportation' occurs, the users are placed in an unoccupied square if one exists which proved an effective solution to this problem.

From the perspective of the static view users, they will only see room changes which will be displayed in the same way regardless of whether the other client is in the static or spatial view. Thus, they will not see any intermediate movements of the spatial clients until they enter a new room, at which point their name will be removed from the list of members in their old room and added to the list of members in the new room.

If the user changes their mind about which interface they would like to see, all they need to do is click the 'Switch to Other View' button which will change their interface to the respective layout of the other view. This ensures users are not locked into their original choice if they selected the wrong option and gives fluidity between views while maintaining the correct view of the underlying system state.

5.2 Technical Evaluation

Through a variety of design choices, we have effectively developed a multi-view video conferencing system using the Links programming language.

5.2.1 MVU

We found that using Links' built-in MVU architecture proved very convenient for maintaining the same underlying system state with alternate views built on top for displaying this state. As hoped, we found that the main aspect of the code which had to be tailored to enable both interfaces was the view function itself, with the client model

remaining the same. This is well highlighted by the view-switching functionality which only changes the selected view in the model in order for them to see the entirely new interface with the system's state maintained.

However, since the movement of characters from room to room differs between views, it was found that some additional messages accepted by the clients' Update function and server had to be added. Fortunately, the additional code required was minimal thanks to the use of the underlying grid structure.

5.2.2 The Grid

This underlying grid structure used to represent locations on the map proved to offer a very simple method of supporting interactions between users on both views. For clients in the spatial view, the server only needs to fetch the next square in whichever direction the client is moving. For those in the static view, the server must search for an appropriate square in the grid to place the user when they enter a room. This simply involves iterating through each square within the known indices of the new room to find an unoccupied square.

Finding such a square in the 'Lobby' proves less efficient as it does not have simple rectangular borders within whose indices we can check for empty squares. However, thanks to each square in the grid storing the name of the room in which it occurs, we can still iterate through each square in the grid until we find an unoccupied square which has its room set as 'Lobby'.

Additionally, Links does not offer an inbuilt method for efficiently replacing a single item in a list. As a result, each time we wish to update the number of occupants on a grid square, this involves iterating through the entire grid to find the desired square and replacing it. This results in a lot of computation on the server if many clients are moving around and is likely to contribute to the lag experienced in the application with multiple users present. The results of these updates are only used sparsely, either when a client newly enters the VCS or when a static view user changes their room. Therefore, it may prove more efficient to instead compute square occupancies only when this information is actually needed to prevent wasted computation.

5.2.3 Interfacing with WebRTC

It was found that using the JavaScript Foreign Function Interface as outlined in the design section enabled the Links code to interface with the WebRTC API functions facilitating real-time peer-to-peer communication between clients. However, some of the FFI functions made use of JavaScripts asynchronous functionality for which Links does not currently have an appropriate method of handling. As a result, each time our application needs to await the completion of a JavaScript function, we must spawn a process which continuously checks whether the FFI function has finished. Having many such processes running creates a lot of processing overhead but due to the lack of other more efficient await functionality, this proved unavoidable.

5.3 Testing

The nature of this project allows for primarily qualitative evaluation but it is also important to ensure that the system can handle multiple users. This can be evaluated without rigorous benchmarking as it is a trivial case of adding more users until the system fails. Thus, we have tested the system both locally and remotely to explore how the system scales in each case as can be seen in papers of a similar nature [4, 21].

5.3.1 Local Testing

In order to test that the application worked as expected, the server was first run locally with separate tabs used as connecting clients.

It was found that the application could easily handle four connected clients, displaying both video and movement data with little to no delay. However, after connecting five or six clients at once, the movement and video data began to lag significantly, with seven connected clients causing the application to become unusable.

The delay seen in the video stream transfer makes sense while running the system locally. This is because, for each connected client, the same device must hold endpoints for sending and receiving data at both ends of the connection. With five active users, this amounts to five clients connecting to four other clients and thus, a total of $5 \times 4 = 20$ or $n(n-1)$ open connection endpoints.

The delay in character movement display could have been due to either the server or the client being overloaded. However, it was discovered that when a client moved to a separate room with a small number of other clients, the delay in character movement display was reduced significantly. This demonstrated that the number of client WebRTC connections must be the largest contributing factor to the delay even though the client movement is not sent over WebRTC. Since very few messages are sent through the server regarding ongoing WebRTC connections, this further implies that the issue lies within the client itself. In particular, the processes spawned per connection to repeatedly query the FFI for any new ICE candidates.

This discovery prompted the insertion of a short delay between each request to the FFI such that the client can spend more time servicing other messages such as character movement messages rather than continuously sending messages to the FFI which rarely requires action. This alteration showed significant improvement in the display time of character movements.

Local testing also revealed that when a client exits the application by refreshing the page or closing the window, they are not removed from the system. As a result, their video stream to other users will freeze but will not be removed and their icon and name will continue to be displayed as if they are participating in the system. This highlighted that the server would require an alerting mechanism such that when a client has left the system it can be removed from the system and any relevant connections can be closed.

5.3.2 Remote Testing

Remote testing was carried out using two devices connecting to the application. It was observed that there is some delay before the remote video is displayed upon a new connection being initiated but this delay is short when there are few connected clients and grows slightly as more clients connect. This is likely due to there being more concurrent processing occurring on the client with each new connection as well as increased message traffic through the server.

As anticipated, more members can connect to the application without causing a significant lag in the video streams since there are fewer connections maintained on a single device. Unfortunately, we could not test exactly how many clients could connect without lag as we did not have enough devices available. However, we found that with two devices we could now connect six clients without experiencing significant lag and this suggests that with more devices to spread out the connections, we could connect more.

Chapter 6

Conclusions

6.1 Achievements

In conclusion, we have presented a system which successfully uses the Links programming language to produce a video conferencing application. This system not only allows the exchange of live video and audio data but also gives users a choice of interface for interacting with this system.

The system offers a static interface similar to the traditional VCSs used in everyday life, with the addition of a list of rooms available to join. The system also provides a gamified spatial interface which gives the user an experience more synonymous with real-life conversations in which the user navigates around a map of available rooms which represent each group conversation.

These interfaces were implemented on top of the same underlying system state using a model-driven design approach which also allowed us to offer a switch between the displays at any point during the application. We found that this model-driven approach was made particularly easy to implement using the built-in Model-View-Update architecture supplied by Links.

We found that the system can successfully connect to and utilise the WebRTC API which we accessed through the JavaScript Foreign Functions Interface provided by Links. Additionally, this interface allowed us to successfully capture media from the user's devices.

Through testing the system on a single device, it was found that up to five users can successfully connect without experiencing significant issues with the video streams and using multiple devices allowed even more users to connect.

6.2 Future Work

The evaluation of the project highlighted some aspects of the system which could be improved. One such aspect is the method by which the system places users in unoccupied squares of the grid. Currently, this requires the system to update the entire

grid every time a user's character moves such that it reflects the occupancy of each square. Instead, this process could be modified such that the system calculates the occupancy of squares only when this information is required such as when a new user joins the VCS or when a static interface user selects a new room.

Additionally, the spatial interface does not yet take into account the proximity of players to one another as priority was given to developing the simplest implementation of cross-view interactions. Having accomplished this basic implementation, a promising future direction for this project would involve adding support for proximity-based interactions. With only the spatial interface this would be relatively simple as it would simply involve checking for users within adjacent squares and connecting to them. However, representing these proximity-based conversations in the static view presents a challenge. A potential method which could be explored is to add any of these proximity-generated calls to the list of rooms available to join. Then, upon joining the conversation, the user's character position in the system will be updated to place them within the proximity of these users.

The system is currently unable to handle users exiting the application by refreshing the page or closing the window. Thus, an important future addition to the application would be to interface with the 'monitorConnection' API which was recently added to the standard Links library. This API signals when a client becomes unresponsive and has therefore left the application. When this occurs, we would instruct the server to remove the client from its state and wrap up any ongoing connections.

Another future improvement to our system would involve adding support for mobile devices as some functionality is currently lacking for these devices. For this to work, the spatial view would need to accept touch input through either buttons or a joystick interface for navigating the map. Additionally, we would also update the CSS to ensure an appropriate layout is presented on mobile devices.

Finally, it would be beneficial to add a variety of additional features usually offered by video conferencing applications such as screen sharing, selective muting, and text-based conversations.

Bibliography

[1] Cisco. The leader in collaboration customer experience | webex. https://www.webex.com/, April 2023.

[2] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, pages 266–296, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[3] Kjeld Egevang and Paul Francis. The ip network address translator (nat). Technical report, 1994.

[4] Weston Everett. A dynamic video conferencing app in links. 2022.

[5] Simon Fowler. Distribution · links-lang/links wiki. https://github.com/links-lang/links/wiki/Distribution, June 2017.

[6] Simon Fowler. Javascript ffi · links-lang/links wiki. https://github.com/links-lang/links/wiki/JavaScript-FFI, September 2017.

[7] Simon Fowler. Model-view-update-communicate: Session types meet the elm architecture. *CoRR*, abs/1910.11108, 2019.

[8] Henrike Gappa and Gabriele Nordbrock. Applying web accessibility to internet portals. *Universal Access in the Information Society*, 3(1):80–87, Mar 2004.

[9] Carl Hewitt. Actor model of computation: scalable robust information systems. *arXiv preprint arXiv:1008.1459*, 2010.

[10] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[11] Erzhen Hu, Md Aashikur Rahman Azim, and Seongkook Heo. Fluidmeet: Enabling frictionless transitions between in-group, between-group, and private conversations during virtual breakout meetings. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, New York, NY, USA, 2022. Association for Computing Machinery.

[12] Gather Presence Inc. Gather.town. https://www.gather.town/, October 2022.

[13] Gather Presence Inc. Integrated games. https://support.gather.town/help/integrated-games, October 2022.

[14] Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. In *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications*, pages 36–43, 2009.

[15] Amal Khalil and Juergen Dingel. Chapter four - optimizing the symbolic execution of evolving rhapsody statecharts. volume 108 of *Advances in Computers*, pages 145–281. Elsevier, 2018.

[16] Rob Manson. *Getting Started with WebRTC*, pages 24–34. Packt Publishing Ltd, 2013.

[17] Colin McClure and Paul Williams. Gather.town: An opportunity for self-paced learning in a synchronous, distance-learning environment. *Compass: Journal of Learning and Teaching*, 14(2), 2021.

[18] Microsoft. Video conferencing, meetings, calling | microsoft teams. https://www.microsoft.com/en-us/microsoft-teams/group-chat-software, March 2015.

[19] Mozilla. Webrtc connectivity - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity, April 2023.

[20] M Petit-Huguenin, S Nandakumar, G Salgueiro, and P Jones. Traversal using relays around nat (turn) uniform resource identifiers. Technical report, 2013.

[21] Lewis Raeburn. A spatial metaphor for dynamic video calling in links. 2022.

[22] Jonathan Rosenberg. Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols. Technical report, 2010.

[23] Jonathan Rosenberg, Rohan Mahy, Philip Matthews, and Dan Wing. Session traversal utilities for nat (stun). Technical report, 2008.

[24] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings First International Conference on Peer-to-Peer Computing*, pages 101–102, 2001.

[25] Morgan Smith. Bringing your team to gather. https://www.gather.town/blog/guide-onboarding-your-team, February 2023.

[26] Richard Soley et al. Model driven architecture. *OMG white paper*, 308(308):5, 2000.

[27] Xin Zhao and Colin Derek McClure. Gather.town: A gamification tool to promote engagement and establish online learning communities for language learners. *RELC Journal*, 0(0):00336882221097216, 0.

[28] Inc. Zoom Video Communications. One platform to connect | zoom. https://zoom.us/, April 2023.