Scaling a Dynamic Video Calling Application

Lewis Raeburn

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

2023

Abstract

The video calling application implemented prior to this project was a dynamic one in the sense that users could only begin video calls with one another when they moved their characters near each other. Throughout the rest of this report, I refer to this dynamic video calling application as the “dynamic app” for convenience.

The dynamic app had several software bugs and was not able to scale well in certain aspects. These aspects included the maximum number of users who could be in a video call at once as well as the maximum number of users who could move their character at the same time before the dynamic app became slow and unstable.

In this report, I detail the fixes I made to the list of bugs present in the current state of the dynamic app as well as various tests that measure the capability of it under different configurations and parameters. Then, I describe how I used the results of these tests to make optimisations to the dynamic app such that it scales to more users.

Near the end of the report, I evaluate the project in terms of the test methodologies used and the results obtained, as well as how well the optimisations improved the performance of the dynamic app. Finally, I discuss future extensions and conclude.

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.

(*Lewis Raeburn*)

Acknowledgements

I would like to thank my brilliant supervisor, Sam Lindley, for his constant useful support and feedback throughout both parts of my project. I enjoyed the work as well as the meetings involved, and I am glad I chose this project.

I would also like to thank my family and my partner for supporting me throughout my degree, as without them my achievements may not have been possible.

Table of Contents

1 Introduction **1**
- 1.1 Project Motivation . . . 1
- 1.2 Project Goals . . . 1
- 1.3 Summary of Contributions . . . 2
- 1.4 Report Structure . . . 3

2 Background **4**
- 2.1 Web Development in Links . . . 4
- 2.2 Web APIs Used for Video Calling . . . 5
  - 2.2.1 The Connection Process . . . 5
- 2.3 Accessing Web APIs in Links . . . 7
- 2.4 Distributed Actor-Style Concurrency Model . . . 8
- 2.5 Model-View-Update Framework in Links . . . 8
- 2.6 Outline of Current Design and Implementation . . . 9
  - 2.6.1 Links WebRTC API . . . 9
  - 2.6.2 Dynamic Video Calling Application (Dynamic App) . . . 11
- 2.7 Deficiencies Identified in the Dynamic App . . . 14
  - 2.7.1 Software Bugs . . . 14
  - 2.7.2 Scalability . . . 15
- 2.8 Terminology . . . 16

3 Empirical Limit Testing **17**
- 3.1 Benchmarking Notes . . . 17
- 3.2 Computers Used . . . 17
- 3.3 Test Methodologies . . . 18
  - 3.3.1 Video Calling Tests . . . 18
  - 3.3.2 Movement System Test and Measurements . . . 23
- 3.4 Test Results . . . 26
  - 3.4.1 Video Calling Test Results . . . 26
  - 3.4.2 Movement System Test Results . . . 30

4 Dynamic App Changes **31**
- 4.1 Software Bug Fixes . . . 31
- 4.2 Optimisations . . . 34
  - 4.2.1 Video Calling Changes . . . 34
  - 4.2.2 Movement System Changes . . . 36

5 Discussion **37**
5.1 Evaluation of Test Methodologies 37
5.2 Evaluation of Test Results 37
5.3 Evaluation of Video Calling Changes 38
5.4 Evaluation of Movement System Changes 39
5.5 Conclusion 39
5.6 Possible Extensions 40

Bibliography **41**

A Raw Movement System Test Results **43**

Chapter 1

Introduction

1.1 Project Motivation

The dynamic app is a program I implemented using the functional programming language Links [21], prior to this project. Similar to the original spatial metaphor for video conferencing Gather Town [8], the dynamic app allows users to move a character on their screen using the arrow keys on their keyboard and begin video calls with each other when they move their characters close enough to each other. It is mainly implemented using Links, but required a large chunk of JavaScript to be able to access the relevant live streaming APIs.

The main motivation for implementing the dynamic app originally was that Links did not support live streaming of user media, so my main goal was to implement an API in Links to support this. Then, the dynamic app was a thorough example of how to use the API. Another motivation was that the source code of the popular Gather Town web application was not open source, so my project and its source code - publicly available on my GitHub repository and on the main Links webpage - provided this.

Regarding the motivation for this project, the dynamic app had many software bugs (**Section 2.7.1**) and was capable of having a maximum of 15 users in a single video call at once (over 4 remote computers) before it became unstable and unusable. On a single machine, a maximum of 8 users could be in a single video call at once. Also, it could only support up to 8 users moving at once before each user's browser became slow and extremely delayed. These shortcomings rendered the dynamic app virtually unusable when the number of users reached nearly double digits.

1.2 Project Goals

The main objective of this project is to improve the dynamic app developed previously such that observed software bugs are fixed and that it scales to more users. More specifically, the improved dynamic app should have none of the software bugs identified in **Section 2.7.1**, should be able to support more than 15 users in a single video call at once (over 4 remote computers) through software optimizations only, and show no

instabilities as well as no slowed performance when more than 8 users are moving their characters at once.

The workflow of this project includes multiple separate parts. Firstly, the software bugs identified in the original dynamic app should be fixed, as well as bugs that appear thereafter. Secondly, the (fixed) original dynamic app should be tested in various ways and the results of these tests recorded (e.g., measure how many users can use the dynamic app at once when the resolution of streamed video media is reduced by a factor of 10 before the dynamic app becomes slow and unusable). Thirdly, once the results of these tests have been captured, the results should be used to optimise the dynamic app such that more users can be in a video call at once and also such that more users can move their characters at once than in the original dynamic app.

1.3 Summary of Contributions

I successfully fixed multiple software bugs (listed in **Section 2.7.1**) that were identified in the original dynamic app and that arose upon fixing those. Upon fixing these software bugs, I was then able to complete various tests that measured the performance of the dynamic app under different parameters. The following is an exhaustive list of the tests I carried out on the (fixed) original dynamic app, both on a local server on a single computer and on a cloud server on 4 separate computers. I recorded the maximum number of users that can be in the (fixed) original dynamic app at once:

- without movement or media
- with movement but without media (using an automated version of the dynamic app to move all users at once)
- in a single video call with audio only
- in clustered video calls (i.e., with a limit for how many users can be in a single video call, 2 users per call, 3 users, etc.)
- in a single video call with lower FPS
- in a single video call with lower resolution

Using the results recorded from these tests, I identified the parameters where the dynamic app was capable of serving more users than the default parameters (a single video call with baseline FPS and resolution). Two of these parameters were the lowering of FPS and resolution, both of which separately increased the maximum number of users in a single video call (lowering FPS from 60 to 10 or resolution from 200 x 150 pixels to 40 x 30 pixels yielded an increase of 4 users). In the improved version of the dynamic app, I altered the design of the video calling process by forcing users, who have not spoken for 5 seconds, to reduce their streamed video resolution to 40 x 30 pixels (from 200 x 150 pixels). Then, immediately after speaking, their resolution is set back to default (200 x 150 pixels).

Also, I noticed that the test involving every user moving at once (using the automated version, which simulates arrow key pressing) without any media / video calling resulted

in a very low maximum number of users (8 users). In the same test, I measured the number of position-update messages being received by a single user, while every user was moving (and therefore constantly sending position-update messages to every other user) and found that, after 2 minutes with 9 users, the user was receiving 128 messages per second on average and the average time between the user receiving a position-update message and updating the position of the user that sent the message was 22.62 seconds.

Observing the inefficiency of this design of movement system, I changed the design such that users move by setting a target position and moving towards it, so instead of broadcasting countless position-update messages, each user broadcasts their target position whenever it is updated. When I tested the automated version of the dynamic app with this new design, the user received only 2 messages per second on average and the average time between the user receiving a new-target message and updating the position of the user that sent the message was 0.05 seconds. In the default non-automated version of the dynamic app, this maps to having users clicking where they want to move on their screen instead of using the arrow keys on their keyboard.

1.4 Report Structure

In **Chapter 2**, I introduce the background and important concepts used throughout the project as well as information about the project I completed previously that implemented the dynamic app and the Links WebRTC API used by it. In **Chapter 3**, I describe the test methodologies used to test the dynamic app under various configurations and the test results obtained. In **Chapter 4**, I describe the implementation changes made to the dynamic app that includes the bug fixes and optimisations. In **Chapter 5**, I evaluate both the testing completed and the optimisations made, and I also conclude and discuss potential future extensions in terms of scaling the dynamic app further.

Chapter 2

Background

In this chapter, I explain the key concepts and relevant software technology involved in the dynamic app implemented previously, as well as an outline of the design, implementation, and evaluation of it. Software technology includes the functional programming language Links [7] and the WebRTC peer-to-peer real time media streaming API [22]. The key topics include distributed actor-style concurrency and the model-view-update pattern.

2.1 Web Development in Links

Building a functional, appealing, and interactive web application requires knowledge of the three tiers of web development and learning the following languages: HTML, CSS, and JavaScript to set and update the state of the DOM (Document Object Model) and control what the user's browser displays; Java, PHP, or Python to implement the server (known as back-end); SQL to send queries to the database if applicable. For a novice web developer who simply wants to build their own website, learning such a myriad of languages (i.e., the full-stack) is not optimal. It is also difficult to link these tiers to ensure that messages sent from one tier to another match the data type the receiving tier expects (e.g., to make sure that a form in HTML produces data of a type that the server expects). This difficulty of linking these tiers is known as the "impedance mismatch" problem [7]

With Links, this language alone is enough for the programmer to develop an aesthetically pleasing and well-functioning web application. Therefore, the novice web developer may benefit more from learning just one language - Links - instead. Also, Links solves the impedance mismatch problem by providing a single language to handle all three tiers, producing JavaScript for the browser, a bytecode for the server, and SQL for the database.

Throughout the initial implementation of the live streaming API and the dynamic app, as well as the improvements made in this project, I made use of multiple features of Links. Features include the distributed actor-style concurrency model and the Model-View-Update framework. However, before describing these, I explain how users connect to

each other using the WebRTC live streaming API.

2.2 Web APIs Used for Video Calling

The Links API I implemented to support live video streaming (previously) makes use of 2 web APIs. Firstly, the MediaDevices web API [15] is required to access the user's microphone and webcam. Secondly, the web API used to allow users to stream and communicate data coming from their media devices is the WebRTC web API [22]. WebRTC is vital in enabling rapid transfer of live media streams between users' browsers without necessarily needing an intermediary server, so the connections are peer-to-peer.

To summarise how WebRTC is used to connect users and allow them to share their live media streams, firstly, there must exist a simple server to act as an intermediary to allow the users to exchange connection, media, and security details. Next, using this data, the users' browsers must find the best way to access each other. Lastly, they initiate a WebRTC peer connection between each other and transfer media streams.

In the subsection below (**Section 2.2.1**), I elaborate on the details of how a WebRTC connection is executed. Throughout this elaboration, I make the assumption that every user wishes to begin a WebRTC connection with every user they discover, as soon as they discover them.

2.2.1 The Connection Process

The Signalling Server

Let there be two users (user 1 and user 2) who wish to connect and share real time data such as live video streams with one another. To begin the process, each user connects to a "signalling server". In my implementation, this is the server process that the clients use to communicate initially. This server simply broadcasts any messages sent to it to all users connected. When user 1 connects, they send a message to the server that reaches no one else since they are the only one connected. When user 2 connects, they send a message to the server that is propagated to and received by user 1. Since user 1 discovers user 2, they begin the WebRTC connection process with user 2 by creating an RTCPeerConnection object (through the WebRTC API), which is associated with user 2, then begin collecting ICE candidates (more on ICE candidates in the "Identifying ICE Candidates" section, below). Upon receiving this message, user 1 sends a reply to the server that is propagated to and received by user 2 so that they can begin the same process too.

Identifying ICE Candidates

Upon creating an RTCPeerConnection object, the user identifies the IP addresses through which they can transfer and receive data. These may include the user's private IP addresses within a local network, which can be accessed by other devices in the same local network, or a public IP address, which can be accessed via the internet.

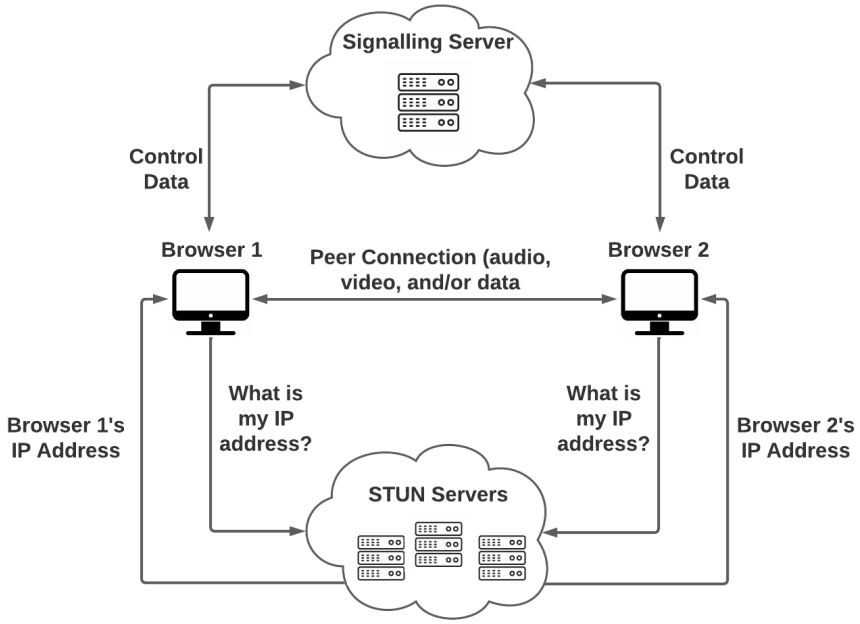

Figure 2.1: Communication with the signalling server and STUN servers.

These IP addresses are referred to as ICE (Interactive Connectivity Establishment) candidates. When a RTCPeerConnection object is created, it immediately begins to search for ICE candidates. To assist with this, the programmer can provide the URLs of various STUN (Session Traversal Utilities for NAT) servers as an argument to the RTCPeerConnection object. These STUN servers are queried by the user to identify any external (public) IP addresses of the user that can then be added to the user's collection of ICE candidates. **Figure 2.1** depicts the process of exchanging data between users and how they use STUN servers to find their external IP addresses.

Session Description Protocol (SDP)

Once each of the users has their own set of ICE candidates, they are ready to set up their connection. Firstly, user 1 creates an SDP (Session Description Protocol), which is a JSON object that contains key information including details about their media, security, their collection of ICE candidates, and multiple other things. User 1 sets this SDP object as their "local description", then converts the SDP object to a string data type so it can be sent to user 2 (via the signalling server). Once the SDP reaches user 2, user 2 sets it as their "remote description", then creates their own SDP, sets this as their "local description", and sends this back to user 1, who sets this as their "remote description".

Completing the Connection

Once the users have exchanged descriptions (SDPs) successfully, they know about each other's details and the multiple ways they can connect to each other through the various ICE candidates. The RTCPeerConnection object will determine the ICE candidates of each user that yield the best connection and use these as the IP addresses to transfer data (e.g., video, audio, arbitrary data) to and from one another. At this point, any media streams added to a user's RTCPeerConnection object will be streamed between the

chosen IP addresses (ICE candidates) and will be accessible by the corresponding user who can then use this media. The implementation of the WebRTC connection process is described in **Section 2.6.1**. **Figure 2.1** shows the entities involved and how they interact. The control data mentioned in the figure can be mapped to the sharing of SDP objects and ICE candidates between the users through the server.

2.3 Accessing Web APIs in Links

The MediaDevices and WebRTC APIs are required to access the user's camera and microphone and implement the WebRTC connection process. These APIs are accessed by JavaScript and so there must be a way to run JavaScript code through Links. This is done with Links' foreign function interface (FFI).

As an example, let there be two JavaScript functions that the programmer wants to run in Links: `getCameraReady` and `checkIfCameraLoaded`. The former takes a string as input, prepares the user's camera, and returns nothing. The latter takes nothing as input and returns a boolean indicating whether the user's camera is ready. In the JavaScript file, the declarations of these functions are prepended with an underscore. Again in the JavaScript file, these underscored functions are wrapped in a call to `LINKS.kify` to fit in with Links' CPS (Continuation-Passing Style) calling convention, so they can be accessed by the Links code. The code below shows how the JavaScript file is implemented in order to use Links' FFI.

```
function _getCameraReady(camID) {
    ...
}

function _checkIfCameraLoaded() {
    return ...
}

let getCameraReady = LINKS.kify(_getCameraReady);
let checkIfCameraLoaded = LINKS.kify(_checkIfCameraLoaded);
```

Before the Links program can run these JavaScript functions, it must know where the JavaScript file containing these functions is located. This is done by declaring a module in the Links file with the inclusion of the `alien` keyword followed by `javascript` and the relative file destination. The JavaScript functions to be used in the Links program are included inside this module, along with the input and return types of these functions. This is shown in the code listing below.

```
module JSFuncs {
    alien javascript "js/jsFuncs.js" {
        getCameraReady : (String) ~> ();
        checkIfCameraLoaded : () ~> Bool;
    }
}
```

Finally, to call one of these functions in the Links program, the module name is written followed by the desired function using dot notation, as displayed below.

```
JSFuncs.getCameraReady(cameraId)
var isCameraReady = JSFuncs.checkIfCameraLoaded();
```

2.4 Distributed Actor-Style Concurrency Model

As discussed above in the section describing the connection process (**Section 2.2.1**), users who wish to connect to each other send messages to a "signalling server" that broadcasts the messages to all connected users. Information such as the sender's unique ID and the intended recipient's unique ID may be included in the message so that the recipient can determine who the message is from.

There are two main components involved in this communication: the clients (users) and the server (signalling server). Depending on the software or programming language being used, the method used to implement this messaging system between clients and server varies. An expert in HTML, CSS, and JavaScript may use those to program the client, and a JavaScript runtime environment such as NodeJS [16] to implement the server. This set of languages and environment alone would be sufficient to implement a video calling application using WebRTC [13].

In Links, due to the fact that a single Links program runs both the client and the server, the messaging system between the clients and the server differs from above. A communication model that Links supports is known as the distributed actor-style concurrency model (based on the actor model [5]).

With this model, the program comprises of several independent units of computation (processes) that are referred to as "actors". These processes are concurrent, so any messages sent from them are asynchronous and the program need not wait for them. The actors wait to receive a message, then use pattern matching to perform actions based on the message. It is distributed in the sense that not all actors need to be run on the same computer (i.e., an actor can be run on a different computer from another and still receive messages from it, provided that the sender knows how to reach it).

The implementation of this model in Links involves the use of multiple processes to act as clients and one process to act as the server. The server process waits for messages to arrive from client processes, then broadcasts these messages to all client processes.

Using this distributed actor-style concurrency model in Links, it was possible to implement the complete WebRTC connection process.

2.5 Model-View-Update Framework in Links

Links provides various ways of interacting with the DOM to control what the browser displays for the user. The simplest of these is an imperative one that involves the use of DOM operations to add nodes to the DOM, or change nodes already in the DOM. For example, `insertBefore(xml, beforeNode)` inserts an XML node (e.g., <p>

or <video> to insert text or a video) into the DOM before `beforeNode`. This set of operations is similar to that of JavaScript's, where the developer makes use of an API of functions to manipulate the DOM.

The other approach, and the one used to implement the dynamic app, is to make use of Links' Model-View-Update (MVU) framework [19]. This method of updating the DOM is similar in some ways to that of the JavaScript library, React [18], and is purposely based on the front-end web programming language, Elm [6]. A program using the MVU pattern has three main components: a model datatype that contains the state of the program, a view function that takes the model as input and renders it as HTML and displays it on the browser, and an update function that takes as input the model and a message and returns an updated instance of the model.

The message taken as input in the update function relates to the MVU framework's concept of "subscriptions", which can be thought of as event handlers that run the update function with a particular message when triggered. Within the update function, pattern matching is used to check which message has been received, and runs a particular chunk of code depending on the message. Once the update function makes its changes to the model, the view function takes the new model and generates new HTML to display on the browser.

2.6 Outline of Current Design and Implementation

In this section, I summarise the details of the design and implementation of the Links WebRTC API and the dynamic app built on-top of it, which I completed prior to this project [14].

2.6.1 Links WebRTC API

The main functions implemented in the Links WebRTC API are listed below, along with their type signatures. Each of these functions makes calls to JavaScript functions in the JavaScript file `jsWebRTC.js` through the FFI.

- `gatherDeviceIds : (DeviceType) -> ()`
- `getDeviceIds : (DeviceType) -> [DeviceID]`
- `getDeviceLabels : (DeviceType) -> [DeviceLabel]`
- `readyMediaDevices : (DeviceID, DeviceID) -> ()`
- `registerUser : () -> ()`
- `connectToUser : (Uuid) -> ()`
- `disconnectFromUser : (Uuid) -> ()`
- `checkifConnectedToPeer : (Uuid) -> Bool`

All data types specified in the type signatures above are of primitive type `String`, except `Bool`. Despite multiple bugs being present, these functions were enough to implement

the video streaming component of the dynamic app.

As their names suggest, the top 3 functions allow the caller to identify the media devices available on their computer, and get the IDs and labels of each device. `gatherDeviceIds` uses Links' foreign function interface (FFI, **Section 2.3**) to access the MediaDevices web API. It uses the `enumerateDevices` method of this API to scan the computer for available devices, then stores the ID and label of each device in JavaScript arrays. Since `enumerateDevices` returns a JavaScript promise, `gatherDeviceIds` is not able to return the IDs and labels immediately, which is why they are stored in JavaScript arrays so they can be accessed later. `getDeviceIds` and `getDeviceLabels` check that the storing of device IDs and labels has completed before returning them.

Once the user chooses a pair of video and audio devices (using the device labels, which are mapped to IDs), `readyMediaDevices` is called with these device IDs to access the streams of these devices. It does this by using the FFI to call the `getUserMedia` method of the MediaDevices API, which also returns a JavaScript promise. It then stores the video and audio streams in JavaScript variables, which are used later to share the streams via the WebRTC web API, and also for the user's local video stream to be able to see themselves.

The `registerUser` function simply sends a message to the server process letting it know of the user's presence, after which the server adds it to its list of registered users. Once a user sends a broadcast message to the server, the server propagates the message to all registered users.

Onto the WebRTC connection part of the API, a user (user 1) calls `connectToUser` with the ID of the user they wish to connect to as an argument. In short, this function begins the WebRTC connection process described in **Section 2.2.1**. It creates a RTCPeerConnection object associated with the other user in JavaScript using the FFI, adds user 1's stream to it, and sets up event listeners that trigger when ICE candidates are found (which are added to an array) or if a media stream is received by the other user. In the JavaScript file, there is a dictionary that maps other users' IDs to RTCPeerConnection objects associated with them (one per user). Every half a second, a client process in the Links program retrieves the array of ICE candidates from the JavaScript code and broadcasts them via the server process. At the end of `connectToUser`, a message of type `ConnectionRequest` is sent to the user they wish to connect to through the server process.

Once the other user (user 2) receives this `ConnectionRequest` message, it also creates a RTCPeerConnection object associated with the sender of the message. In response to this message, user 2 creates a SDP (Session Description Protocol) object using the WebRTC API's `createOffer` method in JavaScript. Each RTCPeerConnection object must have 2 SDP objects associated with it: a local SDP object representing the user who created the object, and a remote SDP object representing the user to be connected to (there are 4 SDP objects in total as both users have their own RTCPeerConnection object representing the connection). Once user 2 creates their SDP object, they call the WebRTC API's `setLocalDescription` method to set the local description of the RTCPeerConnection. Then, user 2 sends a message of type `SDP` with the SDP object (stringified) attached and, upon receiving this message, user 1 calls the WebRTC API's

`setRemoteDescription` using the attached SDP object to set the remote SDP object of the RTCPeerConnection object. User 1 repeats this process and once user 2 sets their remote SDP object the connection is established, and any media streams that were added to the RTCPeerConnection object are shared and available.

The `disconnectToUser` function takes the ID of the user that the caller wishes to end the connection with and calls a function in JavaScript to delete the entry in the RTCPeerConnection entry associated with the user.

Lastly, `checkIfConnectedToPeer` takes the ID of the user of which the caller wishes to query the existence of the connection with. It returns true if the connection exists, and false otherwise.

2.6.2 Dynamic Video Calling Application (Dynamic App)

On a high level, the dynamic app presents the user with a box-shaped area and a circular icon that represents the position of the user on the screen (also referred to as the user's character). The user is able to move their character around the screen using the arrow keys on their keyboard. The user is also able to see the characters of other users who have also joined the server. When a pair of users get close enough to each other, they begin a WebRTC connection and their live streams of video and audio appear on each other's browsers. When the same users move apart enough, the WebRTC connection is closed and the shared media streams are removed. An image of the dynamic app in use with 2 users is shown in **Figure 2.2**.

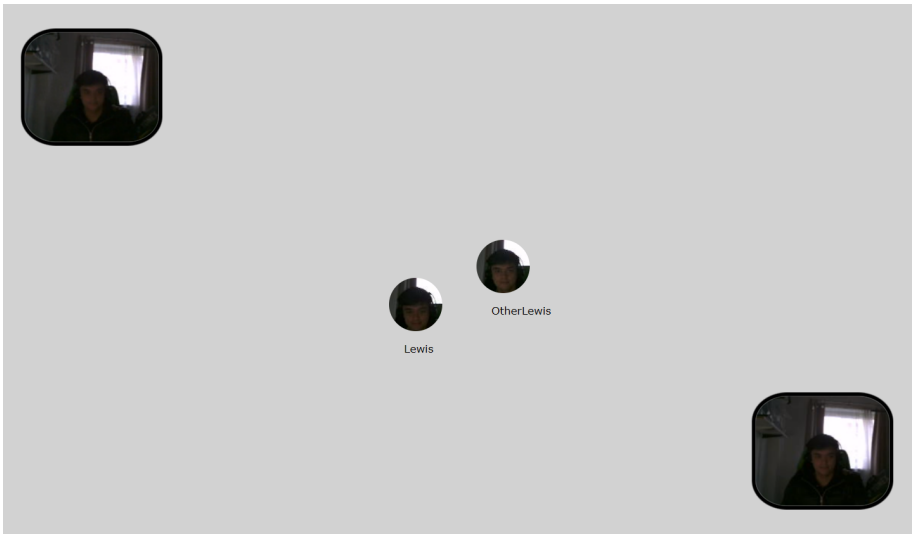

Figure 2.2: When another user connects, the local user can see their character and its position on the page. A video call begins when the users move within close proximity of each other.

I implemented the dynamic app's user interface using Links' MVU framework. As described in **Section 2.5**, the MVU framework is one of Links' systems of maintaining the state of a program and displaying it to the browser and consists of a model datatype, an update function, and a view function.

In the dynamic app, the model datatype is a Links record, which is a tuple with its fields indexed by field names rather than integer indices. This record contains 8 fields, as shown in the `Room` datatype displayed below.

```
typename Room = (charData : CharacterData,
                 directionV : CharacterState,
                 directionH : CharacterState,
                 state : String,
                 nameField : String,
                 cameraId : String,
                 micId : String,
                 others : [CharacterData]);
```

The first field represents the user's character data and has the type `CharacterData`, which holds information about the character's position on the screen mainly. The second and third fields have the type `CharacterState` and represent the direction that the character is moving in along the horizontal and vertical axes. The values that these direction datatypes hold depend on the arrow keys that are pressed by the user (e.g., pressing the left arrow key sets the horizontal direction record field to `Left`).

The dynamic app has an introduction phase split into 3 parts. Firstly, the user is asked for their name - to be displayed to other users on their character. Secondly, they are asked to choose the camera and microphone they wish to use. Thirdly, they are asked to take a picture of themselves for the character icon image (shown in **Figure 2.3**). Each of these parts of the introduction phase are presented to the user in their own separate user interface. The fourth field of the model record is a string that indicates which of the 4 user interfaces should be displayed to the user (the fourth being the main interface around which the user can move their character).

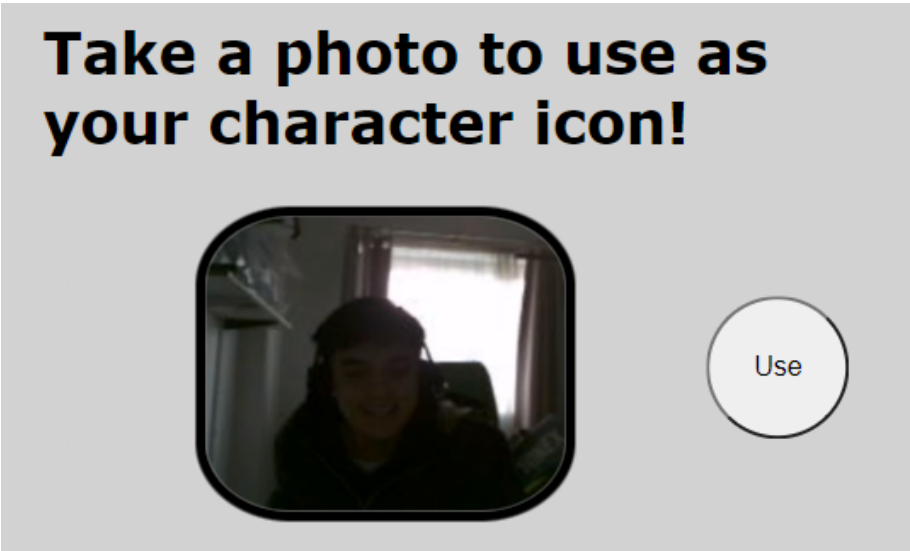

Figure 2.3: User Interface 3. The user can use their own live video stream to position their face for the picture that will be used as their character's icon.

Fields 5, 6 and 7 of the model record are used in user interfaces 1 and 2 to store the

user's name as it is being typed in and keep track of which camera and microphone the user has selected, so that when these details are confirmed, these fields can be accessed to retrieve the chosen values.

Lastly, the final field is a list of `CharacterData` instances to keep track of the state of each of the other users' characters.

The view function looks at the `state` field of the model record (fourth field) to determine which user interface to display. If the user has completed the 3 introduction phases of the dynamic app, the view function displays the user's character with the photo they took of themselves as the icon, and loops through the final field of the model record to display all the other users' characters.

The update function accepts 9 different types of message, which are received when certain events are triggered. Two of the accepted messages are `MoveCharV` and `MoveCharH`, which are sent to the update function when the user presses the up or down arrows or the left or right arrows, respectively. Upon receiving either of these two messages, the direction fields of the model record are updated as well as the position of the character. Also, the user broadcasts their position to all other users via the server process. Another type of message is `NewFrame`, which is triggered after every new frame update. When this is received, the position of the character is updated depending on direction fields of the model record.

Four other types of message that can be received are `UpdateNameField`, `NameEntered`, `MediaChosen`, and `Joined`. The first of these is received during the first part of the introduction phase whenever the user types in a character of their name. It updates the user's name field of the model record. `NameEntered` is received when the user confirms their name, and changes the model record's `state` field to the second part of the introduction phase - choosing media devices. Once the user has confirmed their chosen camera and microphone, the update function receives the `MediaChosen` message. This sets the camera and microphone ID fields in the model record (fields 6 and 7), and also the `state` field to the last part of the introduction phase - live photograph. When the photograph is taken, a JavaScript function is called to take a snapshot of the camera at that moment, which is converted to an image URL. This also triggers the `Joined` message to be sent to the update function, which sets the name field and image URL field of the user's `CharacterData` record to the current value of the name field of the model record and the image URL of the photograph taken, respectively. It also updates the `state` field of the model record to "joined", which tells the view function to display the main user interface of the dynamic app, where the user can move their character and interact with others.

The update function can also receive a message of type `ServerMsg`. When the user receives position-update messages from other users (following the actor model, described in **Section 2.4**), these messages are stored in the parameters of a function that waits for messages. There is a separate client process that repeatedly queries this waiting function to check whether any messages were received by other users. If so, a `ServerMsg` message is sent to the update function with the remote user's message attached. Since the message contains the ID of the user who sent it, the update function looks through its list of remote users (all other users running the dynamic app) until it finds the ID of

the message sender, then updates the position values in the `CharacterData` record of that user (or the image URL of that user, if it is the first message received by them).

Onto the video calling part of the dynamic app, a user starts a video call with another user when they move into the vicinity of the other user (within 150 pixels currently). They start the video call by calling the Links WebRTC API function `connectToUser` with the ID of the user they moved near as an argument. When the other user detects that the moving user has moved nearby, they check whether they have an existing WebRTC connection with them. If so, they get the live video stream and append it to the top of the user interface as a `HTMLVideoElement`. When either user detects that they or the other user has moved out of the vicinity of each other, they call the Links WebRTC API function `disconnectFromUser` to end the WebRTC connection and remove the video element at the top of the user interface.

2.7 Deficiencies Identified in the Dynamic App

2.7.1 Software Bugs

As this is the background chapter, I describe these bugs without specific reference to the implementation. I provide more detailed information about these bugs in the implementation chapter.

Bug 1: Clients attempt to connect while already connected

When one user (user 1) moves into range of another user (user 2), the dynamic app acts as expected and user 1 begins a WebRTC connection with user 2. However, if either user presses a movement key while still in range and after the connection has been made already, the browser reports an error stating that a WebRTC API method is being called in the wrong state. Referring to the WebRTC connection process, one of the users attempts to set their "remote description" (SDP), even though this has been set already and the connection process has completed.

Bug 2: Character movement functions incorrectly

When a user is moving along an axis (e.g., horizontally using the left and right arrow keys), if one key is pressed, and then the opposite key is pressed, the resulting direction is the one signalled by the latest key press, whereas the correct movement should be no movement. Also, if both keys are pressed and one is released, then the character stops moving, whereas it should move in the direction signalled by the key that is still pressed.

Bug 3: Message handler leaks messages

When two or more users use the dynamic app, it is likely that the user that joined the server last is missing at least one peer user on their screen. This is because the function that waits for and stores messages from remote users (to be retrieved by the MVU loop) in the dynamic app had its own bug whereby if more than 1 message (from remote users) was received by the function without the MVU loop retrieving a message from

it, then only the last message that was received would be stored, and the prior ones removed unintentionally.

Bug 4: Dynamic app freezes when two users move into each other

When two users move into each other's vicinity, the dynamic app freezes for each user. This is because the dynamic app was designed such that the user who moves into the vicinity of the other user is the one who starts the WebRTC connection, and both users are moving in this instance, so a synchronization issue arises and neither user completes the connection. The dynamic app freezes because the function that initiates the call to `connectToUser` in the dynamic app waits for confirmation of the WebRTC connection, so the flow of execution of the dynamic app never leaves the waiting function.

Bug 5: Client disconnects immediately after connecting

Occasionally, when a user (user 1) moves into the vicinity of another user (user 2), the expected video call does not occur, which is immediately noticeable as no live video streams appear on either user's screen. This is because each user is constantly checking its distance from other users and invoking the `disconnectToUser` function from the Links WebRTC API with users who are outside their vicinity if a WebRTC connection exists between them. When user 1 moves into the vicinity of user 2, user 1 calls `connectToUser`, which causes both user 1 and 2 to create a RTCPeerConnection object and become connected. However, occasionally the WebRTC connection process completes before user 2 notices the position-update message from user 1 that puts user 1 in their vicinity, so user 2 sees user 1 outside their vicinity after the WebRTC connection has just completed, which causes user 2 to delete the RTCPeerConnection object and close the connection immediately after it is created.

2.7.2 Scalability

As mentioned in **Section 1.1**, in the current state of the dynamic app, no more than 8 users could be in a single video call at once when all users are connecting from the same computer. As for multiple computers (3 high-end computers and 1 low-end), no more than 15 users could be in a single video call at once before the dynamic app became slow and unusable. The distribution of users in this instance was 4 for each high-end computer and 3 for the low-end computer.

Despite these measurements, sources [3, 20, 13] suggest that WebRTC video calling applications that follow a mesh network topology (which this application does) struggle to reach numbers of users beyond single digits. On the other hand, another source [2] found that 122 media streams could be shared among 512 data channels with the Chromium sandbox disabled. However, this investigation was less realistic as it used the network loopback interface on a single computer, and made use of multiple data channels instead of 1. Also, according to the WebRTC source code [10], the maximum limit for the number of peer connections that can exist simultaneously is 500.

After comparing these different sources of data including the dynamic app, it is evident that many factors impact the scalability of applications using WebRTC. The sources suggesting a lower maximum number of users seem to have used high resolution (either

HD or 1080 x 720) for all users' video streams, which may be a large reason for the lower maximum. The source suggesting a higher maximum number of users completed their tests on a very high-end computer with 64GB of main memory. The dynamic app being tested and improved in this project had users stream video with resolutions of only 200 x 150 on computers mostly with large amounts of memory (**Section 3.2**). These parameters seem to sit between the parameters used in the above sources, which may suggest the higher maximum number of users (due to lower resolution) than the first 3 sources.

2.8 Terminology

Throughout the rest of this report, I make frequent use of various terms. In some instances, these terms may have an ambiguous meaning, so I have introduced a list of terms and their specific meanings in this report:

- **User**. This refers to a human being who uses the dynamic app.
- **Client**. When a user opens their browser and joins the dynamic app server (via the URL of the dynamic app), the client side of the dynamic app runs on the browser. This "client side" is what client refers to.
- **Character**. This refers to the user's movable icon when using the dynamic app.
- **Dynamic App**. This refers to the dynamic video calling application designed and implemented prior to this project, which is also tested and improved in this project.

I also make use of specific notation to distinguish between the different types of functions implemented. Functions denoted by "fun" are Links functions, functions denoted by "function" are JavaScript functions, and JavaScript functions whose names are prepended with an underscore are designed to be called via Links through the foreign function interface.

Chapter 3

Empirical Limit Testing

In this chapter, I detail the empirical tests I completed to identify the maximum number of users that can use the dynamic app before it becomes slow and unstable. Each of these tests are differentiated by varying the parameters with which the dynamic app is run.

3.1 Benchmarking Notes

Since the main goal of this project was to measure the performance of the original dynamic app, make changes to optimise it, then check whether these changes improved the performance, it makes sense to include notes on benchmarking practices.

In the empirical tests where I measure the maximum number of users that can use the dynamic app before it becomes slow and unstable, the numbers obtained for this test remained the same over 10 trials for each different test. Therefore, no method was required to combine the results of each trial into a single value.

On the other hand, measuring the performance of the movement system yielded different values over the 10 trials (apart from the maximum users). Since it is convenient to have a single number to benchmark performance against, I had to find a method of combining the results of all 10 trials. Fleming and Wallace [17] show that it is acceptable to take the arithmetic mean of the data as long as each individual measurement is of equal importance. Since each of the 10 trials are repetitions of the same measurement and the variance of them is not significant, they are all virtually of equal importance, so it is acceptable to take the arithmetic mean. To indicate the significance of the measurements, I also present the variance over the 10 trials in the results.

3.2 Computers Used

In the tests completed throughout the rest of the report, I made use of 4 different computers. 3 of these were of similar quality and more high-end, and the other more low-end. To avoid any misleading reports throughout the rest of this paper, I list the specifications of each computer below. Also, any test that makes use of a single

computer makes use of computer 1. Computers 1 and 2 are desktop computers and computers 3 and 4 are laptops, and computer 4 is the low-end one.

Computer 1

Operating System: Windows 10 Home 64-bit (10.0, Build 19044)
Processor: Intel(R) Core(TM) i5-8600K CPU @ 3.60GHz (6 CPUs)
GPU: NVIDIA GeForce GTX 1060 6GB
Memory: 16384MB RAM Age: 5 years

Computer 2

Operating System: Windows 10 Pro 64-bit (10.0, Build 19045)
Processor: Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz (8 CPUs)
GPU: NVIDIA GeForce GTX 1070 6GB
Memory: 16384MB RAM Age: 4 years

Computer 3

Operating System: Windows 11 Home 64-bit (10.0, Build 22621)
Processor: AMD Ryzen 5 5500U @ 2.10GHz with Radeon Graphics (12 CPUs)
GPU: AMD Radeon(TM) Graphics
Memory: 16384MB RAM Age: 1 year

Computer 4

Operating System: Windows 10 Home 64-bit (10.0, Build 18363)
Processor: Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz (4 CPUs)
GPU: Intel(R) HD Graphics 620
Memory: 8192MB RAM Age: 6 years

3.3 Test Methodologies

In general, I carried out two types of tests: ones that tested the video calling part of the dynamic app, and one that tested and measured the performance of the character movement system.

3.3.1 Video Calling Tests

As mentioned in **Section 1.3**, each of the following tests were completed on both a single computer (also running the server) as well as 4 computers connected to a cloud server powered by Google, as described in **Section 3.2**. The actual video output from each user is quite static (i.e., users are sitting still and likely moving their mouth only), which is arguably realistic in a typical use case of a video calling application.

Avoiding WebRTC and Media Altogether

In this test, the dynamic app was modified such that whenever users get close to each other, video calls are not initiated (i.e., `connectToUser` is never called). Also, the video element showing the user's own video stream is not shown. As initially intended, this test is used as the control test as a reference for the other video calling tests. **Figure**

3.1 shows the 100th user's browser after having entered the dynamic app and moved to the next position. Although not necessary, moving all users' characters to their own positions was useful to get an idea of what 100 users looked like, as opposed to leaving every character in this same position.

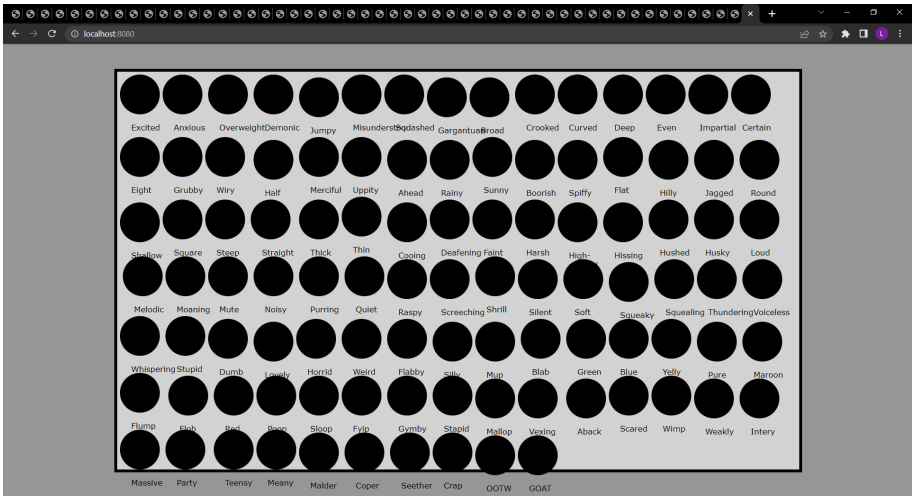

Figure 3.1: The end result of the control test with no WebRTC or media on the single computer.

Local Media Only

In this test, I adjusted the implementation such that video calls still did not occur, but the video element showing the user's own video stream is displayed. Although intended to be used as another control test to check whether the dynamic app performance bottleneck was the number of video streaming elements on the browsers of each computer, rather than the number of WebRTC connections, it was found that it is not possible to have more than 16 video elements of the user's own video stream in use at the same time.

Figure 3.2: The user interface of the dynamic app after connecting 16 users to the server on the single computer and moving them into their own positions.

The error shown in **Figure 3.3** appears when more than 16 media elements attempt to access the same audio source simultaneously. Despite this error, the test was carried out 10 times to ensure the consistency of this error. The look of the user interface after having 16 users join is shown in **Figure 3.2**.

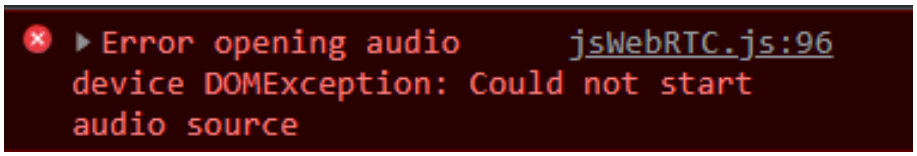

Figure 3.3: This error appears when the dynamic app attempts to access the same audio source for separate media elements more than 16 times.

Audio Only

In this test, I changed the implementation such that whenever users moved close to each other, they began a WebRTC connection but only shared their audio streams with each other. This involved adding functionality to the Links WebRTC API and the dynamic app to be able to add HTML audio elements to the DOM when audio streams from other users are detected. The code listing below shows a new Links WebRTC API function added that checks whether the client is running as audio only.

```
fun setAudioOnly() {
  JSWebRTC.setAudioOnly();
  ()
}
```

This Links function sets a variable `audioTrackOnly` in the corresponding JavaScript file, which is checked when the client creates a new RTCPeerConnection object and adds its own media streams to the connection. If this variable is set to `true`, only the client's local audio track is added to the connection. This variable is also used when a remote client's media tracks are received. In the JavaScript function that deals with receiving remote client's media tracks, this variable is checked to decide whether to add the remote client's video and audio streams to the user interface or just the audio stream. The code listing below shows this addition of functionality.

```
} else if (!!audioTrack && audioTrackOnly == true) {
  const audElement = document.createElement('audio');
  audElement.setAttribute('id', 'remoteAudio_' + peerID + 'Temp');
  audElement.srcObject = new MediaStream([audioTrack]);
  document.body.appendChild(audElement);
}
```

In this snippet of code, the conditional statement checks if both the audio stream has been received and that the `audioTrackOnly` variable is set to `true`.

Video and Audio (Default)

In this test, I did not need to make changes to the implementation as it could be carried out under the default circumstances of the dynamic app. This test involved moving

every user into the same area of the user interface to have every user in the same video call. New users were made to join the server and move into the same position until the dynamic app became very slow or unstable. **Figure 3.4** depicts this testing in action, where the fifth user has joined and is connected to the other 4 users.

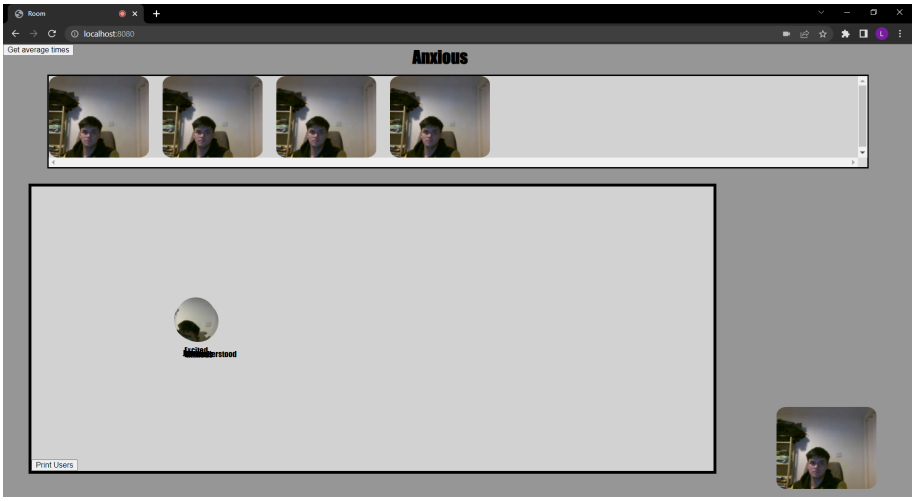

Figure 3.4: In the default mode of the applications, a video call begins between users who are in close range. There are five users grouped together in this example.

Clustered Video Calls

This is the first test that did not involve changes in the implementation. In this test, the video calling component of the dynamic app was tested without the constraint that all users must be in a single video call. This time, I tested the dynamic app to identify how many users could be in video calls in groups of 2, 3, and 4 users. For example, for the groups-of-2 case, every video call had only 2 users. This mode of using the dynamic app relieves the WebRTC part of it as there are much fewer connections involved. **Equation 3.1**, below, gives the total number of connections per cluster, where n is the number of users per cluster, and T is the total number of connections per cluster.

$$T = \frac{n(n-1)}{2} \tag{3.1}$$

As an example, take a cluster size of 3 and a total of 9 users in the dynamic app. In the default mode - all 9 users connecting at once in a large group - there would be $9 \cdot 8/2 = 36$ connections in total. In the clustered mode - 3 clusters with 3 users each - there would be $3 \cdot (3 \cdot 2/2) = 9$ connections in total. The latter case is much better in this instance.

However, this generalises to any size of cluster. The two modes of using the dynamic app can be analyzed in terms of how the total number of connections grows with the number of users. In terms of space complexity and using Big-O notation, in the first mode of the dynamic app - a single cluster with all users - the total number of connections increases in a polynomial fashion, so at a rate of $O(n^2)$. In the clustered

mode of the dynamic app, the total number of connections is a function of the number of clusters multiplied by the number of connections per cluster. Since the number of connections per cluster is a constant for a given cluster size, and the size of each cluster is no more than the number of users, the total number of connections increases in a linear fashion, so at a rate of $O(n)$.

Lower FPS (Frames per Second)

In this test, I altered the JavaScript function that specifies the desired configuration to access the camera device. This function, which is wrapped in a function as part of the Links WebRTC API, calls the MediaDevices API function `getUserMedia` with a configuration object as an argument. This configuration object is where the FPS can be specified. The returned media device stream contains the video stream with the specified FPS value. The code listing, below, displays this function along with the configuration object (`constraints`).

```
function _getCameraReady(camId) {
  const constraints = {
    video: {
      width: {max: 200},
      height: {max: 150},
      frameRate: {max: 60},
      deviceId: camId != "_" ? camId : null
    },
    audio: false
  };
  navigator.mediaDevices.getUserMedia(constraints).then(
    function(stream) {
    localVideoTrack = stream.getVideoTracks()[0];
    ...
```

I tried 3 different FPS values and moved as many users as possible into the same area of the user interface, as with the video and audio test. The 3 values were 30, 10, and 3 FPS, as the video and audio test had already covered the test with 60 FPS.

Lower Resolution

This test involved adjusting the values in the same function described in the "Lower FPS" section, above. However, in this instance, the `width` and `height` properties of the configuration object were altered. With the default mode of the dynamic app using values of 200 and 150 for the width and height, I tried 4 lower pairs of values including 100 by 75, 40 by 30, 20 by 15, and 4 by 3.

Since the sent video stream is at a lower resolution, the receiving clients end up with video streams that are smaller in resolution, which automatically results in smaller video elements being displayed at the top of the user interface. To solve this issue, I set the CSS attribute `transform` of each received video element to the value `scale(x)`, where x is the value of the previous resolution divided by the the new value of the resolution. For example, when 40 by 30 was used, the scale value was 5. **Figure 3.5** shows this test with two users in a video call, where the video element displayed at the top has

a resolution of 40 x 30, whereas the user's own video stream (bottom right) has the default resolution, 200 x 150.

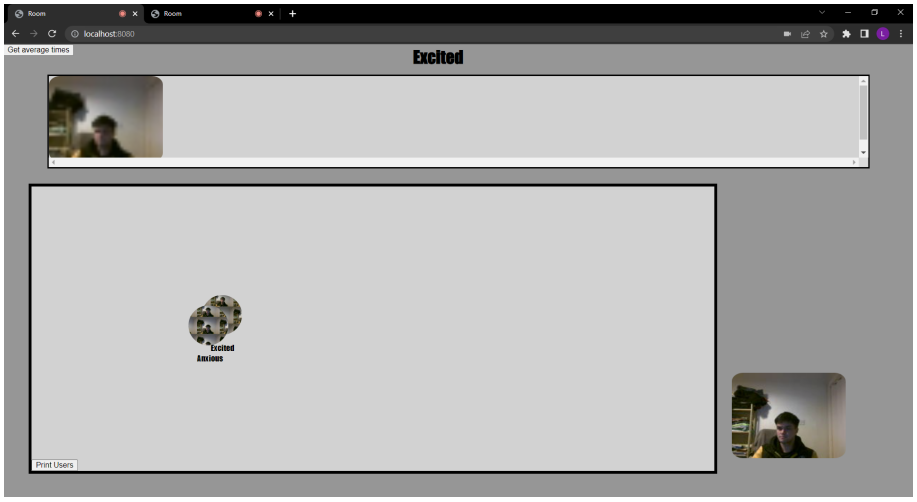

Figure 3.5: In this example, the resolution of each user's video was 40 x 30 pixels. The received stream at the top is at this resolution.

3.3.2 Movement System Test and Measurements

In this section, I describe the single test that tested and measured the movement system of the dynamic app, which uses Links' distributed actor-style concurrency model (described in **Section 2.4**) to broadcast messages to all remote clients. In the current state of the dynamic app, whenever a user moves, they send their updated position to all remote clients connected to the dynamic app server. This test required the movement of all users' characters at once to simulate a use case of the dynamic app where every user is moving simultaneously. However, doing this manually would be impossible on a single computer and very difficult on multiple computers. Therefore, I implemented a modified version of the dynamic app where movement is automated. In this automated version, instead of having the user to press arrow keys to move their character, the dynamic app simulates these key presses in such a way that every user moves at once.

Implementation of the Automated Dynamic App

The main modifications made in the dynamic app to account for automation were in the update function. The first approach thought of was to have the server send direction updates to every user simultaneously every few seconds so that every user moves at the same time. However, this would have required use of the messaging system to broadcast these messages, which would have put more load on the messaging system and rendered the test invalid. Therefore, I had to think of an approach that made every user move simultaneously without using the messaging system as part of the automation.

The approach I came up with involved the use of the current time. Since the current time (in seconds or milliseconds) is the same for all users on the same machine, this was used as a means of synchronizing the actions between all users. At a high level, I implemented the automated version to move all users back and forth between two

locations on the user interface every 4 seconds. More specifically, the current time is checked and two variables are used to store the time modulo 8 and 4 (i.e., time % 8 and time % 4, where time is the current time). If the current time (in seconds) modulo 8 is equal to 0, then the user moves to the first position, and if the current time modulo 4 is equal to 0, then the user moves to the second position. This way, every user moves between the two locations every 4 seconds, so every user ends up moving simultaneously as the current time is a global variable. This high-level description is depicted in the activity diagram in **Figure 3.6**.

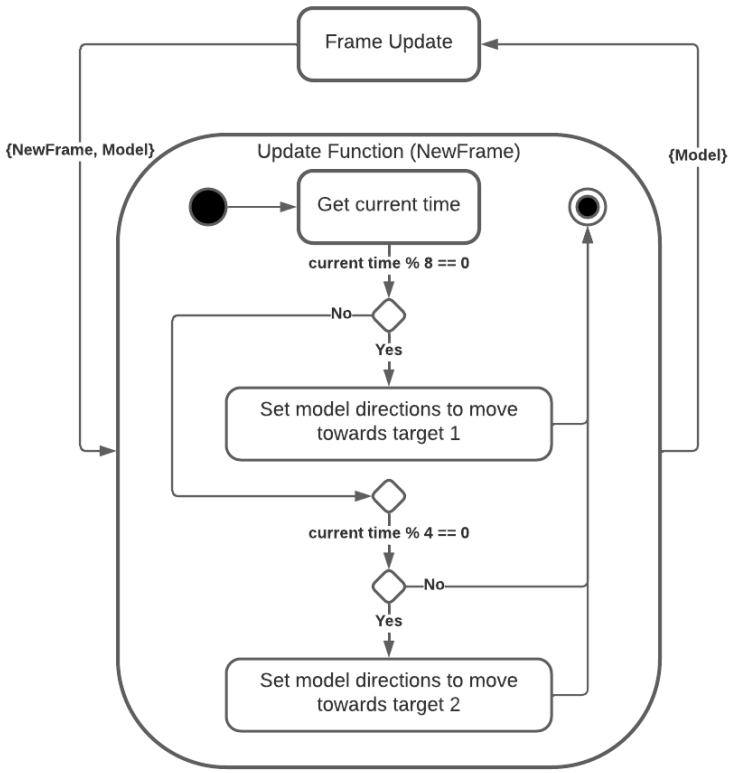

Figure 3.6: The activity diagram shows how the directions of all characters are updated every 4 seconds to one of the 2 targets.

In the actual implementation, in the update function, the most frequent message received is the `NewFrame` message, which is received after every frame update. In the body of this message handler, the client-side Links function `clientTime` is used to retrieve the current time in seconds. Then, two variables are created to store the values of the `clientTime` modulo 8 and modulo 4. The first variable (`clientTime` modulo 8) is checked for equality with 0. If this check passes, the current position of the character and the position of the first target are input into a function that determines the direction to move in (also newly implemented). All users then move towards this target for the next 4 seconds as well as broadcast their updated positions (after every `NewFrame` message) to every user. If this check fails, then the second variable (`clientTime modulo 4`) is checked for equality with 0. If this passes, the character's directions are set to move towards the second target, and the same happens as with the first target. If this check

also fails, then the current direction is used to set the next position of the character, and this new position is broadcasted to all users. Since the `clientTime` is in seconds and the `NewFrame` message is received many times per second, an extra variable is added to the model record to keep track of whether the new target and direction has been set already, so the code to set the new target and direction is not run multiple times when the modulo checks pass.

The activity diagram displayed in **Figure 3.6** depicts part of the process described above where the client chooses a new target for the character to move towards every 4 seconds. Missing from the diagram is the part where the client updates the position of the character and broadcasts this position to all other users. It also excludes the check that is made to ensure that the directions are not updated more than once every 4 seconds.

Use of the Automated Dynamic App

To use the automated dynamic app to complete the movement test, it was as simple as opening new browser windows and opening the automated dynamic app in each one until it performed slow or in an unstable manner (i.e., behaved unpredictably or induced errors). However, each browser window had to be somewhat visible otherwise any hidden windows would be considered inactive and the character in that window would not move.

Since the goal of this test was to test the performance of the movement system and how the dynamic app handled numerous messages being exchanged at once, I decided it would be useful to measure a few metrics involved in the movement system. Such metrics included the number of messages received by each user per second and the time taken to process messages after receiving them (i.e., how long it took to update the position of the character after receiving the message with the position update).

The plan was to calculate averages (over 10 trials) of both of these metrics over a time period of 2 minutes with 9 users. To measure the number of messages received by a given user per second, I added an extra parameter to the function receiving the messages that was incremented by 1 for every message received, as well as a parameter to keep track of the time at which the user joined the dynamic app server. I also implemented a button in JavaScript that printed the number of messages received divided by 120 seconds. Then, after 2 minutes, I clicked this button to get the average number of messages per second over the 2 minutes.

Regarding the second metric, my initial approach was to append an extra field to the received message with the time at which the message was received, then access this field when the message is processed to calculate the time taken to begin processing the message after being received. Also, a field of type list was added to the model record to keep track of the time taken for each message received to be read (i.e., processed). After every message was read, the average of these times was calculated and printed to the console. However, this ended up consuming much of the processing power and worsened the performance of the dynamic app. Therefore, it was important to come up with an approach that did not involve completing lots of computation between receiving messages.

In the second, more efficient approach, I stored each recorded time in a JavaScript array and calculated the average time taken to read messages after receiving them at the end of the 2 minutes using a JavaScript button, as with the first metric measurement approach.

3.4 Test Results

In this section, I present the results of both the video calling tests and the movement system test along with the recorded measurements of metrics.

Throughout testing the dynamic app over the distributed computers (**Section 3.2**), I came across a new bug that was related to the serialisation of images being exchanged between the clients (via the server). Since the image for each user's character icon was converted into an image URL before being sent to other users, this resulted in a very large chunk of data attempting to be serialised. Occasionally, the OCaml JSON library (Yojson) that does this serialisation had trouble serialising the image URLs (likely due to the large size of each URL), which resulted in a console error appearing and the dynamic app failing to function properly (remote client's character did not appear sometimes). As a temporary fix, I removed the code that sent the image URLs. However, a more permanent fix may include reducing the size of the images being sent.

3.4.1 Video Calling Test Results

The results in this section are for each test completed on both a single computer with the server on the same computer, and on multiple computers with the server on a Google cloud server. Except for the results of the test avoiding media altogether, for the results of the tests on the multiple computers, I capped the maximum number of users at 20 due to limited access of the 4 computers available (each test took a while to complete as they had to be repeated 10 times). Also, since the goal of these tests was to examine the capabilities of the video calling part of the dynamic app only, only 1 user was ever moving at once to avoid the overhead cost of message exchange from moving the character.

Avoiding WebRTC and Media Altogether

It appeared that it became tedious trying to find the maximum number of users who could join the dynamic app without any media (and with only one user moving at once). Even at 100 users, the dynamic app still functioned fine and did not slow down significantly, so I concluded that more than 100 users could use the dynamic app in this manner (**Table 3.1**).

Metric	Single Computer	4 Computers
Maximum Users	100+	100+

Table 3.1: Maximum number of users in the dynamic app without WebRTC or media.

Local Media Only

Since this test involved the use of media, I decided to include the number of video streams / elements displayed on the user interface of each user, as this provides insight into how much the video streams take up resources. In the results from this test, only 16 users could use the dynamic app at once on the single computer as an error, depicted in **Figure 3.3**, popped up when too many entities accessed the same audio source at once.

As for the test on the 4 computers, at least 20 users could use the dynamic app at once without it slowing down at all (**Table 3.2**).

Metric	Single Computer	4 Computers
Maximum Users	16	20+
Total Video Streams per User	1	1

Table 3.2: Maximum users and metrics in the dynamic app with local media only.

Audio Only

In the results of this test, it was found that using audio only allowed only 13 users to use the dynamic app on the single computer but at least 20 on the 4 computers. Comparing these results with the video and audio test results below, it seems that the lack of video allowed many more users to use the dynamic app without a performance degrade. Therefore, removing the video streams of certain users in a large video call may be a method of scaling the dynamic app.

Table 3.3 shows the results of this test. Also included in this table is the total number of WebRTC connections amongst all users. These values were calculated using **Equation 3.1**, and they were included to provide a better picture of the WebRTC metrics. However, the bottleneck on the single computer is likely the number of audio elements as there are 12 audio streams being played on each browser tab, and 13 browser tabs, so $13 \cdot 12 = 156$ audio elements in total streaming audio on the computer.

Metric	Single Computer	4 Computers
Maximum Users	13	20+
Total Audio Streams per User	12	19+
Total WebRTC Connections	78	190+

Table 3.3: Maximum users and metrics in the dynamic app with audio only.

Video and Audio

For video and audio, the maximum number of users able to use the dynamic app at once was 8 on the single computer, and 15 on the 4 computers. However, since one of the 4 computers is not as capable, it was only able to support up to 3 users until it became extremely slow, while the other 3 could support 4. The likely reason for the single computer failing at such a low number is the number of video element playing video streams over all 8 browser tabs. Each browser tab is playing 7 video streams from the other users and 1 from itself (bottom left self camera), so there are $8 \cdot 8 = 64$ video streams in total on the computer. These results are displayed in **Table 3.4**.

Metric	Single Computer	4 Computers
Maximum Users	8	15
Total Video Streams per User	8	15
Total WebRTC Connections	28	105

Table 3.4: Maximum users and metrics in the dynamic app with video and audio.

Clustered Video Calls

These results are split into 3 sets of results: one for clusters of 2 users, clusters of 3 users, and clusters of 4 users. In the results of the test involving clusters of 2 users, I found that I was able to have 16 users using the dynamic app at once on the single computer, and at least 20 users on the 4 computers. The single computer could only handle up to 16 users because no more than 16 entities can access the same audio source at once before inducing an error, as depicted in **Figure 3.3**. Also, there are only half as many WebRTC connections as users because each cluster has a single connection. **Table 3.5** shows these results.

Metric	Single Computer	4 Computers
Maximum Users	16	20+
Total Video Streams per User	2	2
Total WebRTC Connections	8	10+

Table 3.5: Maximum users and metrics in the dynamic app with clusters of 2 users.

As for the results of the test on clusters of 3 users, I was able to have only 16 users as before on the single computer due to the audio source restriction. However, on the multiple computers, I was still able to get at least 20 users using the dynamic app at once. In terms of WebRTC connections, each cluster had 3 connections and there were 3 users per cluster, so for 16 users there were 5 clusters, which results in $5 \cdot 3 = 15$ connections. For 20 users there were 6 clusters, so 18 connections in total (the 2 other users did not form a valid cluster). **Table 3.6** shows these results.

Metric	Single Computer	4 Computers
Maximum Users	16	20+
Total Video Streams per User	3	3
Total WebRTC Connections	15	18+

Table 3.6: Maximum users and metrics in the dynamic app with clusters of 3 users.

Finally, for clusters of 4 users, I found that the dynamic app could allow only 15 users as a maximum on the single computer, and at least 20 users on the multiple computers. On the single computer, at 15 users, each browser showed 4 video streams except the 3 that were not in a cluster, which means that the computer was displaying $(12 \cdot 4) + 3 = 51$ video streams in total. This is likely the reason it could not allow any more users without it becoming unstable. As for metrics, the single computer had

Metric	Single Computer	4 Computers
Maximum Users	15	20+
Total Video Streams per User	4	4
Total WebRTC Connections	18	30+

Table 3.7: Maximum users and metrics in the dynamic app with clusters of 4 users.

$3 \cdot 6 = 18$ connections in total as each cluster had 6 connections. The multiple computer had $5 \cdot 6 = 30$ connections for 20 users. These results are shown in **Table 3.7**.

The most important point to take from these results is that splitting the video calls into groups of smaller size results in more users able to use the dynamic app at once. Therefore, applying this mode to the dynamic app (i.e., by limiting the number of users in a single video call) may allow it to scale.

Lower FPS (Frame per Second)

These tests involved running the dynamic app in the same mode as the "Video and Audio" tests such that every user is in the same video call, but for every test run, the FPS of each user's video streams were varied. **Table 3.8** shows these results. As can be seen in the table, there is a correlation between the FPS and the maximum number of users. Particularly, for the multiple computers, there is a large jump in maximum users from 30 FPS to 10 FPS. Lowering the FPS of users to 10 may be a method of scaling the dynamic app when there are many users in a video call. An old study [11] showed that lowering the frame rate of video reduces computation and use of bandwidth, which agrees with my finding that the dynamic app can handle more users streaming video at lower FPS.

Metric	Single Computer	4 Computers
Maximum Users on 30 FPS	8	15
Maximum Users on 10 FPS	9	18
Maximum Users on 3 FPS	10	19

Table 3.8: Maximum users in the dynamic app with varying video FPS.

Lower Resolution

These tests involved running the dynamic app with the users streaming their videos at varying resolutions. **Table 3.9** shows the results of these tests. As with FPS, there is a correlation between the resolution and the maximum number of users. As the resolutions of the users' videos were decreased, the maximum users increased. However, there does not seem to be an improvement when changing from a resolution of 20 x 15 to 4 x 3, so it would not make sense to use such a small resolution in the dynamic app. Using the data gained in these tests, one modification that could be made to the dynamic app to scale it would be to reduce the resolution of certain users when the size of the video call gets large (e.g., users who are not talking have their video resolution decreased to 40 x 30).

Metric	Single Computer	4 Computers
Maximum Users on 100 x 75	8	17
Maximum Users on 40 x 30	9	19
Maximum Users on 20 x 15	9	20
Maximum Users on 4 x 3	9	20

Table 3.9: Maximum users in the dynamic app with varying video resolutions.

3.4.2 Movement System Test Results

After using the automated version of the dynamic app to test the movement system on the single computer, I found that the dynamic app could only support up to 8 users moving at once before the remote characters on each user's screen became several seconds behind, and the movement of each character became very jittery. In **Table 3.10** I present the maximum number of users able to move at once as well as the average number of messages per second and the average time taken between receiving messages and reading them. As can be seen in the table, each message is taking over 22 seconds to be read by the update function after being received by the receive function. This is due to the high influx of messages in a short period of time - the MVU loop can retrieve only so many messages every second from the function that receives and stores the messages, so the queue of messages in the receive function gets larger as time passes. A movement system design that resulted in much fewer messages being sent would result in this problem being mitigated.

I should also note that Links does virtually no optimisation. Although it benefits on the client side from highly optimised JavaScript engines, it is interpreted on the server, which results in very slow computation. This is another reason to have fewer messages exchanged between clients and the server.

Metric	Mean	Variance
Maximum Users	8	0
Message Rate (messages per second)	128	11.33
Time Taken to Read Message (seconds)	22.62	2.49

Table 3.10: Maximum users and metrics in the dynamic app with every user moving at once on the single computer over 10 trials, where the message rate and time taken to read messages were recorded while testing with 9 users.

Chapter 4

Dynamic App Changes

In this chapter, I detail the design and implementation changes made to the dynamic app to fix the software bugs described in **Section 2.7.1** and optimise the performance of it to maximise the number of users able to use the dynamic app at once.

4.1 Software Bug Fixes

The bug fixes listed below were the main and most noticeable ones, but there were numerous smaller bugs present in the dynamic app that were fixed also.

Bug 1 Fix: Clients attempt to connect while already connected

This bug was related to the WebRTC part of the dynamic app. If one user (user 1) moved into the range of another user (user 2) and began a WebRTC connection with them, then if either user moved again, the browser crashed and reported a WebRTC error regarding the SDP being set in the wrong state. This error occurs because one of the clients attempts to set their "remote description", although it has already been set.

The reason for the client attempting to set their "remote description" while it was already set was because the client attempted to begin a new WebRTC connection after every movement within the range of the other user. This happened because there was no check made to ensure that there was not an existing connection between the users before attempting to begin a new connection. To fix this, I implemented a function in the JavaScript file that checks whether the client is already connected to another client with a provided ID. I updated the `connectToUser` function in the Links WebRTC API to check that this function returns false before beginning a new connection with the client. This new check is shown in the code snippet below.

```
fun connectToUser(peerID) {
    var localID = JSWebRTC.getLocalID();
    if (not(checkIfConnectedToPeer(peerID))) {
        setUpNewPeer(peerID, "_", false);
        ...
```

Bug 2 Fix: Character movement functions incorrectly

This bug was rooted in the movement system of the dynamic app. If a user moved along an axis (e.g., pressed the left or right arrow key to move left or right), and subsequently pressed the opposite arrow key while the other key was pressed (i.e., to stand still), then the character would move in the direction guided by the latest key press, instead of staying still. Also, if both keys were pressed and one was let go, then the character stopped moving, instead of moving in the direction indicated by the key that was still being pressed.

This bug existed because of a design flaw in the movement system. The flaw was that the model only kept a record of 2 aspects of movement: horizontal and vertical movement (i.e., the model record stored two fields representing which direction the character was moving horizontally (either up, down, or still) and vertically (either left, right, or still), respectively), so if the user pressed the left arrow key to move left, then pressed the right arrow key while the left arrow key was still pressed, then the horizontal direction would be updated to right, and the fact that the left arrow key was still pressed was forgotten.

This was fixed by replacing the two fields in the model record with a single record that stores information for all 4 directions, instead of just 2. This new record has 4 fields, each set to a value of type `boolean`, which is true if the character is moving in the direction represented by that field. Also, if the user lets go of an arrow key, a message of type `StopChar` with the direction value attached is sent to the update function to change the `boolean` value of the relevant direction field in the direction record (which is in the model record) to false. The code snippet below shows the code that runs when a `MoveChar` message is received in the update function, which uses the newly implemented `setCharacterDirection` function to set the relevant direction field to true.

```
case MoveChar(dir) ->
    var newDirections = setCharacterDirection(room.directions, dir);
    ((room with directions = newDirections), MvuCommands.empty)
```

Bug 3 Fix: Message handler leaks messages

This bug was related to the messaging system of the dynamic app. If two or more users connected to the dynamic app, often the user that joined the server last was missing at least one of the remote user's characters on their screen. This was because the function that waited for and stored messages received from remote clients (to be retrieved by the MVU loop) in the dynamic app had its own bug - if more than 1 message was received by the function from remote clients before the MVU loop retrieved any, then all messages except the last message received were stored, and the prior ones removed.

```
fun drain(msgs, pids) {
  switch (msgs) {
    case [] -> (msgs, pids)
    case msg :: msgs ->
      switch (pids) {
        case [] -> (msgs, pids)
        case pid :: pids ->
```

```
...
```

In the implementation, upon receiving a message from a remote client, this receiving function called a function called `drain`, which, in simple terms, sent each message to each process waiting to retrieve a message. The point of failure in the function is shown in the code snippet above. If there are no processes waiting to retrieve a message, then the function ends at the empty list case (after `switch (pids)`), so the message (`msg`) is removed from the list without being sent to any process. I fixed this by renaming the `msgs` function parameter to `messages`, then replacing `msgs` in the lower empty case with this new parameter name. This way, no messages could be lost without being sent to a process.

Bug 4 Fix: Dynamic app freezes when two users move into each other

This bug relates to the way that users began connections with each other. Originally, I implemented the connection system such that when one user moves into the range of another user, they (the mover) begins the WebRTC connection with the other user. However, in the case that two users move into each other at the same time, since both users start a WebRTC connection with each other simultaneously, a synchronization issue arises and neither user completes the connection.

I fixed this bug by altering the design of the Links WebRTC API. Firstly, I changed the ID assigning system such that, instead of using a function to generate a pseudo UUID to use as each client's ID, each client gets their ID by requesting one from the server, and receives their ID based on how many clients have joined the server (e.g., the 4th user to join will have an ID of 4). Using this ID system, I updated the connection process such that when two users move into each other simultaneously and begin video calls, the user with the greater ID continues the connection process, and the other user discontinues. This case occurs when the client (client 1) receives a `ConnectionRequest` message from another client (client 2), but has already issued a `ConnectionRequest` message to client 2, so results in comparing the IDs of both clients to determine whether to continue the connection or not, as shown in the code snippet below.

```
} else if (localID > peerID) {
  JSWebRTC.updateConnectionID(peerID, connectionID);
  ...
```

The update of the connection ID in the code snippet is also an update to the dynamic app design - each connection is associated with its own ID to further prevent synchronization issues. Each `ConnectionRequest` message has associated with it the connection ID, so that each client has the same IDs associated with the connection.

Bug 5 Fix: Client disconnects immediately after connecting

This bug also relates to the WebRTC connection process and occurs due to the way that connections are created and destroyed. When a client (client 1) moves into the range of another client (client 2), the video call that is expected to begin does not, noticeable by the lack of video streams appearing on screen. When moving the character, the client is constantly checking whether other clients' characters are in range and beginning WebRTC connections with them if so, or deleting connections otherwise. Occasionally,

client 1 completes the WebRTC connection before client 2 notices the position-update message from client 1 that puts client 1's character in their vicinity, so client 2 sees client 1's character outside their range after the WebRTC connection is completed and deletes the connection immediately (as each client is constantly checking distances and deleting connections if a remote client's character is outside the range).

This bug was fixed by introducing a new connection checking function that ensure that each WebRTC connection has existed for at least one second before allowing the connection to be destroyed. This function was included in the Links WebRTC API as a wrapper for the JavaScript function shown in the code snippet below.

```
function _oneSecondElapsed(id) {
  if (!peerConnections[id]) return false;
  const timeConnected = peerConnections[id].timeConnected;
  const currentTime = Date.now();
  const timeElapsed = currentTime - timeConnected;
  return timeElapsed >= 1000;
}
```

Although this worked for stopping connections from being immediately deleted, this gave rise to another bug where clients connected immediately after disconnecting. To fix this, the same technique was applied in the other direction, so clients must be disconnected for at least a second before connecting again.

4.2 Optimisations

In this section I utilise the results obtained in the empirical limit testing (**Section 3.4**) to make changes to the dynamic app that optimise its performance. These include improvements in the video streaming and movement system components of the dynamic app.

4.2.1 Video Calling Changes

There were multiple parameter settings used in the empirical limit testing that yielded performance improvements, such as lower FPS, lower resolution, audio only, and clustered video calls. I updated the design of the dynamic app to include resolution lowering where possible to allow video calls to scale better. In particular, if multiple users are in a video call, and any of them stops speaking for 5 seconds, then the resolution of their video output is lowered to 40 x 30. Then, if any of them speak again, their resolution is reset to the default (200 x 150).

The main logic of this feature is implemented in the JavaScript function `_getMicReady` that is called when setting up the user's microphone. I added functionality to this function that makes use of the Web Audio API [4] (as others have [12]) to record 10 values of audio output levels (measures of volume) by the users over 5 seconds in an array (i.e, every new output value pushes the oldest output value out as in queue data structures). Firstly, an `AudioContext` object is created, which represents an audio-processing graph made up of nodes that interact with each other. In the case of this

audio-measuring feature, I used this object to create two nodes within the graph: a media streaming source node and an analyser node. This former of these takes a media streaming source (already retrieved via the `getUserMedia` method of the MediaDevices API in `_getMicReady`) and allows other nodes of the graph to process and manipulate the audio output. The latter takes this audio output and exposes audio time and frequency data to use for processing (i.e, measuring volume). **Figure 4.1**, below, visualises this graph, where the analyser node exposes an API for access to real-time frequency and time domain data.

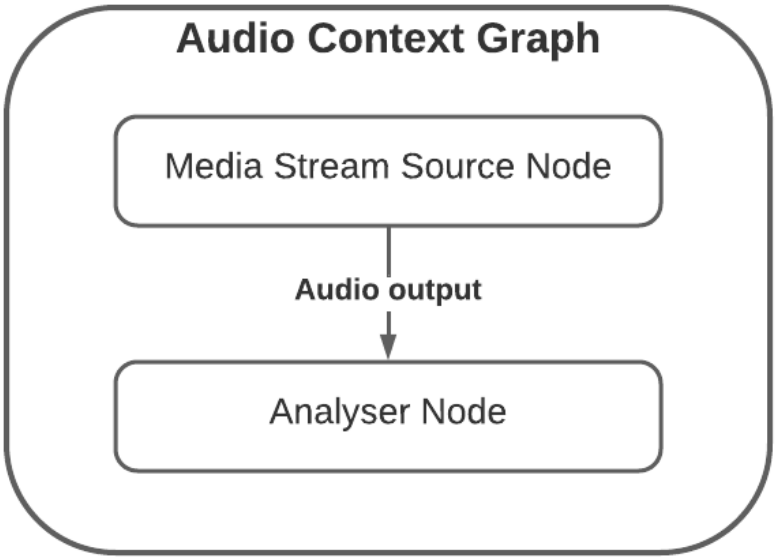

Figure 4.1: The audio-processing graph used in this instance. It contains a media source node that represents a source of audio and an analyser node that uses the audio output of the media source node to provide an API for real-time frequency and time domain data.

The analyser node has a method called `getFloatTimeDomainData` that copies the current waveform / time-domain data into a provided array. In my implementation, I use a definition of volume that takes the square root of the sums of the squares of the values (amplitudes) in this array. This resulting value is pushed into an array that contains 10 values of output volume over 5 seconds as described in the previous paragraph. This array is exposed to the Links API via a wrapped function so that the volumes can be accessed by the dynamic app.

In short, the dynamic app code checks this array of volumes constantly and checks whether at least one of the values is above a certain threshold (0.001). If this check fails, the resolution of the user's video streaming output is reduced to 40 x 30 by calling a new function in the Links WebRTC API called `updateResolution`. This is simply a wrapper of a JavaScript function I implemented that calls `_getCameraReady` with the new desired resolution to update the user's output video stream, then updates the video track being streamed on each of the client's WebRTC connections with this new stream. This update of each of the WebRTC connection video streams (for each remote client)

is non-trivial, and requires fetching each WebRTC connection's RTCRtpSender object and calling `replaceTrack` on it to update the live video stream.

These new functions, along with some CSS code to reposition and scale each user's video streams as their resolutions are updated provided the required functionality to be able to dynamically decrease and increase the resolution of a user's video output while in a WebRTC video call.

4.2.2 Movement System Changes

In **Section 3.4.2**, I presented the results of the test on the automated version of the dynamic app that moved every connected client's characters simultaneously. Since only 8 users could move at once before the dynamic app became unstable and the time taken to read position-update messages after being received was over 20 seconds, this was considered a severe performance bottleneck as up to 15 users could be a single video call at once. Since I also measured the rate of messages being received by a single client in the empirical testing, I discovered that there were too many messages being exchanged such that each client was receiving messages faster than they could process them.

To reduce the number of messages being sent by each client, I changed the design of the movement system. Instead of having each character move using constant direction updates (i.e., repeatedly pressing different arrow keys to change direction), I set target positions for each character to move towards (simulating click movement). Then, each character broadcasts messages to other users only when they change target position, instead of repeatedly broadcasting their updated positions numerous times per second. Based on the current position and target position of each character, the new position is updated using a function called `determineVector`.

Since the new movement system in the automated version of the dynamic app relies on movement through setting target positions and moving towards them, the original version that takes user input had to allow users to specify a point on the screen to move their character towards. The simplest way for users to do this was by having them click their mouse at a desired location on their screen. However, although the Links MVU library supported key presses, it did not have an event handler for mouse clicks. Therefore, I had to make a change in the MVU subscriptions file to include a function that acted as an event handler for mouse clicks, in the same fashion that the key press handler was implemented. After updating the dynamic app to wait for mouse clicks to pass a new message of type `Click` to the update function, the program was able to make updates to the target position of the user's character after every mouse click, then broadcast it to all remote clients. As can be seen in the code snippet below, the `Click` message also contained the desired target position for the character to move towards.

```
case Click(xPos, yPos) ->
  var targ = (x = intToFloat(xPos), y = intToFloat(yPos));
  var targPos = TargetPosition(id = room.charData.id, target = targ)
  serverPid ! BroadcastMessage(targPos)
  ...
```

Chapter 5

Discussion

In this evaluation sections of this chapter, I present an evaluation of the work undertaken in this project. To ensure that my evaluation was robust and my conclusions were reliable, I followed ACM's empirical evaluation guidelines [1] as well as Gernot Heiser's list of systems benchmarking crimes [9].

5.1 Evaluation of Test Methodologies

I was unable to test the dynamic app to its full potential due to lack of computers. To be able to yield a true maximum for the number of users who can use the dynamic app at once, I would require a computer for each user, as this is what would simulate a real use case. However, with the 4 computers available, I was able to investigate which circumstances and dynamic app parameters yielded the greatest maximum users. Since every test was completed on the same 4 computers, this made for a fair comparison between the results of each.

In terms of using a sufficient number of trials, 10 trials for each test was adequate as the results did not vary much and there were no noisy results. This number of trials was enough to characterise the behaviour of the movement system.

Lastly, the methodology section offers an in-depth description of the steps taken to complete the empirical tests. Therefore, there is sufficient information for future studies to repeat the tests I completed.

5.2 Evaluation of Test Results

Although the test that avoids WebRTC and media altogether was used as a control test to show that the performance bottleneck was not caused by any idle inefficiencies in the MVU framework, some of the other tests could also be considered as control tests for different reasons. The local media only test provided a value for the maximum possible number of users on a single computer, which was 16, as seen in **Table 3.2**. The results of the audio only test (**Table 3.3**) showed that the video part of the video call was not the main bottleneck because the maximum number of users was still less

than the maximum possible number of users on a single computer with media (16), even though it shows an improvement. This indicates that there is another factor that is creating the performance bottleneck, which is not video. The results of the video and audio tests (**Table 3.4**) were considered as the baseline to be compared against, as these represented the default configuration when using the dynamic app.

Since the results of the clustered video call tests (**Table 3.5**, **Table 3.6**, and **Table 3.7**) showed that the dynamic app could scale much better when video calls were split up into smaller groups, and therefore less WebRTC connections in total, it would make sense to state that the main bottleneck of the dynamic app, when ran on 4 computers on a cloud server, was the large number of WebRTC connections.

Despite discovering that the main bottleneck was the large number of WebRTC connections, there is not a simple solution to mitigating this when considering software only. Also, it would not make sense to set a hard limit on the number of users who can be in the same video call because there may be situations where groups of over 15 people wish to be in the same video call (e.g., for group sessions, presentations, etc.). Therefore, to significantly scale the dynamic app, more advanced technology such as a media server that acts as a selective forwarding unit (SFU) would be required.

With this software constraint, I identified that I could scale the dynamic app to at least a few more users by taking advantage of the video part of the video call, which was found to contribute a little to the performance bottleneck via the audio only test results (**Table 3.3**). Lowering the FPS of the video stream output resulted in 4 more users able to be in the same video call at once (**Table 3.8**) compared to the baseline video and audio test results (**Table 3.4**), and lowering the resolution to 20 x 15 resulted in 5 more users (**Table 3.9**). These results indicated that a video call could scale to more users if the resolution or FPS of the video stream output from users was reduced.

Regarding the presentation of results, each of the video calling test results were very stable and remained unchanged over the 10 trials. For this reason, it was unnecessary to report data distributions for these.

5.3 Evaluation of Video Calling Changes

A complete test of the performance of the dynamic app with the video calling component optimised would have required an implementation of the dynamic app similar to the automated one used in **Section 3.3.2**, but instead simulating a likely use case where each client would output audio at certain times or randomly. Due to time constraints, I did not implement such a version of the dynamic app, so I came up with a method of testing the optimised dynamic app that did not require as much work. Usually there is only one person talking at once in a group conversation, so I hard-coded a variant of the dynamic app that had only a single client with the default resolution (200 x 150) and the rest as the lowered resolution (40 x 30).

When testing this variant of the dynamic app on the same 4 computers and cloud server as in the empirical testing phase, I was able to get 17 users into a single video call (2 more than in the original dynamic app). I have decided not to include a table for

this data as it is a single number which, as with the previous tests on the video calling component, remained unchanged over 10 trials.

5.4 Evaluation of Movement System Changes

Having successfully implemented a new version of the movement system that involved moving the character based on target positions rather than constant direction changes in an attempt to reduce the load of message exchange, I completed two sets of tests on the same single computer as in the testing of the original movement system: one to determine the maximum number of users that could move at once, and the other to measure the average number of messages and time taken to read each message on a single client with 9 users. **Table 5.1** shows the results of these tests, where the message rate and time taken to read messages were measured while testing with 9 users.

Metric	Mean	Variance
Maximum Users	14	0
Message Rate (messages per second)	2	0
Time Taken to Read Message (seconds)	0.05	5.44e-5

Table 5.1: Maximum users and metrics in the optimised dynamic app with every user moving at once on the single computer over 10 trials, where the message rate and time taken to read messages were recorded while testing with 9 users.

Comparing the results from **Table 5.1** with the results from **Table 3.10** on the original movement system, it is clear that the optimised movement system greatly outperforms the original movement system as the former can have 6 more users moving at once. The reason for this is also clear in the tables, as the new system takes significantly less time to begin processing each message after reading it in the receive function (0.05 seconds compared to 22.62 seconds). The raw data obtained over each of the 10 trials for both the original dynamic app and the optimised dynamic app is displayed in **Table A.1** and **Table A.2** in **Appendix A**.

Also, I tested the optimised dynamic app using the automated version over two computers (top two in **Section 3.2**) on the cloud server and found that 18 users could move at once before the dynamic app became slow and unstable. It was expected that 28 users (14 on each computer) would be able to use the dynamic app at once, which suggests a bottleneck outwith the movement / messaging system, as there were still very few messages being exchanged.

5.5 Conclusion

The goal of this project was to improve the dynamic app developed prior to this project by fixing the five software bugs identified in **Section 2.7.1** and scaling it such that more users are able to use the dynamic app at once. As discovered in the description of the bug fixes in **Section 4.1** and in the evaluation of the dynamic app optimisations in **Section 5.3** and **Section 5.4**, I implemented fixes for every bug and completed the implementations

for the desired dynamic app optimisations. The completed implementations for each optimisation (video calling and movement system) resulted in more users being able to use the dynamic app at once (2 more users in a video call and 6 more users being able to move at once). Therefore, by these figures, the goal was met.

However, in terms of proper evaluations, the test on the video calling optimisation (lower resolution for non-speakers) did not properly represent a real use case, as an assumption was made that only 1 user would be speaking at any one time. In a real use case, it may be the case that more than 1 user is making sounds (e.g., 2 users speak over each other, other users have loud background noise), so simulating only 1 user as talking at all times is not entirely accurate. As hinted in the evaluation (**Section 5.3**), a more accurate result could be found by implementing a variant of the dynamic app that simulated various users talking, either randomly or systematically according to a probability distribution.

5.6 Possible Extensions

Although the optimisations made to the applications yielded improvements, there is still much scope for further improvement. Further work could extend on the rest of the results in **Section 3.4** and utilise lower FPS or make WebRTC communications audio only.

In terms of the movement system, if click movement was undesirable, a different optimisation could be made to the movement system using key presses to change direction, but instead of sending the new position of the character after every frame, send the new position much less frequently and have the receiving user perform interpolation between the updated positions to mimic smooth movement.

If the goal is to scale the dynamic app to significantly more users in a single video call while still using WebRTC, it would be necessary to make use of more expensive technology such as a media server as a selective forwarding unit (SFU) or a multipoint control unit (MCU), since the mesh topology network used in WebRTC connections in this project would not suffice as the number of WebRTC connections grows too large for the server.

Also, the measure of volume used in the implementation of the optimised video calling was based on a time domain graph, as opposed to a frequency domain graph. Since the goal was to measure the volume of the voice of humans, it would have been more accurate to use frequency domain data with human voice frequencies. This would be an option for improving the detection system for users' output volumes.

Lastly, since the tests were completed on static video streams (i.e, users are only moving their mouth usually), another interesting investigation would involve completing the same tests but with as much movement as possible in the video streams of each user in an attempt to check whether this affected the performance of the dynamic app. Without other users, this could be done by simulating a very dynamic video stream by changing the colour of each pixel dramatically.

Bibliography

[1] ACM. Empirical Evaluation Guidelines. https://www.sigplan.org/Resources/EmpiricalEvaluation/. Last accessed: 6th April 2023.

[2] Adam Rehn. 2021. WebRTC Stream Limits Investigation. https://tensorworks.com.au/blog/webrtc-stream-limits-investigation/.

[3] Alberto Gonzalez. 2017. A Guide to: WebRTC Media Servers Open Source Options. https://webrtc.ventures/2017/11/a-guide-to-webrtc-media-servers-open-source-options/.

[4] Audio Web API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API.

[5] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, Morgan Kaufmann Publishers Inc., p. 235–245. https://dl.acm.org/doi/10.5555/1624775.1624804.

[6] Elm website. https://elm-lang.org/.

[7] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *FMCO (Lecture Notes in Computer Science, vol 4709)*, Springer, pp. 266–296. https://doi.org/10.1007/978-3-540-74792-5_12.

[8] Gather Website. https://www.gather.town/.

[9] Gernot Heiser. Gernot's List of Systems Benchmarking Crimes. https://gernot-heiser.org/benchmarking-crimes.html.

[10] Google Git. 2023. Source Code of WebRTC. https://chromium.googlesource.com/chromium/src/third_party/+/master/blink/renderer/modules/peerconnection/rtc_peer_connection.cc#155. Last accessed: 18th March 2023.

[11] Jeremiah Scholl, Peter Parnes, John D. McCarthy, and Angela Sasse. 2005. Designing a Large-Scale Video Chat Application. In *Proceedings of the 13th Annual ACM International Conference on Multimedia*, MULTIMEDIA '05, Association for Computing Machinery, p. 71–80. https://doi.org/10.1145/1101149.1101160.

[12] Jim Fisher. 2021. Measuring audio volume in JavaScript. https://jameshfisher.com/2021/01/18/measuring-audio-volume-in-javascript/.

[13] Jirka Hladis. 2019. Multi User Video Chat With WebRTC. https://www.dmcinfo.com/latest-thinking/blog/id/9852/multi-user-video-chat-with-webrtc.

[14] Lewis Raeburn. 2022. A Spatial Metaphor for Dynamic Video Calling in Links. 4th Year Dissertation for Master of Informatics. https://links-lang.org/papers/undergrads/ug4_20223144.pdf.

[15] MediaDevices Web API. https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices.

[16] NodeJS Website. https://nodejs.org/en/.

[17] Philip J. Fleming and John J. Wallace. 1986. How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results. *Commun. ACM 29*, 3, 218–221. https://doi.org/10.1145/5666.5673.

[18] React website. https://reactjs.org/.

[19] Simon Fowler. 2020. Model-View-Update-Communicate: Session Types Meet the Elm Architecture. In *34th European Conference on Object-Oriented Programming (ECOOP 2020)*, Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 14:1–14:28. https://doi.org/10.4230/LIPIcs.ECOOP.2020.14.

[20] Stanislav Zayarsky. 2022. How many participants can we place in one WebRTC peer 2 peer room. https://trembit.com/blog/how-many-participants-can-we-place-in-one-webrtc-peer-2-peer-room/.

[21] The Links Programming Language. https://links-lang.org/.

[22] WebRTC Web API. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API.

Appendix A

Raw Movement System Test Results

Trial	Message Rate (messages per second)	Time Taken to Read Message (seconds)
1	123	20.88
2	133	21.30
3	128	25.48
4	125	22.34
5	131	21.11
6	128	23.55
7	129	22.87
8	127	20.94
9	124	24.21
10	132	23.54

Table A.1: Raw data used for **Table 3.10**.

Trial	Message Rate (messages per second)	Time Taken to Read Message (seconds)
1	2	0.05
2	2	0.06
3	2	0.04
4	2	0.05
5	2	0.05
6	2	0.06
7	2	0.04
8	2	0.06
9	2	0.05
10	2	0.05

Table A.2: Raw data used for **Table 5.1**..