A Gather.town Clone in Links

Caitlin McDougall

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

2023

Abstract

This skeleton demonstrates how to use the `infthesis` style for undergraduate dissertations in the School of Informatics. It also emphasises the page limit, and that you must not deviate from the required style. The file `skeleton.tex` generates this document and should be used as a starting point for your thesis. Replace this abstract text with a concise summary of your report.

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

Any acknowledgements go here.

Table of Contents

Chapter 1

Introduction

Chapter 2

Background

2.1 Web Technologies

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.

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 to provide additional styling, and a language such a JavaScript to provide dynamic functionality. 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 [8].

2.1.2 Links

For this reason, the Links programming language has been created, offering a new 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 [1]. 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 the Actor Model. In this model, an Actor is responsible for managing its own state and performing computations. 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 [4]. The type and order of these messages are ensured by defining session types. However, the linear nature of session typing 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 [2]. 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 these models and produces new models. The extension to the MVU architecture allows for session typing by adding commands (which enable side effects as seen in traditional web technologies), linearity, and model transitions.

Since the Links language is still in active development, it still lacks some functionality which is provided by JavaScript. Fortunately, Links offers a Foreign Functions 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. In particular, this paper will make use of FFI to make use of the WebRTC framework to achieve real-time communication between clients.

2.2 WebRTC

WebRTC 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 [10]. This framework allows peers to send and receive information directly, without first sending the data through a server which can create delay.

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 [12]. 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 [12]. 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 [10].

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.

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). 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. 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.

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.

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). 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, 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, Microsoft Teams, and Cisco Webex 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 [6], 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. 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.

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 through 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 [7].

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 [13]. 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 [11] determined that both educators and students preferred using Gather.Town in comparison with using other static VCS such as Zoom and Teams. However, accessibility of Gather.Town may be an issue for those with visual impairments or sensitivity to visual stimuli. Therefore, 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 [5]. 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 [5]. 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.

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 [3]. 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 paper 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

A Model-Driven Architecture (MDA) is one approach which allows us to abstract this problem of having multiple views from the underlying technical details. MDA uses an incremental approach to the system design process in which models are kept at the forefront of thinking [9]. These models represent different levels of abstraction of the underlying system. For example, in our VCS, the two visual representations offered to the user are two different models which each represent a high-level abstraction of the underlying system model. 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.

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.

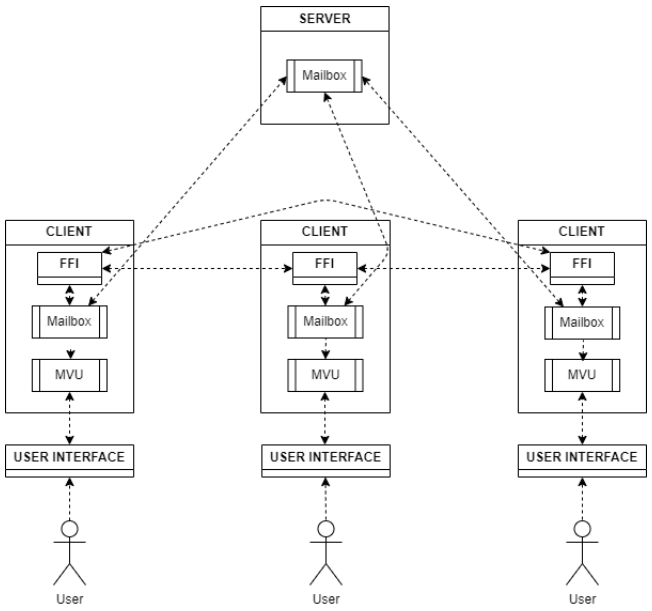


Figure 3.1: System diagram of VCS components and interactions.

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 functions interface 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' in-built 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.

Full descriptions of each component of the system are given below.

3.2 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 maximum communication latency, a server is still necessary to act as a middle-man between these clients when they initially set up these direct Peer-to-Peer connections.

3.2.1 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 a square within the correct room and using the number of clients on each square, they can be placed in a square which is unoccupied if such a square exists.

Upon initialisation of the server, it will immediately begin creating this grid such that it

is ready when clients begin accessing the application. 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.3 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 Functions 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.

3.4 Client

3.4.1 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 invoke a variety of functionalities, many of which invoke the WebRTC JavaScript Foreign Functions as described above. Functionalities offered by the mailbox 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.

3.4.2 Model-View-Update

So far, the client follows Links' standard Actor-based model. However, as described in subsection 2.1.2, this model's linear session typing is not ideal for the creation of GUIs and so our client also makes use of the Model-View-Update architecture. 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 model held by the client tracks the state of both this client 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.

3.5 User Interface

As well as providing a functioning Video-Conferencing System, this project also aims to offer a choice between interface designs to improve accessibility. The client's MVU architecture described above gives the perfect solution for providing this capability. The application can therefore provide 3 view functions depending on which page the user is on: The landing page, the spatial map-based view of the VCS, and the static view of the VCS.

3.5.1 Landing Page

The landing page of the application 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.

3.5.2 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 3.2 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.

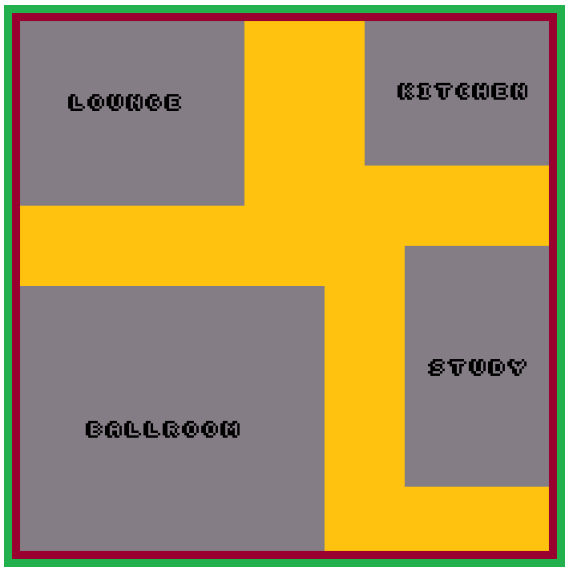
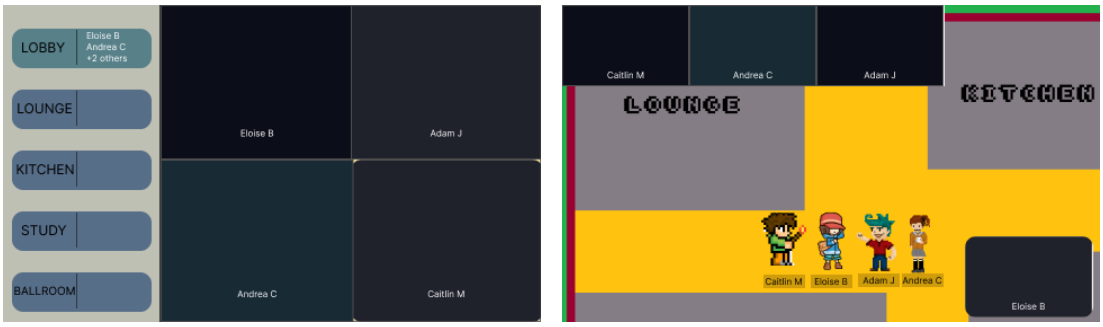


Figure 3.2: System diagram of VCS components and interactions.

3.5.3 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.

A side-by-side comparison of initial spatial and static view designs of the same state is displayed in Figure 3.3. Four users are using the application and, in the spatial view, this is demonstrated by displaying the four user's icons and names in the orange lobby area. The video streams of other users are displayed along the top of the screen and the user's own stream is displayed in the bottom right. In the static view, the users' names are listed in the Lobby room to the left with the video streams of each user taking up the rest of the screen to 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) Initial Design of Static View (b) Initial Design of Spatial View

Figure 3.3: Side-by-side comparison of (a) the static view and (b) the spatial view in the same state

.

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 map squares, an empty list of connected clients, and a complete list of rooms available which 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 squares consisting of its placement in terms of distance from the top and left of the webpage, which room it is contained within, and the number of occupants in the 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.

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, a view and update function to be used, the id of the HTML element for the result of the view function to be placed, and a subscription function which will listen for any specified events which in our case listens for onKeyDown events.

In the initial model state, the clients' id is originally set to 'None' as the server has not yet allocated it an id. Similarly, the x and y indices and pixel positions are set to 0 as these too are yet to be allocated by the server. The client's name is temporarily set to a default name of 'Nameless' and the icon set to 'None' as the user will enter these on the first page. The list of other clients' positions is initially empty, and the list of rooms contains each room's name along with an 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 want 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 the client a string ID equal to the length of connected IDs upon entry. This string 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 along with an initial room defined as 'None' since the client has not yet entered the video-conferencing space. 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 Functions interface of the client is also initialised with an empty dictionary ready to store data about any open connections with peers. The property which represents whether there is a local video stream associated with the client is initially set to 'False' such that the client can wait until a local video stream is available before registering to the VCS which 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 will repeatedly query the FFI until it discovers that the media stream has been fetched successfully. Meanwhile, the FFI will make requests to any user devices for any media which captures

audio or video. 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. This step is also independent of which view the user has chosen as the server's actions and the client's underlying MVU model will be the same. 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.

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 a new view to the user which depends on which view option was selected.

4.2.2 Server Registration Process

Upon receipt of this message, 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 will then broadcast the new client's PID to those clients also residing in the 'Lobby' by initiating a call between them. 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. These details are also sent to the new client's own mailbox such that it can update the model of its own position. To other clients, a request is also sent for their current position. The other clients will respond to this request from the server which will then forward their position to the newly joined client such that it has an up-to-date view of the current VCS state. 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. Once this is done, the server registration of the client to the VCS is complete.

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 elements are shown in **IMAGE PLEASE**

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, the view function first loops through the members of the model's 'othersPositions' list and for each character, it calls the 'getCharacterHTML' function and adds the result to its list of other characters' HTML. In both cases, the 'getCharacterHTML' function can then use the xpos, ypos, icon, and name properties of the argument to create nodes with their left, right, background-image, and text properties set respectively. Next, the 'Switch to static view' button is generated which allows a switch of views by passing the onClick property a function which sends the 'StateChange' message to the client's update function as described above. Finally, all of these created elements are added as children to a node which has the map as its background image such that they will appear on top of this map.

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 can be seen in **IMAGE PLEASE**.

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.

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’ such that keypresses in other views do 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 **??**. 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 which room they would like to join from a list of available rooms. Each of these room buttons has its ‘onClick’ property set to a function which 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. 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 will first use the user’s current x/y indices and key code to get the new grid position after moving in the given direction. Since the grid array is created from top to bottom, when the key code is “ArrowDown”, the x-index of the client is incremented to move down to the next row and the square at the resulting indices is found. The information in this 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. If the new room entered does not match the client’s previous room, it will send an ‘UpdateRooms’ message to all clients which will tell them to update the member lists in their model of the VCS rooms. 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 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 indeed 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. If an unoccupied square is not found, 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 information along with its new grid indices, pixel positions, and room name 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 check the IDs of currently stored client positions and if the client is already contained in this list, it will be replaced with the new position. If the ID is not already present, then the position is appended to the list. A new model is then produced with the ‘OthersPositions’ state updated to hold the new list of positions.

Again, when the client mailbox receives an ‘UpdateRooms’ message from the server, it dispatches an ‘UpdateRoomPids’ message to the 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 new room of the client, then the client 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

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. 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 will handle this request.

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 **??** 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 server 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 **??**. 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 associated with 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 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 'SendIceCandidates' 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

Conclusions

Bibliography

[1] 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.

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

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

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

[5] 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.

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

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

Appendix A

First appendix

A.1 First section

Any appendices, including any required ethics information, should be included after the references.

Markers do not have to consider appendices. Make sure that your contributions are made clear in the main body of the dissertation (within the page limit).

Appendix B

Participants' information sheet

If you had human participants, include key information that they were given in an appendix, and point to it from the ethics declaration.

Appendix C

Participants' consent form

If you had human participants, include information about how consent was gathered in an appendix, and point to it from the ethics declaration. This information is often a copy of a consent form.