

# **A Spatial Metaphor for Dynamic Video Calling in Links**

*Lewis Raeburn*



Minf Project (Part 1) Report  
Master of Informatics  
School of Informatics  
University of Edinburgh

2022

# **Abstract**

When developing a web application that includes video calling as a feature, it is essential that there exists functionality to support this in the programming language being used. However, Links does not yet support live streaming of video or audio between users.

In this report, I discuss my design and implementation of a live streaming API in Links and a dynamic video calling web application, which I developed using this API. Briefly, the API allows a Links programmer to write a program that finds all the media devices connected to a user's computer, uses a specific camera and microphone, and begins a video call with another user. When using the dynamic video calling web application, users can move around the page using the arrow keys on their keyboard and begin video calls with other users who are nearby.

Near the end of this report, I evaluate the system mostly in terms of error checking, performance, and scalability. I also present conclusions and a discussion of potential future work regarding this project.

# **Research Ethics Approval**

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

## **Declaration**

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

*(Lewis Raeburn)*

# Acknowledgements

I would like to thank my supervisor, Sam Lindley, for his constant support and feedback throughout my project.

I would also like to thank Links developer, Simon Fowler, for assisting me with Links' actor model and MVU framework.

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Project Motivation . . . . .	1
1.2	Project Goals . . . . .	2
1.3	Summary of Contributions . . . . .	2
1.4	Report Structure . . . . .	3
<b>2</b>	<b>Background</b>	<b>4</b>
2.1	Fundamentals of Web Development . . . . .	4
2.2	Web Development in Links . . . . .	5
2.3	Web APIs Used for Video Calling . . . . .	5
2.3.1	The Connection Process . . . . .	6
2.4	Accessing Web APIs in Links . . . . .	7
2.5	Distributed Actor-Style Concurrency Model . . . . .	9
2.6	Model-View-Update Framework in Links . . . . .	9
2.7	Existing Systems . . . . .	10
2.7.1	Gather . . . . .	10
2.7.2	Mozilla Hubs . . . . .	11
2.7.3	GroupRoom . . . . .	12
2.8	Terminology . . . . .	12
<b>3</b>	<b>Design</b>	<b>14</b>
3.1	Using the Live Streaming API . . . . .	14
3.1.1	Using the Media Devices Functions . . . . .	15
3.1.2	Using the WebRTC Connection Functions . . . . .	16
3.2	How the Live Streaming API Works . . . . .	17
3.2.1	How Media Devices are Accessed . . . . .	17
3.2.2	How WebRTC Connections are Completed . . . . .	18
3.3	Using the Dynamic Video Calling Application . . . . .	18
3.4	Design of the Dynamic Video Calling Application . . . . .	21
3.4.1	User Interface . . . . .	21
3.4.2	Communicating Data Between Clients . . . . .	22
3.4.3	Incorporating the Live Streaming API . . . . .	24
3.5	Design Challenges and Justification . . . . .	24
<b>4</b>	<b>Implementation</b>	<b>26</b>
4.1	Live Streaming API . . . . .	26

4.1.1	Accessing Media Devices . . . . .	26
4.1.2	WebRTC Connection Details . . . . .	28
4.2	The Dynamic Video Calling Application . . . . .	31
4.2.1	Model, View, and Update . . . . .	31
4.2.2	Multiple Clients . . . . .	33
<b>5</b>	<b>Evaluation</b>	<b>35</b>
5.1	Live Streaming API . . . . .	35
5.1.1	Error Checking . . . . .	35
5.1.2	Broadcasting Data . . . . .	36
5.2	Dynamic Video Calling Application . . . . .	36
5.2.1	Performance . . . . .	36
5.2.2	Communicating Image URLs . . . . .	37
5.2.3	Scalability . . . . .	37
5.2.4	MVU's Dispatch Feature . . . . .	37
5.2.5	Lack of Feature Support . . . . .	37
<b>6</b>	<b>Discussion</b>	<b>39</b>
6.1	Conclusions . . . . .	39
6.2	Future Developments . . . . .	39
6.2.1	Scalability and Performance . . . . .	39
6.2.2	Implement Same Application in React, Standard Links, and JavaScript Alone . . . . .	40
6.2.3	Scale to Multiple Users Without Video Server . . . . .	40
6.2.4	User Study . . . . .	40
	<b>Bibliography</b>	<b>41</b>
<b>A</b>	<b>Live Streaming API Documentation</b>	<b>43</b>

# Chapter 1

## Introduction

### 1.1 Project Motivation

Links [9] is a functional programming language designed to ease web development. It provides a single language for the three tiers of web development: the web browser, the web server, and the database (more on this in Section 2.2). Links also has a wide variety of features to make web development simpler such as functions to modify the web page (by modifying the DOM), user input handlers to detect events such as mouse clicks, as well as concurrency.

However, a feature Links does not support is the live streaming of user media, specifically in the form of video and audio. Today, the capability of communicating through live video conferencing is a key feature in web applications that involve social interaction [3].

Currently, a web developer who wishes to use Links to build a web application that, for example, allows users to make video calls with each other would not be able to accomplish this with ease. They would need to spend a lot of development time on implementing underlying functionality in JavaScript to support video calling. Further, they would be naive to the problems that can be run into when using the foreign function interface (FFI) to call out to JavaScript functions (more on this in Section 3.2.1 and Section 3.5).

On top of this live streaming feature, at present, many popular applications such as Zoom [28], Skype [26], and gather.town [19] have been built. The web application on which the app building component of this project is based, namely gather.town, aims to enhance virtual social interactions by implementing a visual representation of each user's position on their screen. When users get near each other, a video call begins between them automatically.

However, gather.town has shortfalls with respect to usability and accessibility [6]. For example, it can be awkward to start conversations between users, as well as joining a group conversation, due to the fact that each user's face pops up when they get near each other. A more simple issue is that it is not clear whether a user's camera feed is

visible to other users nearby (i.e., a user may see their own camera feed displayed on their screen, but there is no clear indication that it is also visible to others).

## 1.2 Project Goals

The main objective of this project is to design and implement a flexible API in Links to support live streaming, write documentation for it, then build a dynamic video calling application (i.e., a `gather.town` clone) on top of it. Specifically, the application shall provide a web page interface that allows users to move around their screen as a circle, using the arrow keys on their keyboard. If multiple users are to connect to the app on the same server, they shall see each other's circles in real time. Finally, if two or more users move within close proximity of each other, they shall see each other's live camera feeds. Each camera feed instance shall include buttons that allow the user to turn on/off video and/or audio. The dynamic video calling application shall be used to evaluate the live streaming API, too.

## 1.3 Summary of Contributions

I implemented the live streaming API with a flexible design that allows it to be used not just within the dynamic video calling application, but elsewhere, too. A web developer who wishes to use Links to build a video calling application, or a simpler program that allows users to voice chat, can do so with ease using this live streaming API. I also wrote documentation for the API, which is presented in Appendix A.

As part of the API, which is written in Links and JavaScript, I implemented a set of functions that the Links developer can use to:

- register the user to the server
- prepare the user's media devices - cameras and microphones
- have the user begin a peer-to-peer connection with another user with a given unique ID, and add the user's media live streams to the connection
- disconnect the user from another user with a given unique ID
- check if the user is connected to another user with a given unique ID

Using the functions in this API, I implemented the dynamic video chat application as described in the project goals section, with the exception of not implementing buttons to turn on/off the user's microphone and camera (justification in Section 6.1). Also, I used Links' Model-View-Update (MVU) framework [11] to program the graphical user interface (GUI) in order to update the user's web page regularly (more on MVU in Section 2.6).



## 1.4 Report Structure

In Chapter 2, I introduce the background and important concepts used throughout my project. In Chapters 3 and 4 I discuss the design and implementation of the live streaming API and the dynamic video calling application built on it. In Chapter 5, I evaluate both components, and in Chapter 6 I conclude and discuss potential future developments for the extension of my project next year.

# Chapter 2

## Background

In this chapter, I discuss the key concepts and software technology involved in this project. Such software includes the web-development-targeted functional programming language Links [4], as well as the well-known programming language JavaScript [22], and the WebRTC [2] peer-to-peer real time communication API. The main topics and concepts involved include web development, distributed actor-style concurrency, and communication between users over the internet.

Also, I comment on various web applications around today that are similar to that of `gather.town` and the dynamic video calling application built in this project.

### 2.1 Fundamentals of Web Development

Knowledge of the way in which web applications are designed is key to developing a well-functioning and aesthetically appealing one. I mention the term web application instead of the term web page because of how much more the former encompasses than the latter. Today, the design and implementation of web pages and web applications is known as front-end and full-stack web development, respectively.

Building the front-end of a web application involves writing code that manipulates the Document Object Model (DOM). The DOM can be thought of as a tree structure, wherein each node represents an HTML tag (or XML in Links). On its own, HTML [21] can be used to create a simple web page that does nothing but display some lines and text. Introducing colour and a little bit of dynamism (e.g. changing the colour of text upon hovering over it) is still possible in HTML itself, but much easier if a style sheet language, such as CSS [17], is used as well. Finally, if I wish to manipulate the web page more intricately, or require complex logic to allow a user to carry out a task, then it would be appropriate to make use of the programming language, JavaScript [22], too.

However, a good understanding of these three languages is not enough to implement a dynamic video calling application, which is the goal of this project. Knowledge of WebRTC [27] (discussed in Section 2.3) and how to create a web server to act as an intermediary between the clients is also needed.

The use of a web server is what distinguishes a web application from a web page. If a web user shares their web page with a few peers and asks them to open the web page on their browsers, and one of them makes a change to the page, the other users will not observe this change immediately. There is no way for this change to be communicated, other than to be explicitly sent through email, for example. If a web server is implemented, then any changes made would be sent to the server and relayed to the other clients.

Of course, there is more to web servers than just allowing clients to communicate; web servers may also have security protocols in place (e.g. TLS or SSL) and they will likely incorporate databases, too. However, this report will not explore these areas too deeply.

## 2.2 Web Development in Links

As described in the previous section, building a functional, appealing, and interactive web application requires knowledge of the three tiers of web development and mastering the following languages: HTML, CSS, and JavaScript to program the front-end; Java, PHP, or Python to implement the server-side logic (known as back-end); SQL to send queries to the database if one is used. For a beginner web developer who simply wants to build their own website, learning such a myriad of languages (i.e. the full-stack) is not desirable. There is also the difficulty of linking these tiers to ensure that messages sent from one tier to another are of a data type the receiving tier expects (e.g. to make sure that a form in HTML produces data of a type that the logic in the server expects). This difficulty of linking these tiers is known as the “impedance mismatch” problem [4].

With Links, the programmer needs to know only one language to develop an aesthetically pleasing and well-functioning web application. Therefore, the beginner web developer may be better off learning just one language, Links, instead. Also, Links eases the impedance mismatch problem by providing a single language to handle all three tiers. Links produces JavaScript for the browser, a bytecode for the server, and SQL for the database.

Throughout this project, I made use of multiple features of Links to implement both the live streaming API and the dynamic video calling application. These features include the distributed actor-style concurrency model and the Model-View-Update framework. However, before diving into these, it is important that I explain the way in which users connect to each other using the live streaming API.

## 2.3 Web APIs Used for Video Calling

In order to implement an API in Links to support live video streaming, I had to research into how media devices are accessed for use in web applications. Generally, the MediaDevices web API [1] is required to access the user’s microphone and webcam. More importantly, the technology used to share the data originating from these media devices is the WebRTC web API [2]. WebRTC is vital in enabling rapid transfer of live

media streams between users' browsers, without necessarily needing an intermediary server.

Before using WebRTC to implement a live streaming API in Links from scratch, it is important to understand deeply the process of setting up a WebRTC connection between users and having them exchange live video streams. To summarise how WebRTC is used to connect users and share their live media streams; firstly, there must exist a server that acts as an intermediary to allow the users to exchange connection and media details; next, the users' browsers must find the best way to access each other; lastly, they initiate a WebRTC peer connection between each other. It is through this connection they transfer their streams.

In the subsection below (Section 2.3.1), I elaborate on the details of how a WebRTC connection is executed.

## 2.3.1 The Connection Process

### 2.3.1.1 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/audio streams with one another. To begin the process, each user will connect to a "signalling server". In the case of my implementation, this is the server process that the clients use to communicate. This server will simply broadcast any messages sent to it to all users connected, including the sender. When user 1 connects, user 1 sends a message to the server that reaches no one else since they are the only one connected. When user 2 connects, user 2 sends a message to the server that is received by user 1 who can then create a `RTCPeerConnection` object (see Section 4.1.2) and begin collecting ICE candidates (more on ICE candidates in Section 2.3.1.2, below). Upon receiving this message, user 1 sends a reply to the server that is received by user 2 so that they can begin the same process too.

### 2.3.1.2 Identifying ICE Candidates

Before the users can connect to each other, they must identify the IP addresses through which they can transfer data. These may include the user's private IP address within a local network, which could be accessed by other devices in the same local network, or a public IP address, which could be accessed by the internet.

Each of 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 the data exchange between the users and how they use STUN servers to find their external IP addresses.

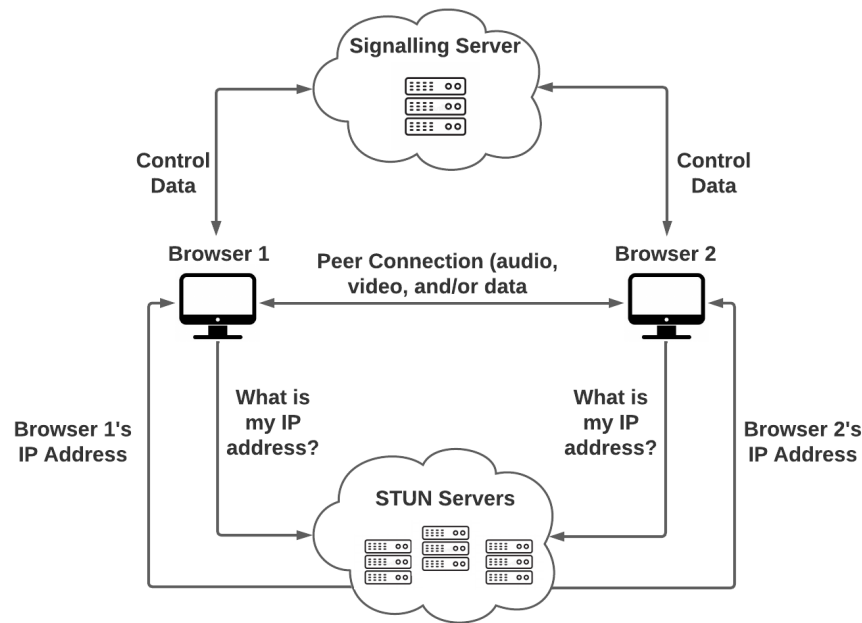


Figure 2.1: Communication with the signalling server and STUN servers.

### 2.3.1.3 Session Description Protocol (SDP)

Once each of the users have their own collection of ICE candidates, they should be ready to set up their connection. Firstly, user 1 creates an SDP, 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 will set this SDP as their “local description” and then “stringify” the SDP so that it can be sent to user 2 (through the signalling server). Once the SDP reaches user 2, user 2 sets it as their “remote description” and 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”.

### 2.3.1.4 Completing the Connection

Once the users have exchanged descriptions (SDPs) successfully they will know about each other’s details and, most importantly, the multiple ways they can connect to each other through the various ICE candidates of each. 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 accessible by the corresponding user who can then use this media. The details of implementation of the WebRTC connection process are described in Section 4.1.2.

## 2.4 Accessing Web APIs in Links

As mentioned in the previous chapter, the `MediaDevices` and `WebRTC` APIs are required to access the user’s camera and microphone and implement the WebRTC connection

process. These APIs can be accessed by JavaScript only and so there must be a way to run JavaScript code through Links. This is accomplished with Links' foreign function interface (FFI).

As an example, let there be two JavaScript functions that the programmer wishes to run via 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 that indicates 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 that they can be accessed by the Links program. The code below shows how the JavaScript file is organised.

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

function _checkIfCameraLoaded() {
    return ...
}

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

Once the JavaScript functions are completed and ready to be called via Links, code must be written in Links to locate and access these functions. In the Links file, executing these JavaScript functions requires Links to know where the JavaScript file 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.

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

Then, 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)
```

## 2.5 Distributed Actor-Style Concurrency Model

As discussed above in the section describing the connection process (Section 2.3.1), users who wish to connect to each other must send messages to a “signalling server” that broadcasts the message 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 whether to reply to it.

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 between clients and server varies. A web developer who is fluent in HTML, CSS, and JavaScript may use those to program the client, and a JavaScript runtime environment such as Node.js [24] to implement the server. This set of languages and environment alone would be sufficient to implement a video calling application using WebRTC [16].

In Links, due to the fact that a single Links program runs both the client and the server, the method used to implement communication between clients and the server differs from above. I came across a communication model that Links supports known as the distributed actor-style concurrency model (based on the actor model [7]).

Generally, 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 and use pattern matching to perform actions based on the message. It is distributed in the sense that not all actors need to be run locally (i.e. an actor can be run on a different machine and still receive messages, provided that the sender knows how to reach it).

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

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

## 2.6 Model-View-Update Framework in Links

Given the background described in this chapter so far, a web developer would be capable of implementing the main logic and back-end of a video calling application. However, nothing has been mentioned about what will be displayed on the user’s screen as they are using the application. As mentioned above (in Section 2.1), web developers typically use HTML, CSS, and JavaScript to build the front-end of a web application and manipulate the web page displayed by the user’s browser.

By using Links, the developer need only use Links to write code that manipulates the DOM and control what the user sees. Links provides several ways of interacting with the DOM. The simplest of these is through the use of DOM operations that can 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 a set of functions to manipulate the DOM.

The other approach, and the one used in this project, is to make use of Links' Model-View-Update (MVU) framework [11]. This method of updating the DOM is similar in some ways to that of the JavaScript library, React [25], and is specifically based on the front-end web programming language, Elm [18]. Briefly, a program using the MVU pattern has three main components: a model datatype that contains the program state, a view function that takes the model as input and renders it as HTML, and an update function that takes as input the model and a message and returns an updated model.

The MVU framework also has the 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, a switch statement 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 HTML to display.

## 2.7 Existing Systems

In this section, I review some existing web applications that build on the video calling functionality to produce interesting and useful features.

### 2.7.1 Gather

The main inspiration of this project originates within the most well-known application of its kind, "gather.town" [19]. In a nutshell, gather.town is a web conferencing web site that allows users to communicate over a video call, but with the added capability of being able to "walk around a room" virtually and video call with those close enough only. It has been recognised as a platform that makes "virtual parties fun" [13] and has hosted events that turned out to be a "masterclass" [10]. Despite this, I, and many others [6], believe there are still aspects of it that can be improved to make for a better experience for various types of users. These shortfalls are described in the project motivation section in this report (Section 1.1). **Figure 2.2** shows Gather in action as five people sit at a table and video call each other.

Referring to more formal work, a paper written by lecturers at Lancaster University [8] describes several experiments carried out on gather.town. In the experiment on using gather.town for teaching or research, they found that, when using the "Breakout Rooms" feature, "the feel of colocation is illusive". By this I believe they mean that gather.town's implementation of breakout rooms does not capture enough of the way in which in-person breakout rooms are carried out. For example, eye contact is limited, the way participants are assigned to rooms does not seem natural, the breakout rooms close abruptly, and it is difficult for teachers to move around groups with ease to check on their progress. They also mention that there is a lack of engagement such that attendees with their cameras off do not make effort to interact.





Figure 2.2: The dynamic video calling application gather.town in action [14].

## 2.7.2 Mozilla Hubs

Mozilla Hubs [23] is substantially complex with lots of useful features. Users can move around in a 3D world and talk to each other through their microphones. It has spatial audio, so users can talk to those only within close proximity (like Gather). Another interesting feature of Mozilla Hubs is its approach with video communication. When users want to video call each other, they can insert and position a frame in the world containing their live camera feed. This is demonstrated in **Figure 2.3**.

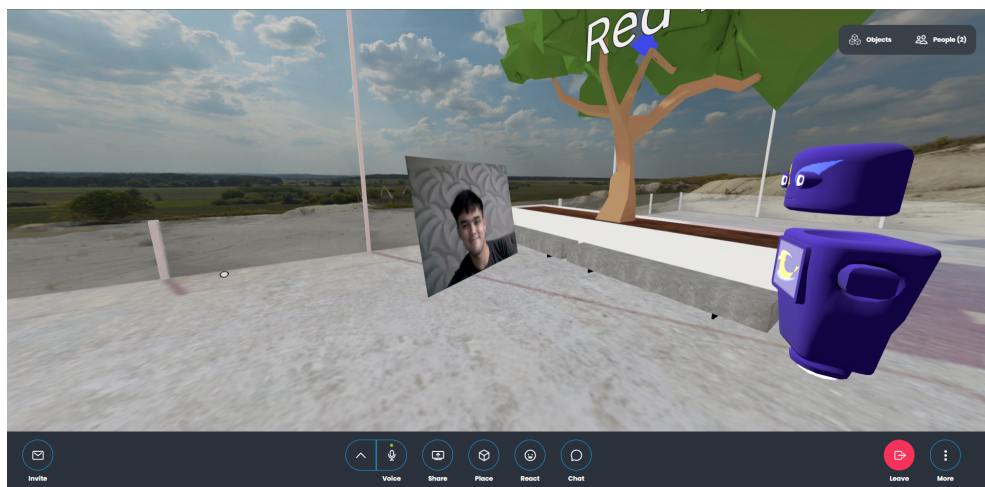


Figure 2.3: Users can insert and position a frame containing their live camera feed and move and rotate it in any way they like.

However, it may be too complex for users with little knowledge of technology because of the many actions that are based on pressing specific keys, and that they have to remember which keys do what.

### 2.7.3 GroupRoom

With GroupRoom [20], a user can create a room using an image as the background (typically an office floor plan) around which a group of users can move. Users can move by clicking parts of the room to move to certain positions or by using the arrow keys on their keyboard. A very novel feature is that the user's position in the room is represented by a circle with the user's camera feed inside it. The app also uses spatial audio in the sense that users can hear users close to them but not users far away. **Figure 2.4** shows a specific use case.



Figure 2.4: In this use case, an image of an office was used and users were positioned at their desktops.

## 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 programmer's web application.
- **Client.** When the programmer runs their web application, only the server side runs initially, since no users have joined yet. When a user opens their browser and uses the web application (by using the URL of the web application to find it), the client side of the web application runs on the browser. This "client side" is what client refers to.
- **Local client.** Unless specified otherwise, this refers to the client that the functions of the live streaming API are being called from.
- **Remote client.** This refers to another client connected to the same server as the local client.
- **UUID.** This refers to the "universally unique identifier" of a specific client. The UUIDs produced by my implementation are not strictly UUIDs [12], however.

- **Character.** This refers to the user's movable circle when using the dynamic video calling application.

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

## Design

In this chapter I present a high level description of the design of the live streaming API, as well the dynamic video calling application built on top of it.

### 3.1 Using the Live Streaming API

I designed the live streaming API as a set of functions that provide the programmer with great flexibility when they wish to feature video calling in their web application. With this set of functions, the programmer can decide specifically which remote clients the local client can begin a call with and when to disconnect from that remote client. Also, they can check whether the local client is connected to a specific remote client given their UUID.

The names and function signatures of the main Links functions in the API are listed below:

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

The empty tuple return type indicates that the function does not return anything.

As hinted by the names of the functions listed above, they can be split into two main categories: the functions that handle the user's media devices, and the functions that handle the peer-to-peer WebRTC connections between users.

### 3.1.1 Using the Media Devices Functions

If the programmer wishes to write code that retrieves the user's microphone and camera to create a HTML video element that displays a live stream of the user's audio and video input, they would use the functions `gatherDeviceIds`, `getDeviceIds`, `getDeviceLabels`, and `readyMediaDevices`.

The first of these, `gatherDeviceIds`, takes as input the type of media device (either video or audio) and searches for devices of this type that are connected to the user's computer. It then stores the IDs and labels of each device in separate lists. **Figure 3.1** depicts the scenario where video input devices are requested.

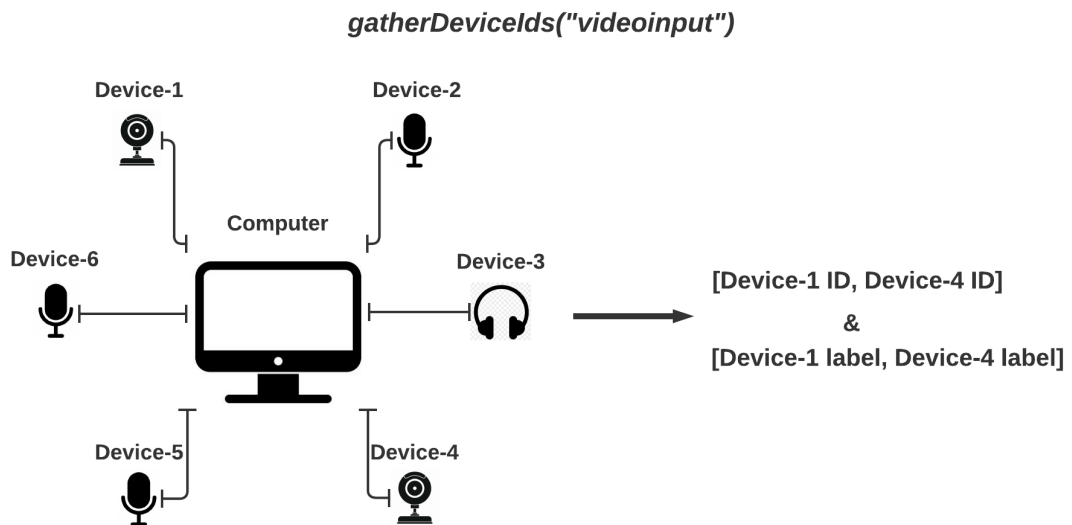


Figure 3.1: `gatherDeviceIds` is called to find all connected devices that take video input (e.g. cameras, webcams). In this case, it stores the IDs and labels of Device-1 and Device-4 as they take video input.

Once this function has been called (twice) and all connected video and audio input devices have been recorded, their IDs and labels can be retrieved. This is done by calling `getDeviceIds` and `getDeviceLabels`. It is then up to the programmer to design an interface that presents the user with a list of options to choose from. The programmer would be best to use the device labels to present the options to the user.

Once the user has chosen an option for both video and audio input, the function `readyMediaDevices` can be used to access the specified devices and build the HTML video element containing the devices' live video and audio streams. In **Figure 3.2**, Device-1 and Device-5 are chosen and used to build the HTML video element.

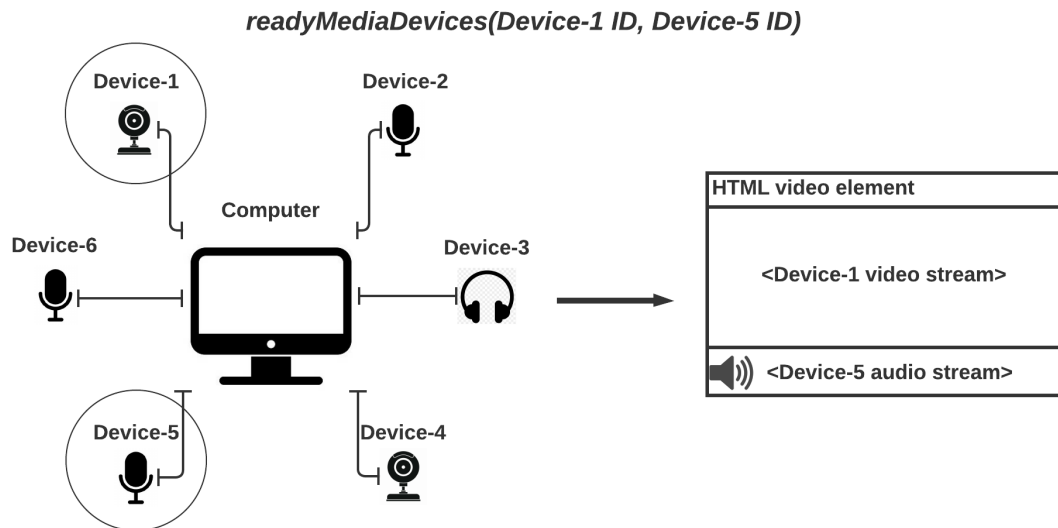


Figure 3.2: `readyMediaDevices` is called to retrieve the media devices specified by the input device IDs. In this case, it retrieves Device-1 (webcam) and Device-5 (microphone) and creates HTML video element outputting the devices' live video and audio streams.

### 3.1.2 Using the WebRTC Connection Functions

Before attempting to connect to a remote client and initiate a video call with them, the local client must have used the above functions already in order to have a live media stream to share. Assuming this, the client can begin to connect to a remote client (assuming there is another client connected to the same server).

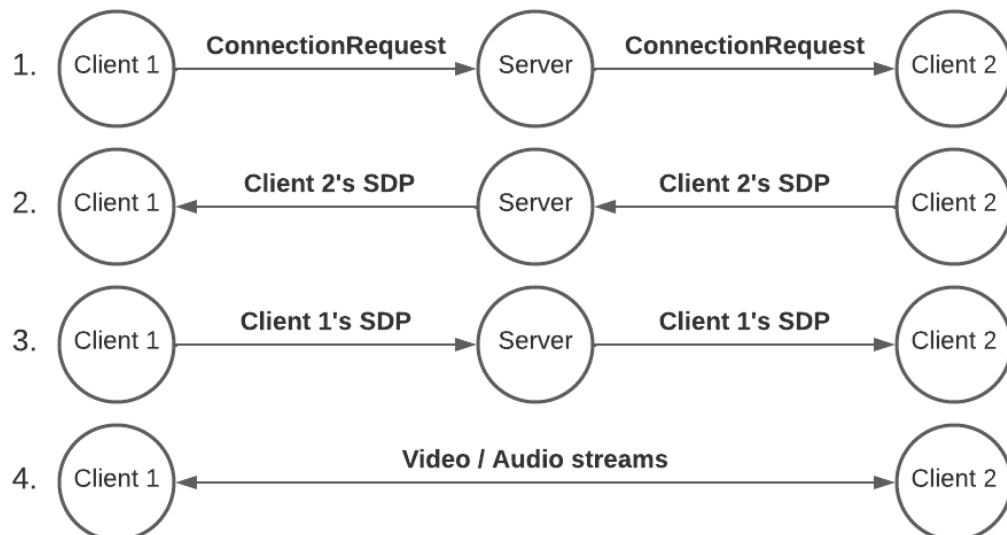


Figure 3.3: The sequence of messages displayed is a result of Client 1 calling `connectToUser` with the UUID of Client 2 as the argument.

When the local client is ready to begin or respond to connection requests, it must call `registerUser` to register itself with the server. Once registered with the server, it can call `connectToUser` to begin a WebRTC connection with a remote client, given their UUID. Calling this function results in a sequence of messages exchanged between the clients (via the server), ending up as either a successful or unsuccessful connection. In a successful connection, the clients' live video and audio streams can begin to be shared between one another. The sequence of messages exchanged is summarised in **Figure 3.3**.

## 3.2 How the Live Streaming API Works

Having walked through the use of the live streaming API, now I describe the underlying design of it.

As mentioned in Section 2.4, accessing the user's media devices requires the use of the `MediaDevices` and `WebRTC` APIs that can be used only through JavaScript. Therefore, it was essential I made use of Links' foreign function interface to call functions in a JavaScript file that accesses these APIs.

### 3.2.1 How Media Devices are Accessed

The function `gatherMediaDevices` calls a function in JavaScript that enumerates the media devices (of the input type) connected to the user's computer. However, another hidden detail is that this function waits until all connected devices of the requested type have been found and their IDs and labels collected. I introduced a Links function `waitForMediaDeviceIds` that repeatedly checks if all IDs and labels of the requested devices have been collected. **Figure 3.4** depicts this waiting functionality.

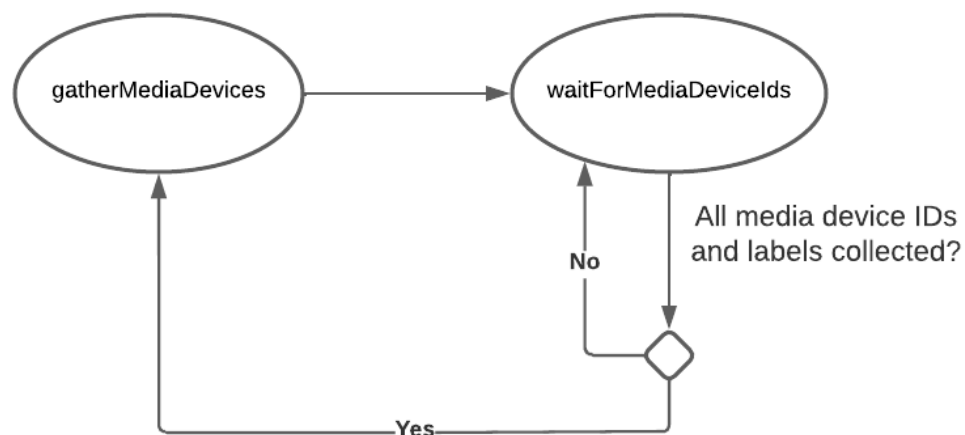


Figure 3.4: Accessing the user's media devices is an asynchronous operation, so it is essential to wait until this finishes before continuing.

Using this wait function is necessary because a JavaScript function that uses the `MediaDevices` API to access the user's media devices makes use of asynchronous JavaScript promises and the `Links` program that calls this function (e.g. `gatherMediaDevices`) does not wait for this promise to return. Since the `MediaDevices` API function that enumerates the connected media devices uses a JavaScript promise, using `getDeviceIds` immediately after will likely function incorrectly and return an incomplete list of device IDs.

After the device IDs and labels have been collected and the user has chosen a specific video and audio device to use, `readyMediaDevices` is used to access the chosen devices' streams and insert these into an HTML video element. Again, accessing these devices involves calling a JavaScript function that makes use of a JavaScript promise, so this function uses a function similar to that of `waitForMediaDeviceIds` to wait until the promise returns, and the media devices' streams have been retrieved.

### 3.2.2 How WebRTC Connections are Completed

When `connectToUser` is called, the local client sends a message to the remote client (via the server) requesting to connect. Once the remote client receives this, it creates its own Session Description Protocol object (SDP, described in Section 2.3.1.3) using a JavaScript function, then sends it to the local client that does the exact same and sends its SDP back. This whole exchange of messages is shown in **Figure 3.3**.

However, at the stage where each client creates its own SDP, a similar situation to that of accessing the media devices occurs. In the JavaScript function that creates the SDP, a JavaScript promise is used. Therefore, I implemented a wait function similar to that of `waitForMediaDeviceIds` that waits until the promise finishes. Using this wait function prevents the client from sending an SDP that does not actually exist yet.

The live streaming API uses `Links`' distributed actor-style concurrency model (Section 2.5) to allow the clients to send messages to the server and vice versa. Referring to **Figure 3.3**, the actors in this use case are the entities (i.e. Client 1, Client 2, and the server).

## 3.3 Using the Dynamic Video Calling Application

The dynamic video calling web application was built on top of the live streaming API. Given that the application has been executed on a server, users can enter the server URL to begin using it.

Once a user has entered the server URL and connects to the server, they will be presented with an interface that asks them to enter their name. **Figure 3.5** depicts this interface.

After the user has entered a name and pressed enter, they will be presented with a different interface that asks them to choose a specific camera and microphone to use for the video call. This interface is displayed in **Figure 3.6**.

Finally, the user will see a rectangle in the middle of the page displaying the video stream coming from the video device they chose. Using this live video stream as a



guide, the user can position their face in the middle and click “Use” to use their face as their character’s icon. **Figure 3.7** depicts this interface.

A gray rectangular interface with the text "Enter your name:" in a large, bold, black font. Below the text is a white rectangular input field with a black border.

Figure 3.5: Interface 1. The user must enter a name before proceeding.

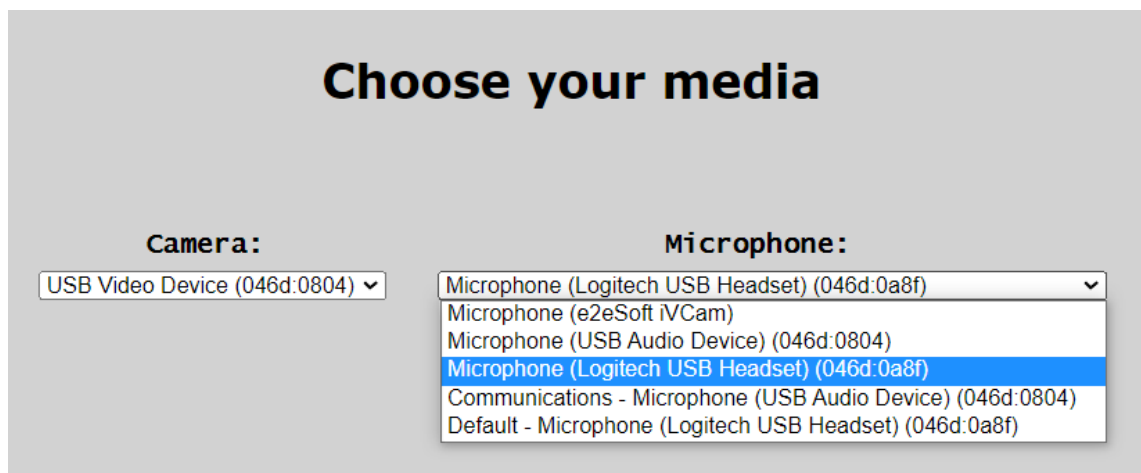
A gray rectangular interface with the title "Choose your media" in a large, bold, black font. Below the title are two dropdown menus. The first is labeled "Camera:" and shows "USB Video Device (046d:0804)" with a downward arrow. The second is labeled "Microphone:" and shows a list of devices: "Microphone (Logitech USB Headset) (046d:0a8f)", "Microphone (e2eSoft iVCam)", "Microphone (USB Audio Device) (046d:0804)", "Microphone (Logitech USB Headset) (046d:0a8f)" (highlighted in blue), "Communications - Microphone (USB Audio Device) (046d:0804)", and "Default - Microphone (Logitech USB Headset) (046d:0a8f)".

Figure 3.6: Interface 2. The user is then asked to choose their camera and microphone.

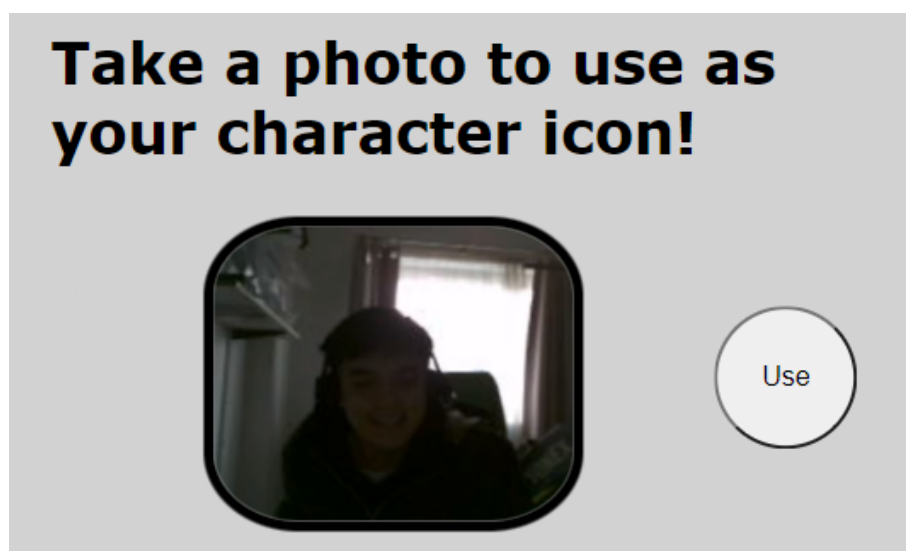
A gray rectangular interface with the text "Take a photo to use as your character icon!" in a large, bold, black font. Below the text is a rounded rectangular video window showing a live video stream of a person's face. To the right of the video window is a circular button with the text "Use" inside.

Figure 3.7: 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.

Once these three steps have been completed the user will see their character containing the picture they took in the previous interface, as well as their own live camera feed in the bottom right corner. The user must use the arrow keys on their keyboard to move their character around the page. **Figure 3.8** shows this.

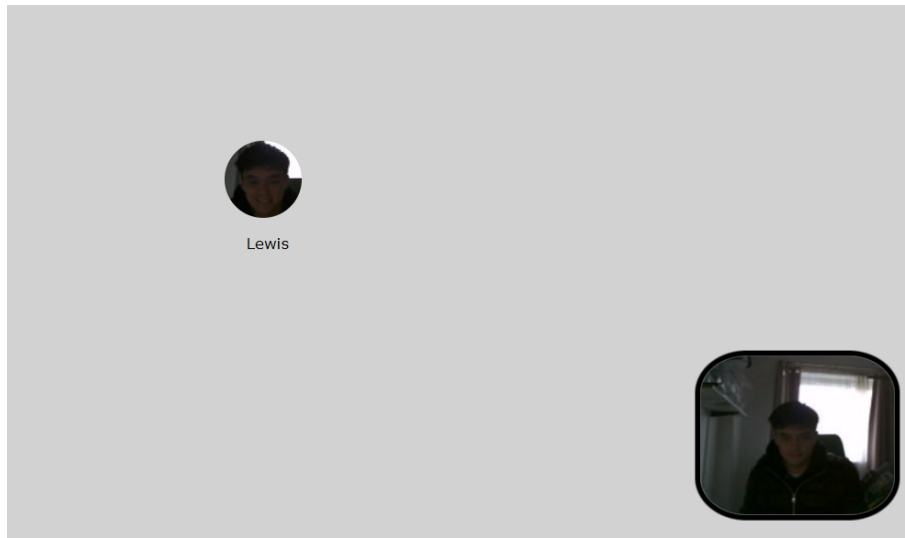


Figure 3.8: Interface 4. The picture the user took in the previous page is used as their character icon, and their own live video stream is displayed in the corner.

So far, the web application is not very interesting as there is only a single user connected. When a remote user connects, the local user will see the remote user's character on the page, and begin a video call with them when they move within close proximity. The remote user's camera feed will appear in the top left, alongside any other remote users who are in proximity too. This is shown in **Figure 3.9**.

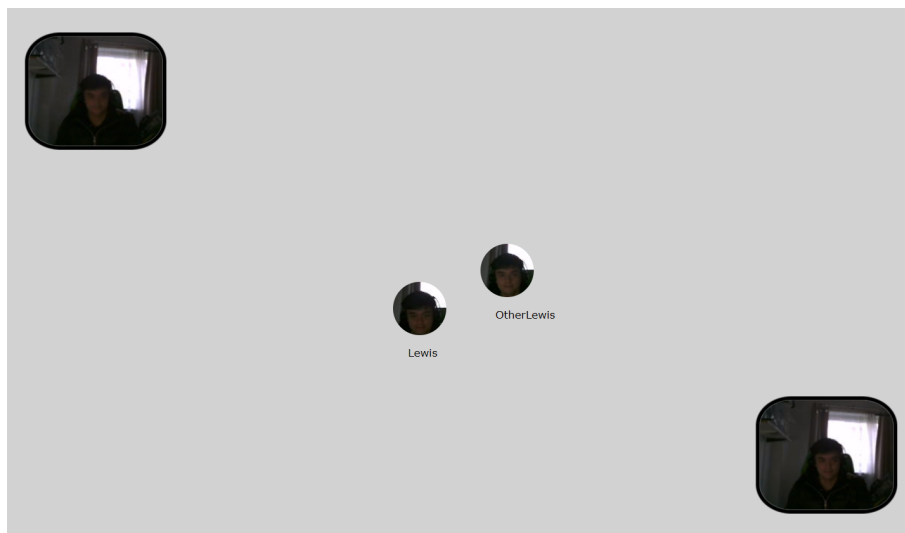


Figure 3.9: When another user connects, the local user can see their character and its position on the page. A video call begins when the users move within close proximity of each other.

## 3.4 Design of the Dynamic Video Calling Application

### 3.4.1 User Interface

I implemented the user interface by using Links' Model-View-Update (MVU) framework. As discussed in Section 2.5, this framework consists of three main components: the model datatype representing the state of the client, the view function that transforms the model into a HTML user interface, and the update function that runs upon the triggering of an event. In this case, the model composes of the client's character data (i.e. UUID, position on page, icon image URL), the character's moving directions (horizontal and vertical), the current main interface (e.g. the enter name interface), and a list of all remote clients' character data; the view function can display a variety of user interfaces (depicted in **Figures 3.5-3.8**) depending on the state of the model; the update function listens for events such as arrow presses on the user's keyboard to update the character's position on the page (by updating the model).

**CharData:** (uuid=123, name=Lewis, x=20, y=47, imageurl=IMAGE-URL)

**Model:** (chardata=CharData, dirVer=Up, dirHor=Left, remotechars=[Character2, Character3])

Figure 3.10: The important attributes of the model (more are specified later). In this instance, the client has UUID as 123, name as Lewis, position as (20, 47), current moving directions as Up and Left, and Character2 and Character3 as remote client characters.

The application also uses MVU's subscriptions feature, which detects when a certain event occurs and calls the update function with the specific event type as input. Section 4.2.1 goes into more detail about these events. The different possible events that can be detected are listed below:

- **New frame.** Triggers every time a new frame renders.
- **Key down.** Triggers when a keyboard arrow key is pressed down to move the character.
- **Key up.** Triggers when a keyboard arrow key is released to stop moving the character.
- **Keyboard input.** Triggers when a keyboard key is pressed while focused on the "enter name" text box.
- **Enter.** Triggers when the keyboard's enter key is pressed to submit the entered name.
- **Click.** Triggers when the "Confirm" or "Use" buttons are clicked.

The top three are used for moving the client's character. Every frame, the update function is called with the new frame event as input. Simply, this inspects the two direction values of the model and moves the character in the respective directions by a

fixed distance (specified as a constant in the program). For the key down and key up events, the update function changes the direction values in the model based on which arrow keys were pressed (e.g. key down on the right arrow changes the horizontal direction value to Right).

The bottom three are used for taking input from the user when they are providing information to the application (i.e. in the interfaces in **Figures 3.5-3.7**). The keyboard input event is linked to the “enter name” box in the first interface and triggers for every character that is typed into it. Then, the enter event saves the typed characters into the name attribute of the model. Finally, the click event is linked to both the “choosing media devices” and “taking picture for character icon” interfaces to detect when the user clicks the “Confirm” or “Use” buttons.

### 3.4.2 Communicating Data Between Clients

Identical to the message exchanging design of the live streaming API, the communication model used is Links’ distributed actor-style concurrency model. Once the application is executed, the server process is created and awaits messages from clients when they connect. When a user connects, their client uses the media devices side of the live streaming API to prepare their live video and audio input, then registers them with the server.

However, when at least two clients are connected to the server, they can only begin exchanging data (i.e. character position, icon image URL) when they have chosen a name, their media devices, and captured a picture for their icon. When both clients have entered interface 4 and can move their characters, they can send and receive their characters’ positions and icon image URLs. This is depicted in **Figure 3.11**.

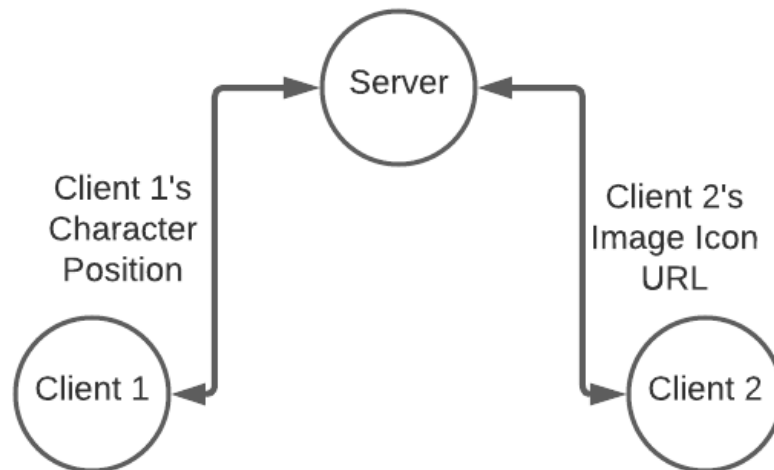


Figure 3.11: Client 1 sends Client 2 its character’s updated position. Client 2 sends Client 1 its icon image URL.

This part of the design of the application has been altered a few times throughout its implementation. In the first place, there was nothing designed to combine Links' Model-View-Update framework and distributed actor-style concurrency model. In other words, there was no standard method of passing received messages from remote clients to the update function. Given this, a few Links developers demonstrated assisted me in finding a way to accomplish this task.

In this method, the client process stores messages sent from remote clients. At the same time, the MVU component creates a process that sends requests to the client process and waits for a response containing the remote client's messages. Then, when the new MVU process receives a message response, it triggers the update function with the message as input. This way, the model gets updated as new messages arrive from remote clients. **Figure 3.12** shows this.

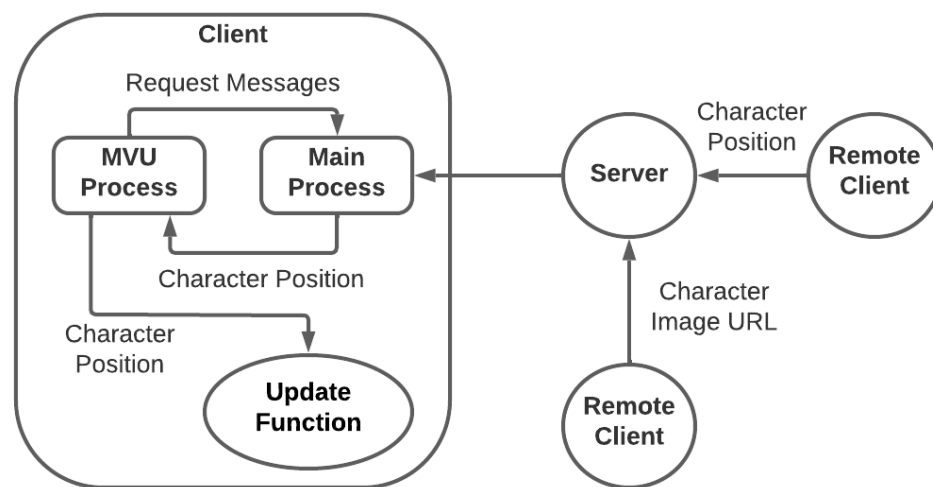


Figure 3.12: In the example displayed, remote clients send their character data to the local client, which are stored in the main process. The MVU process of the local client sends a request to the main process to see if any messages have been received. The main process responds with any messages it has received. The update function is called with this message as input.

At the beginning of February 2022, a new patch for MVU was released [5] that introduced a new function called “dispatch”. This function takes a message and dispatches it directly into the update function. Using this, there is no need for the client to store messages and have an extra process created.

Lastly, in the early stages of the design of the communication method between clients, I implemented only one type of message that could be sent by the client as they moved their character. This message contained both their character's position and icon image URL. Since the image URL is a very large string, frequently sending it slowed down the application significantly. For this reason, I introduced a new type of message that contains the image URL only. This message is sent just once when a client enters the main interface (interface 4), so that remote clients can store it permanently.

### 3.4.3 Incorporating the Live Streaming API

In my design of the user interfaces, I make use of the media devices side of the live streaming API. Initially, its used immediately before the user is presented with options to choose a camera and microphone. Here, `gatherDeviceIds` is used to collect all connected devices' IDs and labels that are presented as options to the user. In the following interface, `readyMediaDevices` is used with the chosen devices' IDs as input and the HTML video element containing the streams of these devices is displayed in the centre of the page as a guide for the user to take a picture for their icon image. Finally, when the user enters the main interface, the same HTML video element is used to display the user's camera feed in the bottom right corner.

The WebRTC connection side of the API is not used until the client moves within close proximity of a remote client or vice versa. Every time the character moves position, or the client receives the new position of a remote client, the distance between the characters is calculated using their `x` and `y` values. If the distance is less than or equal to 150 pixels, the client that moved to make this condition true will use `connectToUser` to begin a WebRTC connection with the other client within proximity.

Once the WebRTC connection completes and each client have live video and audio streams to share, the local client will receive a live stream of the remote client's chosen media devices and create a HTML video element containing this, which is displayed in the top left corner of the page (**Figure 3.9**). The same occurs for the remote client.

When the client detects that it is further than 150 pixels from the remote client, `disconnectFromUser` is called to close the WebRTC connection and remove the HTML video element containing the remote client's live stream.

## 3.5 Design Challenges and Justification

In the underlying design of the live streaming API, I make frequent use of extra “wait functions” that force the Links program to wait until a certain condition is met such that a JavaScript promise has returned. Although this may seem sub-optimal, the Links' foreign function interface has no standard method of dealing with JavaScript promises, so it is essential I explicitly force the Links program to wait. Since this is used quite often throughout my implementation, it may have been a good idea to generalise this functionality by implementing a function called “await” that takes an abstract promise type and returns when the promise finishes.

Also, when the client enters the main interface in the web application, I mentioned that the user's video and audio live streams are inserted into a HTML video element that is displayed to the user. What I did not specify is that this HTML video element is inserted into the DOM explicitly by using Links and JavaScript DOM operations, instead of using the MVU view function. The reason for this is because MVU's own method of manipulating the DOM does not support the attaching of live stream objects to HTML video elements. The `srcObject` property of the HTML video element needs to be assigned the live media streams of the user's devices in order for it to be displayed. Given this, the Links developers plan to resolve this by introducing “abstract types for

alien objects” [15] such as live stream objects in JavaScript.

# Chapter 4

## Implementation

This chapter covers the main implementation details of the live streaming API as well as the dynamic video calling application built on top of it.

### 4.1 Live Streaming API

The Links and JavaScript code that implement the live streaming API are written in the files `webRTC.links` and `jsWebRTC.js`.

#### 4.1.1 Accessing Media Devices

As shown in Section 3.1.1, multiple Links functions are implemented to handle the media devices side of the API. These functions combined with the wait functions abstract away the calls made to JavaScript functions to access the user's media devices.

When `gatherDeviceIds` is called, it calls the JavaScript function `_gatherMediaDeviceIds`, which then uses the standard `MediaDevices` API to enumerate the media devices connected to the user's computer. The specific `MediaDevices` method that does this is `enumerateDevices`, which returns a JavaScript promise containing a list of the found devices. This list of devices is looped through to collect the IDs and labels of each.

```
navigator.mediaDevices.enumerateDevices().then(  
  function(devices) {  
    devices.forEach(function(device) ...
```

The wait function in Links repeatedly calls `checkDevicesGathered` to check whether the device IDs and labels have been successfully collected. Once this returns true, `gatherDeviceIds` finishes and the getter functions can be used to retrieve the IDs and labels. The implementation of this specific wait function is shown below.

```
fun waitForDeviceIds(mediatype) {  
  var gathered = JSWebRTC.checkDevicesGathered(mediatype);  
  if (not(gathered))
```





In the web application I implemented, this video element is muted so the user does not have to hear themselves when they speak.

### 4.1.2 WebRTC Connection Details

As specified in Section 3.2.2, the implementation of setting up WebRTC connections between clients makes use of Links' distributed actor-style concurrency. Here, the actors are the client processes and the server process.

When the Links live streaming API is executed, a server process is created that waits for messages to arrive from client processes.

```
fun serverLoop(clients) server {
  receive {
    case Register(pid) ->
      var newClients = pid :: clients;
      serverLoop(newClients)

    case Broadcast(msg) ->
      broadcast(clients, Message(msg));
      serverLoop(clients)
  }
}
```

Client processes register with the server process by sending a `Register(pid)` message where `pid` is the process ID of the client process. The `clients` parameter is a list of client processes that have registered with the server. Client processes can also broadcast a message to all other remote client processes that are registered with the server process.

At the top of **webRTC.links** I declare a variety of datatypes that are used throughout the rest of the file. Two important datatypes are shown below.

```
typename MessageType = [| ConnectionRequest | SDP | Ice |];
typename PCMessage = (uuid : Uuid, dest : String,
  type : MessageType, sdp : String,
  sdptype : OfferOrAnswer,
  iceCandidates : Candidates);
```

These two datatypes are used frequently throughout the communication of messages to set up the WebRTC connection.

At the top of **jsWebRTC.js**, I initialise an empty global dictionary, `peerConnections`, which keeps track of the WebRTC connections between the client and remote clients.

Firstly, the client calls `registerUser` to register a new client process with the server process. This client process wraps its client process ID in a `Register` variant type and sends it to the server process. It then waits for a message to arrive from remote client processes via the server.

```
fun clientLoop() {
```

```

    receive {
        case Message(msg) ->
            handleMessage(msg);
            clientLoop()
    }
}

```

Once the client has a client process registered with the server process, the client can call `connectToUser` to attempt to begin a WebRTC connection with a remote client. The messages exchanged by the clients will have type `PCMessage`, which is a `Links` record. The most important attribute of this record is `type`, which indicates which type of message is being sent. The three different types of messages are listed below:

- **ConnectionRequest.** This is the first message sent by the initiating client and indicates to the remote client that they wish to begin a WebRTC connection.
- **SDP.** This indicates that a SDP object is included in the message.
- **Ice.** This indicates that the message contains a list of ICE candidate objects.

Back in Section 3.1.2, **Figure 3.3** displays the types of the messages exchanged between the clients. `connectToUser` takes the input UUID of the remote client to connect to and uses the server to broadcast a message of type `ConnectionRequest` to reach that remote client. In the message, `uuid` is set to the local client's UUID, `dest` is set to the remote client's UUID, `type` is set to `ConnectionRequest`, and the rest of the fields are left blank. The message is wrapped in a `Broadcast` variant type to be sent to the server process.

```

var message = (uuid=localUuid, dest=peerUuid, type=ConnectionRequest,
               sdp="_", sdptype="_", iceCandidates="_");
serverPid ! Broadcast(message);

```

However, before sending this message it calls the JavaScript function `_setUpPC`, which uses the standard WebRTC API to create an `RTCPeerConnection` object, which is the basis of the WebRTC connection between the clients. This function creates a new entry in the `peerConnections` dictionary with the remote client's UUID as the key and a nested dictionary with multiple entries as the value. The most important entry in this nested dictionary is the `RTCPeerConnection` object just created. `_setUpPC` also adds the local client's video and audio streams to the `RTCPeerConnection` object and initialises event handlers on it to detect when the remote client adds their video and audio streams. These streams will not be detected until the connection is complete.

The remote client's process, which is waiting in `clientLoop` for messages to arrive, will receive this `ConnectionRequest` message. With the message as an argument, it calls `handleMessage`, which uses pattern matching on the message `type` to decide which block of code to run.

Given `ConnectionRequest`, the remote client does essentially the same as the local client does in `connectToUser` by calling `_setUpPC` and creating its own `RTCPeerConnection` object. However, the remote client also creates its own SDP

object and sets this as its local description using the `RTCPeerConnection` object's methods `createOffer` and `setLocalDescription`, respectively.

```
peerConnections[peerUuid].pc.createOffer().then(
function(description) {
    peerConnections[peerUuid].pc.setLocalDescription(description)
    .then(function() {
        ...
    })
})
```

These two methods return JavaScript promises and so it is essential that a wait function is used in the Links program to wait until the promises finish.

Once the promises have returned, the program execution will exit the wait function and the remote client will be able to retrieve its SDP object and send it to the local client. The client's SDP object can be accessed simply through the `RTCPeerConnection` object's attribute `localDescription`.

The remote client creates a new message containing the stringified SDP object and wraps it in a Broadcast variant type to send to the local client via the server. It also has the field `sdpType` set, which is "offer" in this case.

```
var message = {uuid:localUuid, dest:peerUuid, type:SDP,
               sdp:desc, sdptype:sdpType, iceCandidates="_"};
serverPid ! Broadcast(message);
```

When the local client receives this message, it uses its `RTCPeerConnection` object's method `setRemoteDescription` to set the remote client's SDP as its remote description. Then, it does exactly the same as the remote client and creates and sets its own local description and sends this to the remote client.

Once the remote client receives the local client's SDP, it sets the SDP as its remote description. At this point, the event handler of the local client's `RTCPeerConnection` object should detect the remote client's video and audio streams and vice versa.

The last part to discuss is the retrieval and sending of ICE candidates. Back when the client calls `_setUpPC` there is also an event handler initialised that triggers upon finding ICE candidates and adds them to a list in the nested dictionary. The client process also creates a new process in Links for the sole purpose of collecting ICE candidates. This new process calls `handleIceCandidates`, which repeatedly calls the JavaScript function `_collectCandidates`. This function loops through the `peerConnections` dictionary and checks the ICE candidate list entry of each remote client. It stores the ICE candidates found in a new dictionary and returns this. If any candidates are found, they are put into a message of type `Ice` and wrapped into the Broadcast variant type to be broadcasted by the server. If no ICE candidates have been found, the function returns "No candidates".

```
var message = {uuid:localUuid, dest="all", type=Ice,
               sdp="_", sdptype="_", iceCandidates=candidates};
serverPid ! Broadcast(message);
```

When the remote client receives this message, it runs `handleMessage` with case `Ice`, which calls the JavaScript function `_addCandidates` to add the ICE candidates to its `RTCPeerConnection` object.

## 4.2 The Dynamic Video Calling Application

The implementation of the dynamic video calling application is in **app.links**, **app.js**, and **app.css**.

### 4.2.1 Model, View, and Update

The model in my implementation is the `Room` datatype displayed below.

```
typename CharacterState = [| Up | Down | Left | Right | Still |];
typename CharacterData = (id: Uuid, name: String, x : Float,
                          y: Float, imageURL : String);
typename Room = (charData : CharacterData,
                 directionV : CharacterState,
                 directionH : CharacterState, state : String,
                 nameField : String, cameraId : DeviceID,
                 micId : DeviceID, others : [CharacterData]);
```

The `Room` datatype consists of `CharacterDatas`, `CharacterStates`, and `Strings`. In Section 3.4.1, **Figure 3.10** depicts the `Room` datatype at a more abstract level, but captures the most important details.

The `charData` field holds the data about the local client's character that includes its UUID, name, position on the page, and the image URL used as the character's icon. `directionV` and `directionH` represent which direction the character is moving in. The `state` field indicates which interface the application is currently presenting to the user. The three other `String` fields are used when taking the user's input throughout the first three interfaces. Finally, `others` is a list of each remote client's character data.

Also declared at the top of the file is the `Msg` datatype. This datatype represents which messages (variant tags) can be passed to the update function to update the model.

```
typename Msg = [| NewFrame | MoveCharV : CharacterState
                 | MoveCharH : CharacterState
                 | UpdateNameField : String | NameEntered
                 | MediaChosen : (DeviceID, DeviceID) | Joined
                 | NoOp | ServerMsg : ServerMessage |];
```

The update function `updt` takes one of these variant tags and the current state of `Room` as input and returns an updated state of `Room`. The types of user input that trigger these messages to be passed into `updt` are mentioned in Section 3.4.1. The following bullet points describe how each message is triggered and what the update function does when it receives it.

- **NewFrame.** Triggered every new frame. Checks `directionV` and `directionH` of model and calls `moveChar` on both to move the character's position (`x` and `y`) based on the direction.
- **MoveCharV.** Triggered when the user presses either the up or down arrows on their keyboard. Changes `directionV` to either Up or Down and calls `moveChar` to move the character's position based on the current direction. Then, it checks whether the character is within proximity of any of the characters in the `others` list. It also broadcasts the character's position to all remote clients via the server.
- **MoveCharH.** Same as `MoveCharV` but changes `directionH` to Left or Right.
- **UpdateNameField.** Triggered whenever a character is typed into the "enterName" box of the first interface. Updates the model's `nameField` to whatever has currently been typed into the box.
- **NameEntered.** Triggered when the user presses the enter key to confirm their name. Changes the model's state to "enterMedia" so that the view function changes the displayed interface to interface 2 (choose devices interface).
- **MediaChosen.** Triggered when the user clicks the "Confirm" button once they have chosen their media devices. The IDs of the chosen camera and microphone are wrapped inside this variant tag. It also calls the live streaming API functions `readyMediaDevices` and `registerUser`, and sets the model's `cameraId` and `micId` fields to the IDs passed in. Lastly, it sets the model's state to "enterImage".
- **Joined.** Triggers when the user clicks the "Use" button to take the picture in interface 3. It calls the JavaScript function `_takePicture` to capture the current frame of the live video stream displayed to the user and uses this as the user's character icon. It also broadcasts the image URL to all remote clients via the server.
- **NoOp.** Does nothing.
- **ServerMsg.** Triggers when a message is received from a remote client via the server, which calls the MVU dispatch function to pass the message to the update function. It checks whether the remote client's character is within proximity and broadcasts the character's position to all remote clients.

There is no mention of `gatherDeviceIds` being called. This is because it is called as the client registers with the server. The function `moveChar` takes the current character data and direction and returns an updated character data with a new position.

The view function takes the current state of `Room` and outputs the new HTML to display. In my implementation, the view function calls one of four functions depending on the `state` property of `Room`, which maps to one of the four interfaces in **Figures 3.5-3.8** in Section 3.3.

The model starts with `state` "enterName" and so the view function outputs HTML that renders as interface 1. "enterMedia" and "enterImage" map to interfaces 2 and 3, respectively. "joined" maps to the main interface, interface 4.

The most interesting interface is interface 2. When the state is “enterMedia”, the view function calls `enterMediaView`, which outputs HTML listing two sets of options for the user to choose from. At the top of this function, the API’s functions `getDeviceIds` and `getDeviceLabels` are called twice to retrieve the IDs and labels of every connected video and audio device. A snippet of `enterMediaView` is displayed below.

```
div (id ("canvas"),
    div (id ("joinBox"),
        ch([
            h1 (id ("welcomeText"), textNode("Choose your media")),
            select_(id ("selectCamera"),
                ch(listOfDeviceOptions((camIds, camLabels), "camera"))
            ),
            select_(id ("selectMic"),
                ch(listOfDeviceOptions((micIds, micLabels), "mic"))
            ),
            ...
        ])
```

The keywords `div` and `h1` correspond to actual HTML elements of those tags. In the example above, the `ch` keyword concatenates the HTML elements inside the list together. The `h1` node displays the text “Choose your media”, and the two `select_` nodes display `select` elements listing the options for both the video and audio devices. The other view functions work in a similar manner.

The view function and its auxiliary functions (e.g. `enterMediaView`) make up the view component of the MVU framework. This is the component that renders the model into HTML, which displays the graphical user interface to the user.

## 4.2.2 Multiple Clients

As mentioned in the design section, the web application also uses the distributed actor-style concurrency model to communicate data between clients. In **app.links**, a server process is created and waits for messages to be received (in the form of variant tags). It can either receive a `Register`, `BroadcastPosition`, or `BroadcastIcon` message. The former simply registers the client process with the server process. The other two broadcast either the character’s position or icon image URL.

The datatypes of the messages that can be sent from the client process are shown below.

```
typename PositionInfo = (id: String, x : Float, y : Float);
typename IconInfo = (id: String, name : String, imageURL : String);
typename ServerMessage = [| CharacterPosition : PositionInfo
                          | CharacterIcon : IconInfo |]
```

As described in Section 3.4.2, separating the position and image of the user’s character means that the large image URL need only be sent once. The `PositionInfo` and `IconInfo` records are wrapped in their `ServerMessage` variant tags and wrapped again in either `BroadcastPosition` or `BroadcastIcon` to be sent to the server.

When a remote client receives one of these messages, they will receive either a **Position** or **Icon** message.

```
fun clientLoop(msgs, pids, hndl) {
  receive {
    case Position(msg) ->
      Mvu.dispatch(ServerMsg(msg), hndl);
      clientLoop(msgs, pids, hndl)
    case Icon(msg) ->
      Mvu.dispatch(ServerMsg(msg), hndl);
      clientLoop(msgs, pids, hndl)
  }
}
```

The MVU dispatch function is called to dispatch a `ServerMsg` containing the `CharacterPosition` or `CharacterIcon` into the update function. If the remote client calculates that the sending client is within proximity (150 pixels) of it, it calls `connectToUser` with the sending client's UUID and begins a WebRTC connection with it.

Once they successfully connect, the local client calls the JavaScript function `displayLiveStream` to display the remote client's live video stream in the top left corner and vice versa so they can begin the video call.



# Chapter 5

## Evaluation

In this chapter, I look over both the live streaming API and dynamic video calling application and identify strengths as well as what could be improved.

### 5.1 Live Streaming API

Given that I was able to successfully build a video calling application on top of the live streaming API indicates that the API is at the least, usable. Despite this, it is still sub-optimal in some areas and fails to function correctly occasionally.

#### 5.1.1 Error Checking

The live streaming API seems to work fine in the “sunny day scenario” where the application built on it is up to speed and each of the functions execute successfully. However, in the case where the application is slow or the messages take a while to reach the receiving client, the API can run into timing issues.

For example, when I put the video calling application under great stress by connecting many clients to the server, it becomes very slow and the character seems to move long after the arrow keys are pressed. Then, when I move a client’s character into the proximity of another and move away immediately after, the error message in **Figure 5.1** pops up in the console.

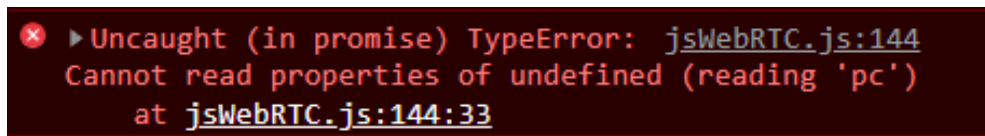


Figure 5.1: A timing issue causes the live streaming API to function incorrectly.

This error occurs because `disconnectFromUser` is called immediately after `connectToUser` and the whole WebRTC connection process does not complete quick enough due to the application being very slow. `disconnectFromUser` simply deletes

the remote client's entry in the `peerConnections` dictionary, so when one of the WebRTC connection functions tries to access the entry in the dictionary, it resolves to "undefined".

A way of fixing this would be to include an "if statement" whenever one of the WebRTC connection functions tries to access an entry in the `peerConnections` dictionary and cancel the whole process if it returns false.

### 5.1.2 Broadcasting Data

When the local client sends a message to the server to reach a specific remote client, the server broadcasts this message to all clients, including the sender. Since the local client only needs the message to reach a single specific remote client, the server is wasting resources by sending the message to other remote clients.

By extending the functionality of the server process, many resources could be saved. Such an extension would involve introducing a distinction between messages that are intended for a single recipient and ones intended to be broadcasted to all remote clients. For example, when sending the client's collected ICE candidates, the broadcasting version would be desired so that the message reaches all remote clients.

## 5.2 Dynamic Video Calling Application

In local testing, the application works perfectly well with multiple users. However, when executing the application on a cloud server, some problems were run into.

### 5.2.1 Performance

Having tested the application on a cloud server I was able to detect problems I would otherwise not have. When I ran the application with two users and moved my character, it seemed to jump from one position to another on the other user's screen instead of moving smoothly. However, while running the application on my local machine, the movement of my character on the remote client's screen was just as smooth as it was on my own. The performance of my application seemed to deteriorate significantly when run on the cloud server compared to my local machine.

The obvious difference between the two testing environments is the time taken for messages to move between clients. On the cloud server the messages will take much longer to reach the server and the remote client compared to the local machine. Although this is the case, in similar applications such as Gather, the movement of everyone's characters is smooth and at worst the remote character's position is behind by several hundred milliseconds.

I was not able to identify the exact cause of the lack of performance on the cloud server. One possible source may be Links' internal implementation of message sending that, given a process ID and message, sends the message and appends it to the process' mailbox.

### 5.2.2 Communicating Image URLs

In the initial implementation of my application, I had each client send the URL of their image along with their character's position every time they moved position. With this implementation, I found that the application ran significantly slow until it crashed my browser. Once I realised I was doing this, I updated the implementation so that each client sends the URL of their icon image only once when they click the button to take the picture of themselves. After making this change, the application ran much faster, as if the icon image feature of the application was not introduced yet.

### 5.2.3 Scalability

Assessing the scalability of my application involves measuring the number of users who can use it at the same time while it functions correctly. On my local machine I am able to connect up to six, sometimes seven users at once before the application slows down significantly, video calls fail to start, and errors appear in the developer console. However, on the cloud server this number is significantly reduced to 3 before these problems occur.

It is understandable that the application starts to deteriorate at around seven concurrent users as each user has a connection with six other users, which amounts to  $7 \cdot 6 = 42$  total connections. However, malfunctioning at only three concurrent users on the cloud server is not ideal. I tested a simpler application where the client begins a video call with all connected remote clients immediately, without significant message sending to communicate character positions and image URLs. In this application, I was able to connect about six users at once on the cloud server, which suggests that the problem lies within the message sending between clients via the server. This links back to Links' internal implementation of message sending.

### 5.2.4 MVU's Dispatch Feature

Although intended to improve the readability and performance of combining concurrent process communication with the MVU framework, it seems to have slowed down the performance of my application considerably. Switching between running the application on my local machine and on the cloud server amounts to negligible difference in performance. Since the MVU dispatch feature release was tested and worked fine on a simple program, this leads me to believe that there may be incompatibilities between this feature and the nature of my application.

### 5.2.5 Lack of Feature Support

In terms of features, my application provides a dynamic environment for video calling. Users are able to take a picture of themselves to indicate who they are and move around the page together using only their arrow keys, and also begin a video call with each other when near.

However, there are many standard video calling features that my application lacks

support for. Such features include muting and hiding the user's and other users' microphones and cameras, screen sharing, and sharing documents.

# Chapter 6

## Discussion

Throughout this chapter, I discuss conclusions made about my live streaming API and dynamic video calling application, as well as potential plans for the extension of my project next year.

### 6.1 Conclusions

Regarding the live streaming API, and referring back to the project goals in Section 1.2, I can conclude that I have successfully designed and implemented a flexible API in Links, and built a dynamic video calling application on top of it.

The API is flexible as it presents the programmer with a range of functions that are not tightly coupled in the sense that the media devices part can be used without the WebRTC connection functions.

I also successfully achieved the dynamic video calling application goal, since my application fulfils the main goal description in Section 1.2. However, one goal not achieved was the implementation of buttons that allow the user to toggle the microphone and camera. The reason for this was that I spent the majority of my time fixing bugs in the main application, as this was my priority,

### 6.2 Future Developments

There are multiple potential ways I can extend my project for next year. These are subject to change between now and the beginning of the next semester.

#### 6.2.1 Scalability and Performance

As seen in Section 5.2.2, my application suffers greatly to both scalability and performance. Next year, I could look into the potential sources of error, such as the internal implementation of message sending and other aspects of Links.

### **6.2.2 Implement Same Application in React, Standard Links, and JavaScript Alone**

In an attempt to assess the performance of the MVU framework, I could implement the same dynamic video calling application in other ways such as through the JavaScript library React, JavaScript itself, or using the imperative method of manipulating the DOM in Links.

### **6.2.3 Scale to Multiple Users Without Video Server**

Although I have not yet used a video server to improve the application's performance, one ambitious extension could be to implement the WebRTC API to have, for example, as most three connections coming from any single user. This would involve learning about video compositing and network theory, and formulating a creative method of accomplishing this successfully.

### **6.2.4 User Study**

Lastly, I could implement more interesting features in my application such as sound attenuation, broadcasting audio further than video, and many more features to enhance the usability and accessibility of the application. Then, I could carry out a user study to assess the usability and accessibility aspects of my application.

# Bibliography

- [1] MediaDevices API. <https://developer.mozilla.org/en-us/docs/web/api/mediadevices>. (Accessed: 20/09/2021).
- [2] WebRTC API. [https://developer.mozilla.org/en-us/docs/web/api/webrtc\\_api](https://developer.mozilla.org/en-us/docs/web/api/webrtc_api). (Accessed: 20/09/2021).
- [3] John Koetsier. Top 10 Most Downloaded Apps and Games of 2021. <https://www.forbes.com/sites/johnkoetsier/2021/12/27/top-10-most-downloaded-apps-and-games-of-2021-tiktok-telegram-big-winners/>, 2021. (Accessed: 04/03/2022).
- [4] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. <https://links-lang.org/papers/links-fmco06.pdf>. 2006. (Accessed: 15/09/2021).
- [5] Simon Fowler. Allow direct dispatch of messages to MVU event loop. <https://github.com/links-lang/links/pull/1098>, February 2022. (Accessed: 01/04/2022).
- [6] eambutu. Launch HN: Gather.Town (YC S19). <https://news.ycombinator.com/item?id=25039370>, 2020. (Accessed: 29/09/2021).
- [7] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular Actor Formalism for Artificial Intelligence. <https://www.ijcai.org/proceedings/73/papers/027b.pdf>. 1973. (Accessed: 13/03/2022).
- [8] Naomi Jacobs and Joseph Lindley. Room For Improvement in the Video Conferencing “Space”. <https://doi.org/10.5210/spir.v2021i0.12188>, 2021. (Accessed: 16/03/2022).
- [9] The Links Programming Language. <https://links-lang.org/>. (Accessed: 15/09/2021).
- [10] Joe Parlock. Wizards Of The Coast’s “Magic The Gathering” Strixhaven Virtual PR Event Was A Masterclass In Pandemic Event Planning. <https://www.forbes.com/sites/joeparlock/2021/04/19/wizards-of-the-coasts-magic-the-gathering-strixhaven-virtual-pr-event-was-a-masterclass-in-pandemic-event-planning/>, 2021. (Accessed: 16/03/2022).

- [11] Simon Fowler. Model-View-Update-Communicate: Session Types Meet the Elm Architecture. <https://drops.dagstuhl.de/opus/volltexte/2020/13171/>, 2020. (Accessed: 30/11/2021).
- [12] Behnam Mohammadi. How to create a GUID / UUID. <https://stackoverflow.com/questions/105034/how-to-create-a-guid-uuid/>, 2017. (Accessed: 25/10/2021).
- [13] Gretchen McCulloch. A Mission to Make Virtual Parties Actually Fun. <https://www.wired.com/story/zoom-parties-proximity-chat/>, 2020. (Accessed: 16/03/2022).
- [14] Sara Saez-Fajardo. Gather Town Brings Mingle Activities to Online Language Teaching. <https://fltmag.com/gather-town-mingle/>, 2021. (Accessed: 07/04/2022).
- [15] Sam Lindley. Abstract types for alien objects. <https://github.com/links-lang/links/issues/1099>, February 2022. (Accessed: 01/04/2022).
- [16] Jirka Hladis. Multi User Video Chat With WebRTC. <https://www.dmcinfo.com/latest-thinking/blog/id/9852/multi-user-video-chat-with-webrtc>, 2019. (Accessed: 15/09/2021).
- [17] CSS Website. <https://www.w3.org/tr/css/>. (Accessed: 08/03/2022).
- [18] Elm Website. <https://elm-lang.org/>. (Accessed: 07/04/2022).
- [19] Gather Town Website. <https://www.gather.town/>. (Accessed: 17/09/2021).
- [20] GroupRoom Website. <https://www.grouproom.io/>. (Accessed: 16/03/2022).
- [21] HTML Website. <https://html.spec.whatwg.org/>. (Accessed: 08/03/2022).
- [22] JavaScript Website. <https://www.javascript.com/>. (Accessed: 07/03/2022).
- [23] Mozilla Hubs Website. <https://hubs.mozilla.com/>. (Accessed: 16/03/2022).
- [24] Node.js Website. <https://nodejs.org/en/>. (Accessed: 13/03/2022).
- [25] React Website. <https://reactjs.org/>. (Accessed: 15/03/2022).
- [26] Skype Website. <https://www.skype.com/en/>. (Accessed: 04/03/2022).
- [27] WebRTC Website. <https://webrtc.org/>. (Accessed: 15/09/2021).
- [28] Zoom Website. <https://zoom.us/>. (Accessed: 04/03/2022).



# Appendix A

## Live Streaming API Documentation

### Functions Designed for External Use

**getUUID : () -> Uuid**

Returns the UUID of the local client.

**gatherDeviceIds : (DeviceType) -> ()**

Takes as input the type of media device to gather device IDs and labels for. This creates a promise that will not finish immediately, so it is important to make use of the wait function shown below.

**waitForDeviceIds : (DeviceType) -> ()**

Wait function. This waits until all media devices have been gathered by waiting until the promise finishes.

**getDeviceIds : (DeviceType) -> [DeviceID]**

Takes as input the type of media device to get the device IDs for. Returns a list of device IDs.

**getDeviceLabels : (DeviceType) -> [DeviceLabel]**

Takes as input the type of media device to get the device labels for. Returns a list of device labels.

**readyMediaDevices : (DeviceID, DeviceID) -> ()**

Takes as input the IDs of the media devices to be prepared. This creates a promise that will not finish immediately, so it is important to make use of the wait functions shown below.

**waitForCamera : () -> ()**

Wait function. This waits until the requested video device's stream has been accessed by waiting until the promise finishes.

**waitForMic : () -> ()**

Wait function. This waits until the requested audio device's stream has been accessed by waiting until the promise finishes.

**registerUser : () -> ()**

Creates a new client process and sends the ID of this process to the server to register it. It then waits in a loop for messages to arrive from the server.

**connectToUser : (Uuid) -> ()**

Takes as input the UUID of the remote client to begin a WebRTC connection with. It sets up the important internal details of the local client's side of the connection, and sends a connection request message to the remote client to connect to.

**disconnectFromUser : (Uuid) -> ()**

Takes as input the UUID of the remote client to end a WebRTC connection with. It then deletes the internal details of the connection on both clients.

**checkIfConnectedToPeer : (Uuid) -> Bool**

Takes as input the UUID of the remote client and checks whether a WebRTC connection exists already with the remote client.

**Internal Functions****broadcast : ([Process], [Message : PCMessage]) -> ()**

Server function. Takes as input a list of client process IDs and a message and sends the message to every client process in the list.

**serverLoop : ([Process]) -> ()**

Server function. This function waits for messages to arrive from client processes and carries out different actions depending on the received message. Given Register(pid), it adds the client process to its list of client processes in the parameter. Given Broadcast(msg), it sends the message to all registered client processes.

**handleIceCandidates : () -> ()**

Repeatedly makes calls to `_collectCandidates` to check whether any ICE candidates have been found. If so, sends the stringified JavaScript object to all client processes registered to the server process.

**waitUntilLocalDescSetForPC : (Uuid) -> ()**

Takes as input the UUID of the remote client whose WebRTC connection is being completed. Repeatedly checked whether the local client's local description has been set in the `RTCPeerConnection` object associated with the input UUID.

**waitUntilRemoteDescSetForPC : (Uuid) -> ()**

Takes as input the UUID of the remote client whose WebRTC connection is being completed. Repeatedly checked whether the local client's remote description has been set in the `RTCPeerConnection` object associated with the input UUID.

**prepareDescriptionForPC : (Uuid, OfferOrAnswer) -> ()**

Sets the local description of the `RTCPeerConnection` object associated with the input UUID and calls `waitUntilLocalDescSetForPC` to wait for the promise to finish.

**sendDescriptionForPC : (Uuid, OfferOrAnswer) -> ()**

Gets the local client's local description from the `RTCPeerConnection` object associated with the input UUID and sends it to the remote client with the input UUID. The SDP type is either "offer" or "answer".

**setUpNewPeer : (Uuid, Bool) -> ()**

Calls `_setUpPC` to instantiate the `RTCPeerConnection` object. If the input Bool is true, calls `prepareDescriptionForPC` and `sendDescriptionForPC`.

**handleOfferForPC : (Uuid, Desc) -> ()**

Called when an SDP message was received by the client. Calls `_setRemoteDescForPC` to set the remote description of the `RTCPeerConnection` object associated with the input UUID as the input SDP. Calls `waitUntilRemoteDescSetForPC` to wait until the promise finishes. Calls `prepareDescriptionForPC` and `sendDescriptionForPC`.

**handleMessage : (PCMessage) -> ()**

Takes as input a message received from a remote client and runs different blocks of code depending on the message type.

**clientLoop : () -> ()**

Waits for messages to arrive from the server. If a Message(msg) is received, it calls handleMessage to handle the received message.

**clientRegister : () -> ()**

Sets the local client's UUID and sends a Register message to the server process.