A Video-Calling API for Links

Weston Everett

MInf Project (Part 2) Report

Master of Informatics
School of Informatics
University of Edinburgh

2023

Abstract

This project focuses on the development of a generalized video-calling API for the web-focused functional programming language Links, building on the application-specific APIs built in previous projects. Similar to those projects, this API relies on Google's WebRTC library accessed through the Links Foreign Function Interface to implement the underlying functionality of the API. To be considered a "generalized" API, this project was built to accommodate as many potential video-calling applications as possible. Designing and implementing this API required solving both conceptual challenges such as designing an underlying call structure that could be used in many different applications, as well as more practical challenges such as implementing the actual calling functionality within Links.

To properly evaluate this API, a range of applications were produced including simple apps to demonstrate a particular API feature as well as the re-implementation of relatively complex applications. When used to directly convert existing Links video-calling applications the API reduced the code programmers had to write by at least 40% in every tested case. Additionally, a framework known as the "Cognitive Dimension Framework" was used to shape a more qualitative assessment. The final results of this project include a robust API that makes setting up video-calls within Links trivial and functions both in the MVU and Actor-based styles of Links programming, as well as a collection of example projects making use of the API.

Table of Contents

Chapter 1

Introduction

Last year, two video-calling applications were written by myself [7] and another student [14] using the Links programming language [1]. As Links does not have a built-in library or method to perform video-calling, both of these projects built application-specific interfaces for video-calling. These made use of Links' Foreign Function Interface to access the JavaScript WebRTC library, which provided the video-calling functionality. While functional for their intended use, these APIs were designed and built for a specific application in a specific design framework and are therefore difficult to apply in other circumstances.

Building this sort of application-specific interface required the programmer to have an understanding both of WebRTC and of how it can interact with Links. Additionally, the design and implementation of these APIs required a great deal of work. Ideally, a programmer would not have to know the intricacies of WebRTC or write hundreds of lines of code to include video-calling in a Links application. Therefore, this project will focus on the development of a generalized video-calling API for Links that can easily be used to implement as many different applications as possible. This API should require as little knowledge of the underlying systems as possible, while still allowing essentially any video-calling application to be implemented.

1.1 Previous Work

Last year, I produced one of the two projects focusing on the development of a video-calling application within Links. This project focused on "Dynamic Video Conferencing", where users of the application could freely and easily move between different group video calls, then modify the internal state of those group calls to their liking. While initially based on Gather.town (which allows continuous movement and proximity-based calling), this video-calling app was most similar to applications such as Discord, as it used a simple mouse-based interface to move between discrete "rooms". When a user entered one of these rooms they automatically began peer to peer calls with all other users in the room.

Within these group video calls, a rudimentary interface allowed users to selectively activate and deactivate audio/video feeds to and from other participants. This functionality combined with "macros" in the UI for affecting large numbers of calls at once allowed the creation of group calls with complex, asymmetrical structures.

However, the API for this application was built specifically for this application with limited ability to generalize, and so was subject to many limitations. Many of these limitations center around how audio/video sources are selected, and how many of each may be contributed by any single user.

For example, every participant in a group call had to provide exactly one audio and video feed, even if it was never used. Users were not allowed to swap audio/video devices after they initially set them up, and were never allowed to use more than one of each. To select a device, the original program required a specific JavaScript function to be run while an HTML element with a specific configuration was viewed. Another JavaScript function then had to be run to collect the provided source data, which was held constant for any other calls made by that client.

The other important limitations were centered around reuse. As an initial hurdling block, the code for video-calling was woven into a great deal of other functionality, and therefore would be relatively time-consuming to separate for any attempt at reuse. Should the code have been separated, a potential programmer making use of it would also need to prepare HTML pages with the correct IDs to properly attach to.

Although this initial project was not particularly suitable as a generalized API for other Links applications, it provides a great deal of example code for the implementation of a more suitable generalized API. Additionally, the application provides a relatively stable way of creating Links calls, although they are subject to the mentioned limitations.

1.2 Goal and Results

The final goal of this project was to create a generalized Links API to make the creation of any future Links video-calling app significantly easier by abstracting away as much of the calling process as possible. This API should be functional in as many different Links programming paradigms as possible (focusing mainly on the more common Actor-Based and MVU systems), and should be capable of implementing the video-calling side of a wide variety of applications. The API also needs to balance maximizing the number of things it can express with limiting the ways a potential programmer could make mistakes in it (such as by "expressing" impossible things). When complete, the API should be robust, easy to install, and easy to understand.

I fulfilled this goal by:

- Researching potential use-cases of video-calling to know what the API needed to be capable of
- Researching and experimenting with multiple formats including MVU and FRP
- Designing and implementing an API video-calling structure capable of functioning in multiple formats
- Expanding on the basic API to offer functionality such as potentially asymmetric video-less calls, active-call manipulation, and multi-caller clients
- Building numerous small example apps demonstrating particular features
- Disassembling and modifying 3 large Links video-calling applications to use the new API (reducing app-specific code by 40+%)
- Conducting extensive stress and edge-case testing across multiple different devices and browsers
- Researching and applying an API evaluation metric, the "Cognitive Dimension Framework" to the API

1.3 Report Organization and Terminology

This report begins with a Background Chapter (Chapter 2) that explores the different methods of programming within Links to understand the potential challenges a universal video-calling API may need to consider. Most important here are the MVU and Actor-based systems, as these were recently used to implement video-calling applications. The Background goes on to explain WebRTC, which is a JavaScript library used for video-calling developed by Google that is fundamental to this project. Finally, a potential way of evaluating the final API is considered.

The report then looks at the final design of the API, including the fundamental design principles that were focused on, in the "Design" Chapter (Chapter 3). This chapter looks at each general "area" of the API and why it was designed that way, as well as providing a simple example of an application built using it.

The next chapter, "Implementation and API" (Chapter 4) is concerned with the final API available to potential programmers. Each function that is available to a potential programmer is provided along with its inputs and outputs, including an explanation for its use if unclear. Additionally, some of the finer details and challenges of the API are discussed.

To evaluate the API and look at what it can potentially do, the next chapter (Chapter 5) showcases a few sample applications built while the API was being developed to test various features. The chapter then covers the re-implementation of larger previously-built Links applications from a variety of programmers in order to prove that the API functions well in different frameworks/applications and to evaluate the reduction in

necessary code to implement these applications. This acts as one potential metric for how useful the API can be. Finally, the chapter evaluates the API through the lens of the "Cognitive Dimensions Framework" in order to consider the variety of factors that impact how "good" an API is.

Finally, the report concludes with the "Conclusions" (Chapter 6), which takes a wider view of the API and summarizes the key takeaways of the final product. It additionally offers suggestions for potential future developments and ways to address future issues.

Throughout this report, the distinction between the person making use of the API by programming an application, and a potential user of that application must be clear. Therefore, this report will refer to a programmer making use of the API to develop an application as a "programmer" or "application programmer", and will refer to the end users of the application as "users". Additionally, the report uses the word "client" to refer to the section of code that is run on a single users browser, including any sub-processes which may be running connected to it.

Chapter 2

Background Information

2.1 Links

2.1.1 Links Overview

Links [5] is a functional programming language for the web that focuses on "tier-less programming". This essentially means that a programmer writes all of their code in a single language, which the compiler then uses to generate code for all the different "tiers" of a web application such as the client, server, and database. This is distinct from the more traditional method, which requires the programmer to use separate languages for database and client (often SQL and JavaScript/HTML/CSS respectively).

Links itself has a few different methods of programming that an API should be usable in such as MVU (Model-View-Update) [10], Actor-Based Programming [5], and Functional Reactive Programming [3]. Any generalized API for Links should comfortably work in any of the common programming schemes for Links, which therefore warrant investigation.

2.1.2 Actor-Based Programming

2.1.2.1 Functional Description

One approach to programming non-trivial applications in Links is "Actor-Based Programming" [5], which is a form of event-driven programming where various "Actor" processes pass messages between themselves and react accordingly to the messages they receive. In the original paper on the Links language, these actors are described as concurrent processes and are the original (and default) approach to asynchronous programming in Links.

This works by concurrently running a collection of processes, each identified by a unique "process ID". The process IDs are used to send messages between the processes using the "!" symbol and relevant ID in the form of "target_id ! message". These messages can be of a variety of programmer-defined types which help the processes to

handle them correctly. For example, in a simple messaging application a client may define both a "Message" type to represent displayable messages and an "Update" type to represent state changes like another client leaving.

When receiving a request, these processes use what is referred to as a "receive-case" block that takes the top message out of the process' "Mailbox" that stores the received messages (in a First-In, First-Out manner) and parses it. This is where the defined types of messages are useful as each type of message can be conveniently linked to a specific response based on its "case". This will typically be part of a loop that continuously works its way through all of the available messages in order. A diagram of this can be seen in Figure 2.1.

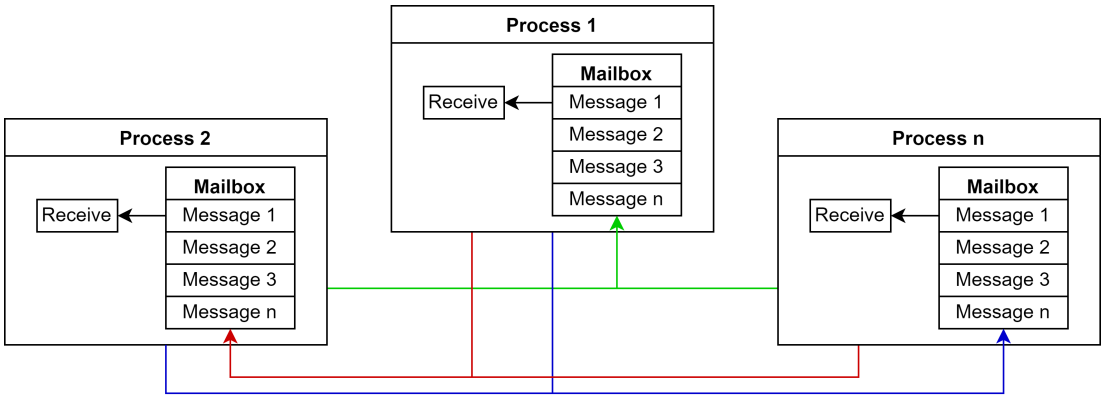

Figure 2.1: Processes with Mailbox Architecture

These processes can be created in a few different ways. The first of these are the *spawnClient* and *spawn* functions which create new processes and return an ID at which that process can be sent messages. Often, these processes are similar to those described above and contain the receive-case blocks (and run for long periods of time). However, these processes can also simply complete a function asynchronously and then end, allowing them to be used for other types of programs as well. Alternatively, new processes can be created using the *spawnWait* function. This spawning function differs from the others as it blocks the original process that is spawning a new process until the new process sends back a specific message. This allows for more control of the timing of function execution in a situation where race conditions may potentially be a problem. These processes can be distributed between different devices, with messages still being able to be passed between them [9].

When used in an application, this Actor-based style is a common way (and the original way [5]) of communicating between servers and clients within Links. However, it can also be used internally in the client for separating responsibilities between different concurrent actors, referred to as "Actor-based Programming". This may result in a somewhat-decentralized approach of handling what is displayed to users, with different actors each being responsible for writing their information to the HTML for the user.

2.1.2.2 Previous Work

The original Links paper references a simple app that makes use of this Actor-Based model which is useful for understanding the basics, where the UI sends updates to an internal Actor that manages a simple state machine[5]. Other sample Links code provides an example of this Actor-Based model being distributed across multiple Clients and a Server, showing how this model can be used for multi-user communication [8].

Finally, the precursor to this project [7] provides an implementation of a video-calling application using Actor-Based Programming methods. This "Actor-Based Calling" application also includes a limited WebRTC-based API for video-calling using an actor-based programming method. The API is written in a mixture of Links and JavaScript, and provides the following functionality:

- Procedure for the creation of a WebRTC P2P call between two actors on different clients
- Functions for ending existing calls
- JavaScript functions for selecting input devices, requiring specifically-named HTML dropdowns and page changes
- Functions for manipulation of existing WebRTC call objects such as muting

This API holds all calls on the main process, and receives the call messages directly in the main process mailbox.

The most important aspect of this API (in regards to this project) is likely the WebRTC calling procedure, as this will remain reasonably constant between applications. This procedure consists of each client setting constraints that describe their local video/audio feeds and exchanging them with one another. Additionally, both clients begin processes that search for and agree to an ICE server that both can connect to (covered more in Section 2.2). This structure can be seen later in Figure 2.2.

Although this API is relatively application and format-specific, it provides a good base for the creation of a format-agnostic, generalized API for video-calling. Additionally, the project provides one potential format that an app using Actor-Based programming may take, with a single server actor and one main client actor per client. This main client then spawns smaller actors for specific purposes.

2.1.3 Model-View-Update

2.1.3.1 Functional Overview

Generally speaking, programs in Links use the Actor model to handle the communication between client and server (as can be seen in the example projects on the Links website) [1]. However, internally within a single client there are other more unified

ways of rendering visuals for the user and storing data.

One of these methods of programming applications within Links is the MVU or "Model-View-Update" architecture brought over from Elm [10]. This draws from the original MVC or "Model-View-Communicate" system suggested in 1979 [16]. Broadly speaking, the MVU architecture consists of three main components, the first of which is the "Model". This "Model" is essentially the current state of the application. This state is processed and rendered for the user by the "View" component, which is tasked with looking at the existing data, processing it into a user-friendly format (typically HTML), and showing it to the user. When this "View" interacts with the User (or other sources like a server) it produces update messages that are sent to the final component, "Update". This component uses the input messages it receives to update the "Model" component which will then change what is displayed by the "View" component and so forth.

This architecture is closely related to the "Model-View-Controller" architecture used in the popular React framework for JavaScript [11]. This is conceptually identical (and is sometimes referred to as an MVU format), although it more directly refers to the "Update" component as a "Controller" component, which is responsible for both the "Model" storage and "View" rendering

2.1.3.2 Previous Work

A previous project implemented an MVU-based video-calling application with a spatial metaphor, which allowed users to move around a virtual space in a continuous manner and call other "nearby" users [14].

This "MVU Gather.town clone" application [15] primarily relies on MVU for tracking and displaying the locations of each user within a continuous virtual "room", and an Actor-based model for communicating between client and server (although there is mention of a new "dispatch" method which can mitigate this somewhat). This communication system is similar to the one seen in the "Actor-Based Calling" project, although it is accessed differently and must handle situations such as each client repeatedly trying to create already-existing calls (a by-product of the way MVU was used to decide which users should be calling).

This "Actor-Based Calling" project makes use of WebRTC to provide the following functionality:

- Procedure for the creation of a WebRTC P2P call between two different clients
- Functions for ending existing calls
- Functions for collecting/selecting input devices

Another project that is currently in development [13] provides a different MVU-based

system. This "MVU Hybrid" project creates a hybrid of spatial and non-spatial projects by offering alternative views of the same underlying system. This works similarly to the other MVU system in terms of video-calling, although calls are triggered on entry into a discrete room projected into continuous space, as opposed to the purely continuous space used by the "MVU Gather.town clone". Additionally, in this application the MVU framework only triggers calls/hangups on a state change, as opposed to repeatedly sending the current state.

2.1.4 Functional Reactive Programming

Functional Reactive Programming (originally Functional Reactive Animation [6]) is a proposed style of programming designed for animation and modelling. This approach to programming is described by its creators as attempting to allow users to express the "what" of an animation so that the "how" can be automated. It is typically a "language within a language", in that the "vocabulary" to describe the functionality of an application in an FRP-style is implemented within another language. Originally, it was implemented in the Haskell programming language, although a Links adaptation was built by a previous Masters student [3].

The language itself consists of continuous, time-varying values called "Behaviors" and sequences of time-ordered events each referred to as an "Event" [18]. Of these, "Behaviors" are perhaps most analogous to variables, in that they contain values of a certain type that may be changed over time. They are often used when applied to UI elements, for example the "Behavior" of the color of a button over time. "Events" are the actions that trigger changes in these "Behaviors", for example key presses and mouse clicks.

FRP programs are written by defining "Behaviors" in terms of "Events". For example, to describe a box that is blue until a button is pressed, and then turns yellow, a "Behavior" would be defined. This "Behavior" would look like the following (depending on FRP implementation):

boxColor = blue until (buttonPress $\rightarrow$ yellow)

Where boxColor is a "Behavior" for color and buttonPress is an "Event".

Using this type of definition, arbitrarily complex descriptions of programs can be built. However, there are complications in the original form of the language, as it does not necessarily require that effects happen after what caused them. For example, a declaration such as "The button's color should change three seconds before it is clicked" could be valid to declare but is impossible to actually implement. This is less of an issue when applied to pure animation, as then the future states could be calculated, but is completely impossible when the future has unpredictable elements such as a user interacting with the program. However, newer formats of FRP attempt to address this, albeit with slightly changed rules.

2.1.5 Foreign Function Interface

To make use of the JavaScript-based WebRTC API within a Links API, some method of integrating JavaScript with Links is necessary. Conveniently, the Links language provides a "Foreign Function Interface" which allows a JavaScript file to define functions that can then be registered and accessed from Links as if they were a native Links function with minimal overhead. This style can run into issues when making use of JavaScript-style asynchronous "promises" as Links does not deal with them natively. However, an extra "wait" function that can be repeatedly checked to see if the promise has resolved solves this issue reliably, if somewhat inefficiently [7][14]. This asynchronous handling is absolutely necessary for any video-calling API, as the base system used for peer-to-peer video-calling in these apps is WebRTC, which uses promises extensively.

2.2 WebRTC[1]

WebRTC [2] (or "Web Real Time Communication") is an API developed by Google which is designed to facilitate direct peer-to-peer communication of various types between browsers on different devices. According to the developers, it currently is supported by "All modern browsers". The API is available in JavaScript and therefore should be accessible through the Links Foreign Function Interface.

To understand how WebRTC works it can essentially be split up into two main segments; the MediaStream API which provides a way to access media on the user's device, and the RTCPeerConnection which allows two different devices to make a peer-to-peer connection.

2.2.1 MediaStream API

The MediaStream API allows for "media capture" which primarily consists of a microphone and webcam, but can also include things like a screen capture for screen sharing. In order to use the video/audio feed (and to get the necessary control of the devices) the API requests access to devices that follow the constraints supplied, and (if given permission from the user) provides the output of the devices as a stream. These constraints may be as loose as "any audio device" or as specific as "a video device with this ID and this resolution".

The API also has the functionality to request a list of all available devices, which can allow the user to choose which will be accessed or allow the application to select one that best fits the provided parameters. The API can also control selection using its constraints, such as setting the requested resolution for video or requiring that the selected source has audio properties. In addition, the API can listen for devices changing during runtime, for example, if the webcam being used by the application was unplugged or a new one was to be plugged in. Once this event happens, it can then trigger the proper

[1] This section is heavily based on the equivalent section from my project last year [7]

response. This feature is most effective on Google Chrome, as a previous project has shown permissions issues on other browsers [7].

The stream output by this can then be used in a variety of ways, such as splitting just the audio or video "tracks" from it (which would for instance allow muting in a video-calling app while retaining the live video). This data can also be sent to another user, using the RTCPeerConnection segment of the API.

2.2.2 RTCPeerConnection API

The RTCPeerConnection API is somewhat more complex (for someone using the API at least) as it requires multiple steps to function correctly. Typically it is used for audio and video, although it can also send arbitrary binary if the clients support RTCDataChannel (a related API).

First, clients establish a connection through signalling, which can be done through a variety of different methods. For this project, this is assumed to be done by using Links message-passing. This signalling is necessary so that both devices can be "on the same page" about how they are connecting, which typically is arranged using ICE (Internet Connectivity Establishment) servers.

After establishing the initial connection, the client will create an object to store the connection, and then select either an "offer" or "accept" behavior based on the system's current state. Once two peers have formed an initial connection an ICE service will search for and send them candidates for connecting until one is found, and the peer connection is fully established.

At any point, even before a peer-to-peer connection is established, a media stream can be attached to the connection so that as soon as the connection is ready data will begin to be sent. Similarly, a Listener can be attached to be ready to immediately receive data. This media stream (often accessed using the above "Mediastream API") can be attached or modified at any point.

A potential structure for this calling procedure can be seen in Figure 2.2. This structure was the one used in the original Actor-Based calling API.

2.3 API Design Principles

Although to an extent what constitutes a "good" API is subjective, a reasonable amount of research exists about the subject due to its impact. As with any field though, approaches and opinions differ [17]. An early paper attempting to standardize evaluation of programming environments proposes a "Cognitive Dimensions Framework" to help analyse the usability of APIs and languages consisting of the following 12 aspects [12]:

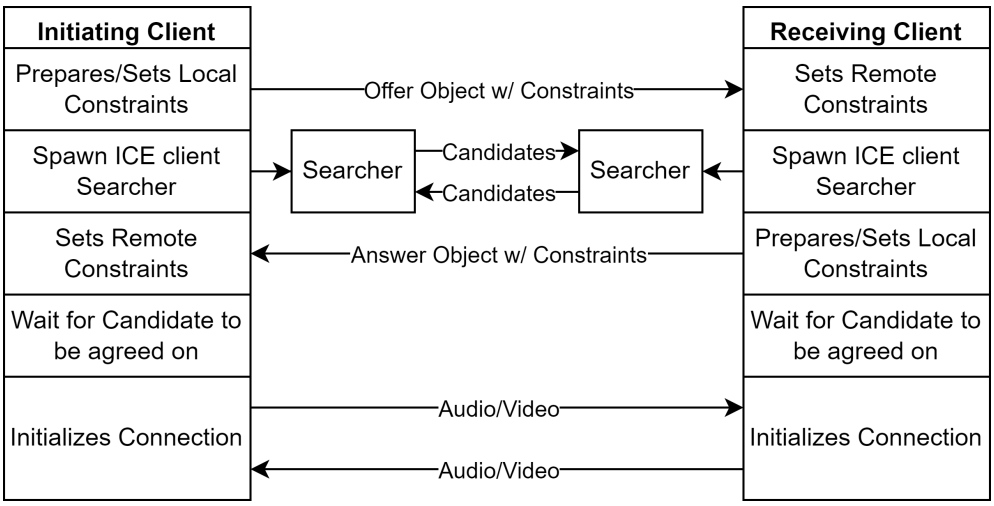

Figure 2.2: Call State Diagram

- Abstraction Gradient: How abstract is the API?
- Closeness of Mapping: How similar is the API to the concept?
- Consistency: How consistent are the terms used?
- Diffuseness: How many pieces are needed?
- Error-proneness: How easy is it to make a mistake?
- Hidden Dependencies: Is what is needed to be done obvious?
- Premature Commitment: Do decisions need to be made before all information is available?
- Progressive Evaluation: Does getting partway through implementation produce results/feedback?
- Role Expressiveness: Can the user see how each piece fits together?
- Secondary Notation: Can the user tell through other means (such as structure) how components work?
- Viscosity: How hard is it to change something?
- Visibility: how hard is it to look at all parts of the API/language in the correct order?

This framework provides a standard set of questions to pose of a potential API, although in some cases it doesn't provide a "correct answer". For example, while high consistency and low error-proneness are clearly both good and important things, it is less clear whether high abstraction is good, or how important a secondary notation is.

Other studies making use of this framework can help to address these questions and help show the values of particular aspects of this framework. For example, Microsoft-funded studies show a significant increase in productivity when programmers use a

well-designed API with good documentation, especially APIs that trend towards a higher level of abstraction [4]. These studies describe APIs that use a slightly modified form of the Cognitive Dimension Framework which nonetheless shares many aspects. While the concept of a "well-designed API" is of course somewhat hard to define, it is clear in this report that sufficiently useful documentation (and therefore, "Visibility") is paramount.

The abstraction level of the API and its relationship with productivity is more interesting. This particular study explained the productivity loss associated with lower levels of abstraction as being caused by programmers having trouble relating the low-level classes they found in the sample API to the problem they were trying to solve. This was worsened further by the low-level classes being more linked to the way that the API designers thought about the general task than any "absolute solution", since there are often many low-level ways of approaching any given issue.

The study was then repeated weeks later using a redesigned API with higher-level abstraction, and a significant increase in productivity was noted.

Chapter 3

Design

In order to create a generalized video-calling API, I began by designing an interface for a future programmer to make use of, as well as the internal structures necessary to make that possible. This interface was shaped by the requirements put forward in Section 1.2, as well as a collection of other important factors such as being "easy to use".

3.1 Design Goals

To "generalize" a video-calling API it has to be usable in as many different situations as possible, whether that is different actual applications (e.g. Zoom, Gather.town, or Twitch) or different programming frameworks (e.g. MVU or Actor-based). Additionally, the API should be as easy and simple to use for a programmer as possible, without requiring knowledge of the underlying systems.

To be considered an "easy to use" video-calling API I focused primarily on the following criteria:

- Abstraction: A user should not need to understand how the API actually functions or how video-calling works to use it
- Ease of Use: The API ought to be composed of a concise set of functions, each with a well-defined purpose
- Error Avoidance: The API should not allow the user to interact with it in unintended ways

However, to also consider the API "generalized", the following criteria also had to be considered:

- Permissiveness: The API should allow as many coherent actions[1] relevant to

[1] A "coherent action" refers to a specific action or function call that accomplishes a meaningful result. For instance, making a call to yourself, while unconventional, produces a functional outcome such as sending audio or video to your device, and therefore qualifies as a "coherent action". On the other

video-calling as possible

- Framework Independence: The API should not require actions exclusive to a certain framework such as MVU's frequent updates

3.2 Overall Client Design

In many video-calling applications, a user participates in a single call or a group call with other users, where each participant provides a single audio and video stream to all others. This design is employed by various applications, including Discord and Zoom, where users can switch between calls or group calls ("breakout rooms" in Zoom), but are never simultaneously in multiple calls. In these apps, users/clients act as a call endpoint, and select a single audio input and video input. This pair of inputs is then used for all of their calls. Based on this, a basic solution to allow calling would be to associate each user with a single audio and video feed and provide a simple interface for initiating calls to exchange these feeds.

However, this form of API is overly restrictive for a generalized video-calling API, as it limits a user/client to sending the same audio or video feed to anyone they are in a call with. Although this could be addressed by allowing the disabling of certain feeds combined with the attachment of multiple audio/video feeds to a single client (and selecting which feeds should be sent to whom), this would be quite complicated for a user and/or programmer to track. Instead, I designed a structure to act as a call endpoint, as opposed to the user/client being the endpoint.

I refer to this structure as a "CallClient", and use it to fulfill the earlier role of the client by providing a call endpoint that can read from one audio and one video source and exchange these sources with any number of peers. However, each Client can have any number of these CallClients, removing the single-call single-source limitation. Any call data received by a CallClient (such as foreign audio/video) is written to a place specified by the user/programmer. Each CallClient is identified by a unique ID that is provided on creation. This structure allows the API to be more "Permissive", extending the range of applications that the API can be used for. Additionally, by encapsulating all call-related data into an object that the programmer is only able to access in set ways, it is difficult for the programmer to make runtime errors as they are unable access the underlying process directly.

This structure can be seen in Figure 3.1, where each CallClient has a unique ID and is associated with only one Client. Each of these CallClients can call any number of other CallClients, feeding an audio and video stream to the specified place in that client.

To illustrate how this API would be used, this report will go through the creation of a

hand, attempting to call a nonexistent entity would not achieve any outcome, and therefore would not be considered coherent.

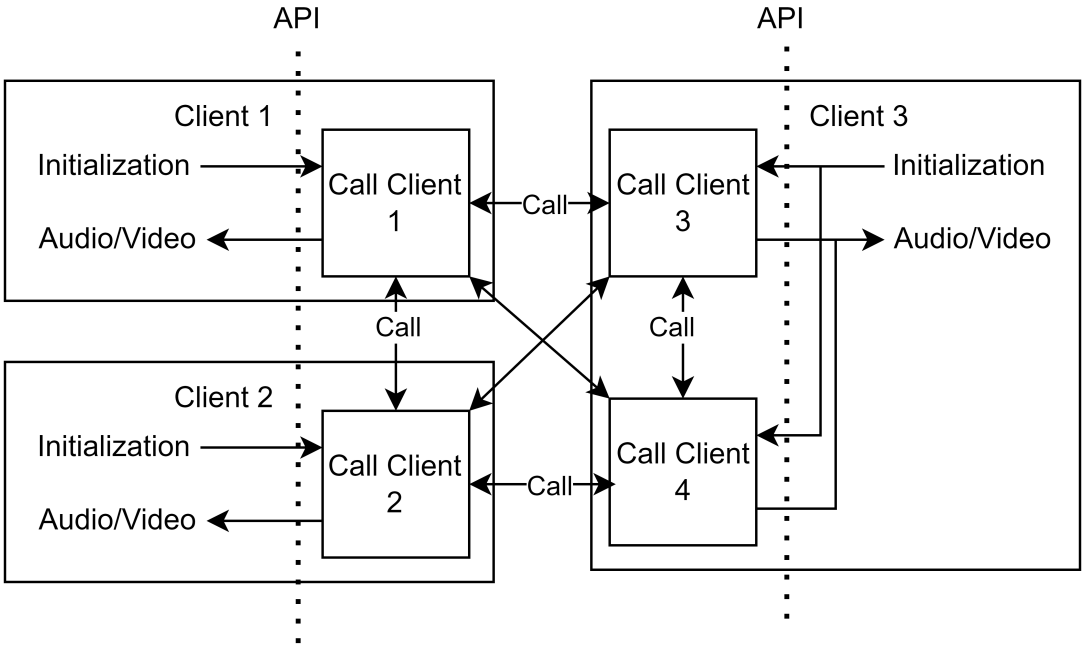

Figure 3.1: API CallClient Overview

basic video-calling application. This will make use of a simple server that tracks the process IDs of each client (to send information to clients) and allows "Register" messages to add new clients, "Broadcast" messages to send data to all connected clients and "Send" messages to send data to a specific client. The specific Links code to define this server can be seen in Appendix A.1.

Using this server as a baseline, an application programmer can associate each user/client with a CallClient using the following code, which creates a client with an attached CallClient for each new User:

```
fun clientLoop(myID) {
  receive {
    ##Call Logic goes here
  }
}

initializeClient() {
  serverPid ! Register(self());
  var myID = VidAPI.startClient("defaultWriteLocation");
  clientLoop(myID)
}

fun mainPage() {
  var clientPid = spawnClient{initializeClient()};
```

```
  page
    <html>
      <div id = "defaultWriteLocation"/>
    </html>
}
```

3.3 Device Selection

Of course, simply having a CallClient with a write location to the page a user sees isn't enough to create a meaningful call between two users. Some sort of video or audio feed to send between the users as part of the call is also necessary.

The API enables this by providing two important functions. First, a way for the programmer to access what audio and video devices are available in a recognizable format is necessary. This is provided by the *getSources* function, which returns an array of the ID Strings of audio devices and video devices. These ID-Strings are generally understandable by humans and can be directly displayed/selected from when using Google Chrome. However, other browsers sometimes obscure this information (anonymizing them to "Microphone 1", "Webcam 2", etc). It can be important to note here that *getSources* does not require a CallClient ID as it is determined by the devices a Client/User has access to, rather than the internal CallClient construct.

Once a video and an audio source have been decided on, the *setSources* function can be used to set the sources for a specific CallClient. Therefore, it requires the ID of the CallClient, as well as a video or audio source. However, audio or video can be left blank, in which case the function leaves the existing audio or video source in place. This was done so that audio and video sources can be selected and updated piecemeal, without the programmer having to track what was last selected. Finally, functions are available to remove existing audio and video source settings without providing a replacement, leaving those aspects empty.

Neither audio nor video sources are technically required to begin a call when using this API, as a potential programmer may want to use the API for a non-video call, or perhaps simply stream video. There is also no requirement that two callers exchange an equal number of streams, so it is also possible for one call participant to send both audio and video while the other sends nothing. This provides a way for potential users who do not have (or do not wish to use) a webcam or microphone to still participate in calls without all of the typically-used devices. Note that on this level, every CallClient sends the same data to every other CallClient it is connected to (although this is modified later in Section 3.4). This allows call structures similar to the ones shown in Figure 3.2.

In Figure 3.2:

- CallClient 1: calling CallClients 2 and 3, sending video
- CallClient 2: calling CallClients 1 and 4, sending audio and video
- CallClient 3: calling CallClients 1 and 4, sending audio
- CallClient 1: calling CallClients 2 and 3, sending nothing

It is important to note here that, for example, CallClient 3 is sending audio to CallClient 4 but receiving nothing back, as CallClient 4 does not have any source devices attached. This holds true for all CallClient calls, where each sends everything it can and receives anything provided back.

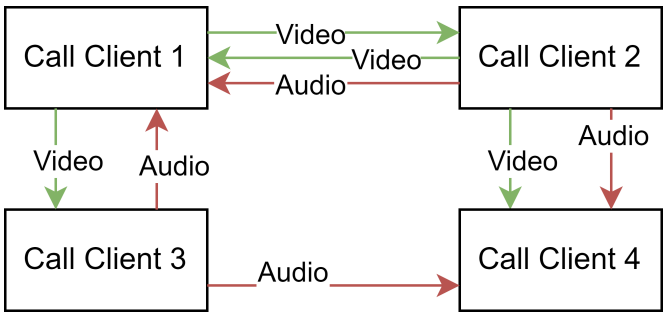

Figure 3.2: Asymmetrical Call

Currently, the set audio and video sources at the beginning of a call are held constant (for that specific call) for the duration of that call. Should a programmer wish to update the devices used in a call live, they can currently reset devices, then hangup and call again.

To extend our earlier example web-calling app to make use of this functionality, we simply replace the initializeClient function with one that also sets an audio and video device for the created client. This is done by simply selecting the first available device for simplicity, in real apps it would likely be preferable to allow the user to select a device by providing an interface for selecting between the devices provided by the API. The simple extension can be seen in the following code ():

```
initializeClient() {
  serverPid ! Register(self());
  var myID = VidAPI.startClient("defaultWriteLocation");
  var (audioSources, videoSources) = VidAPI.getSources();
  VidAPI.setSources(myID, hd(audioSources), hd(videoSources));
  clientLoop(myID)
}
```

An alternative and likely better way of doing this is to allow the internal WebRTC API to select which device to use. This can be done by using the following code instead:

```
initializeClient() {
```

```
  serverPid ! Register(self());
  var myID = VidAPI.startClient("defaultWriteLocation");
  VidAPI.anyAudioSource(myID);
  VidAPI.anyVideoSource(myID);
  clientLoop(myID)
}
```

3.4 Calling

A video-calling app also requires the ability to start calls. I provide this functionality through the *callClient* function, which requires the IDs of the two CallClients that a call should begin between. Unlike most of the previous functions of the API, this does not need to be run from any specific client, which allows any client to begin calls between any two clients. This style of call organization was used as it allows for apps to be created where a single user controls the structure of the calls more easily.

For example, this structure would allow an application programmer to create an app where all calls are centrally controlled by the server, instead of having to synchronize logic between multiple different Clients. The functions to hangup existing calls work in a similar way, with the function calls to end a connection functioning non-locally.

Extending our previous client with this call functionality produces the following code, which creates a simple application that begins a group call between any users who join:

```
fun clientLoop(myID) {
  receive {
      case ServerMessage(id) ->
        if(id <> myID){
          VidAPI.callClient(myID, id);
          clientLoop(myID)
        } else {
          clientLoop(myID)
        }
      }
  }

fun initializeClient() {
  serverPid ! Register(self());
  var myID = VidAPI.startClient("defaultWriteLocation");
  var (audioSources, videoSources) = VidAPI.getSources();
  VidAPI.setSources(myID, hd(audioSources), hd(videoSources));
  serverPid ! Broadcast(myID);
  clientLoop(myID)
}
```

```
fun mainPage() {

  var clientPid = spawnClient {initializeClient()};

  page
    <html>
      <div id = "defaultWriteLocation"/>
    </html>
}
```

3.5 Active-Call Manipulation

With calls in place, it is likely that programmers will want to be able to offer their users the ability to, for example, mute other callers without fully disconnecting a call. This functionality is implemented in a similar way to the "Actor-Based Calling" project [7] (covered in Section 2.1.2.2), where each aspect of a call (incoming and outgoing video/audio) can be disabled from either end of the call. This can be seen in Figure 3.3 (taken from the previous project paper) where each client can choose to enable or disable each of the 4 options. These options are made available through the functions *setIncomingAudio*, *setIncomingVideo*, *setOutgoingAudio*, and *setOutgoingVideo*, which each require the ID of the source client, the ID of the target client, and a Boolean representing the target state of the manipulated stream.

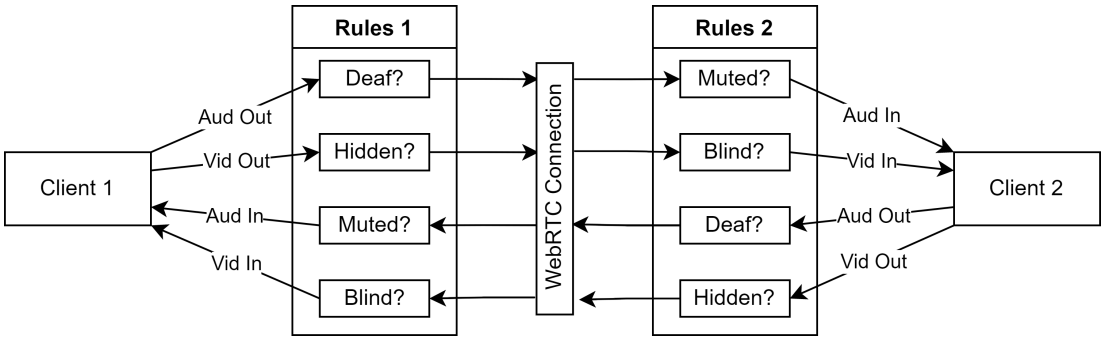

Figure 3.3: Call Rules

Should we wish to extend the earlier example to allow active call manipulation, we would have to extend the Links code to ensure all clients know that all other clients exist. To do this we can make use of the Send case in the simple server (see A.1) to notify any new client of all the IDs of existing clients. For this example, we only add the ability to enable and disable incoming video, as it is the easiest to see in a still image (and to keep the example to a reasonable length).

The logic for extending the client to track foreign IDs is long and unnecessary for understanding this API so it is left to the appendix A.2. However, the following code provides an example of a way to access the active-call manipulation functionality for a user. This is done by extending the "mainpage" HTML to contain a div with the ID

"buttons" and calling the following function whenever a new ID is discovered in order to allow users to interact with the video streams.

```
fun addButtons(myID, id) {
  var buttonDiv =
    <div>
      <button type="submit"
      l:onclick="{VidAPI.setIncomingVideo(myID, id, false)}">
      Disable Incoming Video of {stringToXml(id)}
      </button>

      <button type="submit"
      l:onclick="{VidAPI.setIncomingVideo(myID, id, true)}">
      Enable Incoming Video of {stringToXml(id)}
      </button>
    </div>;

    appendChildren(buttonDiv, getNodeById("buttons"))
}
```

Running this sample application would give the following view (Figure 3.4) for a user who is calling 3 other users but has disabled the video of user 2 (using the functionality added above).

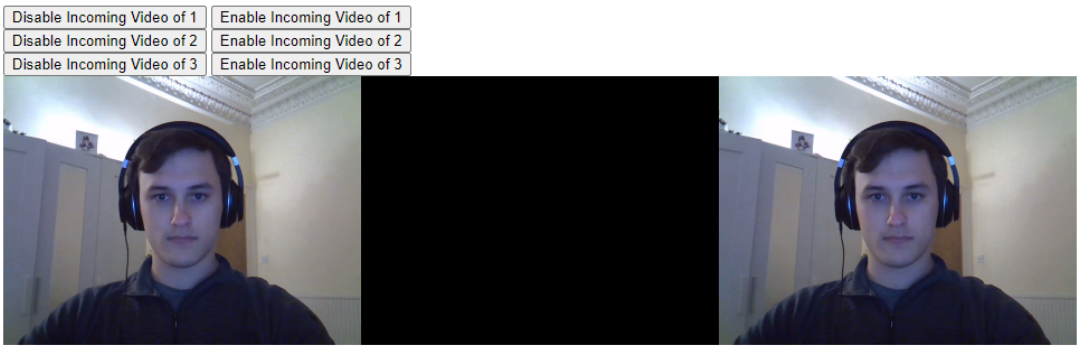

Figure 3.4: Basic Calling App with Incoming Video Manipulation

3.6 API Usage

As mentioned earlier, a programmer using this API is responsible for keeping track of the IDs of each CallClient, as these are used by the programmer to identify which clients they want calls to form between. Programmers are additionally responsible for providing an HTML div that videos can be written to, as well as the ID identifying that div. Should they wish to, it can be useful for programmers to track the internal state of their calls (for example, whether client 1 has muted client 2), however, this is not necessary in most cases.

Beyond that information, the programmer should not have to track any other information about the active calls, with a reasonable amount of information available through miscellaneous functions (covered in 4.4.1).

Chapter 4

Implementation and API

With the broad design of the API covered, a collection of more specific challenges remained. This section will go through the functions I built for this API that are available to a programmer by section (for example, functions concerned with setting up a CallClient). Notable challenges I encountered when implementing each grouping of functions will also be covered.

4.1 Client Initialization

This grouping consists of functions that are concerned with initializing and preparing a CallClient before calls can take place.

4.1.1 API

The API provides the following functions for initializing and modifying CallClients:

- *startClient* : (String defaultWriteLocation) → (String clientID)
- *setWriteLoc* : (String myID, String foreignID, String newWriteLocation) → ()
- *getSources* : () → ([String] audioSources, [String] videoSources)
- *setSources* : (String myID, String audioSource, String videoSource) → ()
- *clearAudioSource* : (String myID) → ()
- *anyAudioSource* : (String myID) → ()
- *clearVideoSource* : (String myID) → ()
- *anyVideoSource* : (String myID) → ()

The CallClient created by *startClient* will run locally with the process that started it, writing any received video and audio to the location given by defaultWriteLocation. The location that the CallClient writes calls to based on which CallClients are involved can then be set using *setWriteLoc*.

The audio and video sources available to the relevant process can be accessed using *getSources*. Once a maximum of one audio and one video source has been selected, *setSources* can be used to set the audio and video source of the specified CallClient. Sources can be left blank if desired if audio or video sources are unavailable.

4.1.2 Implementation

The *startClient* functionality is the most important here, as it influences the rest of the API. It works by using Links "SpawnWait" function to block the process it is called from while it initializes a process on the client (the "CallClient") and then returns an ID unique to the CallClient. It is important to note here that the unique ID is not the same as the process ID (Explained in Section 2.1.2), and therefore programmers may only interact with it through the API's provided functionality. Should this use the process ID, the programmer could use the send (!) functionality in Links (also explained in Section 2.1.2) to send the process unexpected messages, potentially causing unexpected behavior.

A server (whose behavior I defined internally in the API) is notified of any new client initializations and stores a mapping between client ID and process ID. This client ID can then be dereferenced by the server to retrieve the relevant process ID and contact the associated client when necessary due to non-local functions (see Client Calling). The structure created by this internal-API server process and the associated CallClients can be seen in Figure 4.1.

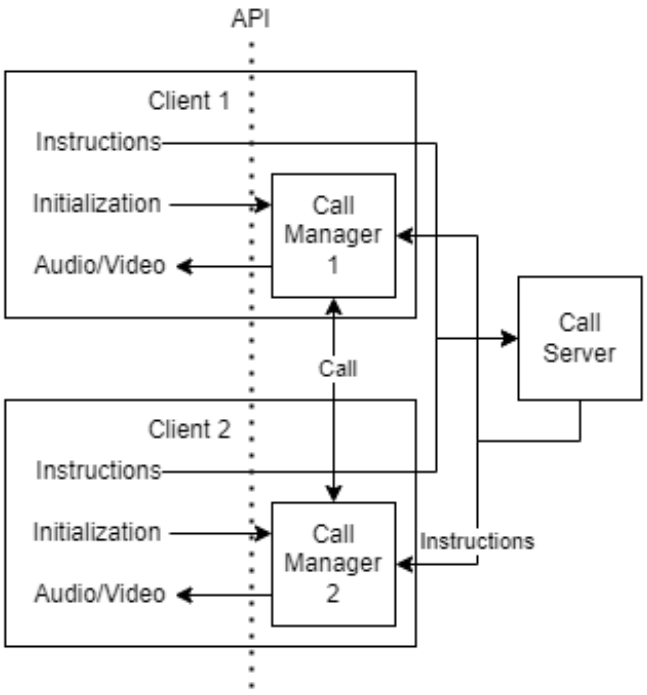

Figure 4.1: Internal Server-Client Structure

The location written to by each CallClient is decided by a look-up table stored locally

per client (rather than per CallClient). When any new call begins, the API first checks the table to see if a connection-specific location has been set (using *setWriteLoc*), and uses that location if available. Otherwise, the call is written to the default location set on CallClient creation.

Similarly, a local look-up table is used to store CallClient audio/video source preferences. This table is set and updated by *setSources* (which sets the audio and/or video source when provided with a non empty string), and *clearAudioSource* or *clearVideoSource*, which remove the relevant existing entry. This table is only checked on the beginning of a call, meaning that sources cannot currently be manipulated live.

4.2 Client Calling

4.2.1 API

The API provides the following functions for starting and ending calls:

- *callClient* : (String myID, String foreignID) $\rightarrow$ ()
- *hangup* : (String myID, String foreignID) $\rightarrow$ ()
- *hangupAll* : (String myID) $\rightarrow$ ()

These functions can be called from anywhere, and cause the CallClient with the first provided ID (myID) to call or hangup a call with the other designated client (foreignID).

4.2.2 Implementation

The *callClient* function, as mentioned earlier, is written so that it can be called from any client whether or not the CallClients involved are running locally. This is done by taking advantage of the internal API server covered in Section 4.1.2, with the API passing a message from the current client to the server, which uses the provided unique ID to identify the intended source CallClient of the call. The server then instructs this CallClient to initialize a call.

The actual process of calling is implemented similarly to the previous project, with the structure seen in Figure 2.2. However, I have made significant modifications to allow the programmer more control over what is done and to extend the framework to allow multiple CallClients on a single client. One of these modifications is that the individual call objects are now accessed and stored in a simple tree structure, as a single JavaScript instance may have to contain several parallel CallClients. To illustrate this, the call storage of a sample user with 2 CallClients calls could look like the structure shown in Figure 4.2.

In this example, there are 4 CallClients (0, 1, 2, and 3) which each have called one

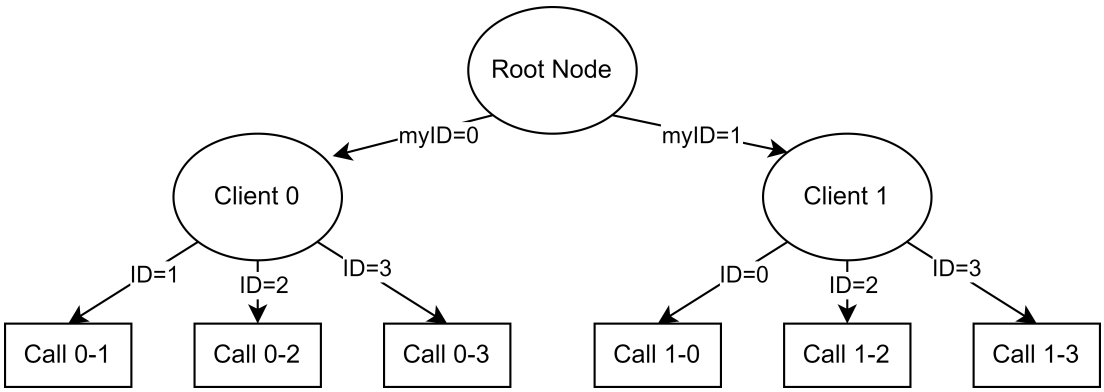

Figure 4.2: Sample Call Storage

another at some point. In the case that two CallClients had never called each other, a call endpoint object would not exist at the relevant leaf. Any new local CallClients are added to the first layer of the tree (just under the root node) with calls appearing under the relevant local client.

4.3 Client Call Manipulation

4.3.1 API

The API provides the following functions for manipulating existing calls:

- *setIncomingAudio* : (String myID, String foreignID, Bool setTo) $\rightarrow$ ()
- *setIncomingVideo* : (String myID, String foreignID, Bool setTo) $\rightarrow$ ()
- *setOutgoingAudio* : (String myID, String foreignID, Bool setTo) $\rightarrow$ ()
- *setOutgoingVideo* : (String myID, String foreignID, Bool setTo) $\rightarrow$ ()

These functions allow the temporary modification of existing calls, as described in Section 3.5. Similar to other functions, the first ID provided identifies the “Source” CallClient. Together, the “Source” ID and the “Foreign” ID describe a unique call endpoint to be manipulated

4.3.2 Implementation

These functions are implemented by taking advantage of the “enable/disable” feature of WebRTC streams, allowing video and audio streams to be modified independently. The choice to have the programmer provide a boolean to set the enable state to, rather than a toggle function (which would require less arguments) is so that the programmer does not need to track or check the internal state of the call modifications, as they can simply provide the call with the intended state of each track.

4.4 Auxiliary and State-checking

4.4.1 API

The API provides the following functions for checking the current state, as well as other miscellaneous functionality:

- *playLocalVideo* : (String myID) → ()
- *getConnectedIDs* : (String myID) → ()
- *checkIfConnected* : (String myID, String foreignID) → ()

The functions *getConnectedIDs* and *checkIfConnected* allow a programmer to see which other CallClients their CallClients are connected to, while *playLocalVideo* simply plays the video feed connected to the CallClient in the set location. The location this is played can be modified using the *setWriteLoc* function (by setting the write location for calls between the selected CallClient and itself).

4.4.2 Implementation

Information-gathering functions are relatively simple to implement and are therefore easy to extend to provide other information. The latter two functions, *getConnectedIDs* and *checkIfConnected* are of this type, and simply traverse the call storage tree mentioned above and filter the results.

The other function, playLcocalVideo, is essentially a special case of *callClient*. It reads the set video source and writes it to the relevant location mentioned earlier. It ignores the audio component so as not to play the user's own audio back to them.

Chapter 5

Evaluation

In order to evaluate this API, I took two major approaches. The first of these is using the API to construct video-calling applications, including the modification of existing applications. This allows for the comparison of quantitative metrics between applications using custom APIs and the proposed API. The other major method was through the use of the "Cognitive Dimension Framework" covered in section 2.3, which considers the API using several different metrics.

Evaluation by implementation is important because it mirrors the actual use case of this API. It is particularly useful as it can provide proof that the project is suitably general through building and re-implementing projects in multiple frameworks. This consists of both smaller apps that were built to demonstrate the API's basic features, as well as re-implementations of all three previous projects.

5.1 Case Studies

As the purpose of the sample projects was to test out and show the basic features of the API, I kept the case studies relatively simple and similar to one another.

5.1.1 Basic App

In order to test the basic call functionality between callers, I built a simple test application similar to the example shown in Section 3.4. This application can be seen below in Figure 5.1, where a user with two clients (and one on another device) is shown. The local CallClients, (CallClient 0 and 1) are calling each other, and CallClient 1 is also calling the non-local CallClient 2. Therefore, 3 videos are shown.

The code for this application consists of approximately 100 lines of code, mostly forming the structure of a server-client relationship to correctly pass notifications between clients. These notifications are used to notify all clients when a new CallClient is produced so that a new call can be started. The code also produces the HTML elements

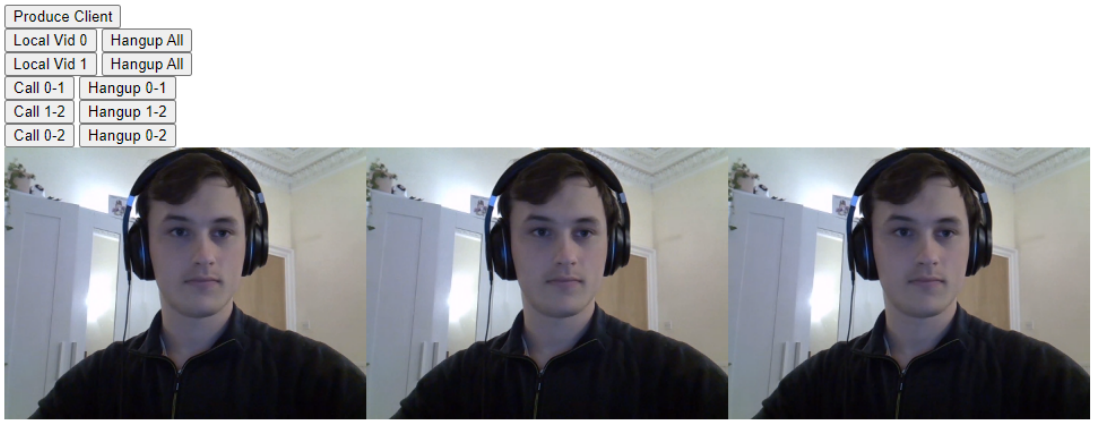

Figure 5.1: Base Calling App

seen in the image. Of these, 7 lines used function calls from the API to handle all calling and hanging up.

Experimentation and stress testing on this application showed the system to be relatively robust, with no fatal issues occurring. Notably, this API fixes a synchronization issue appearing in previous projects, where users quickly hangup and re-call causing de-synchronization issues [7]. This is done by triggering *hangup* on both ends simultaneously as opposed to waiting for the side that did not initiate the *hangup* to notice a dead connection and disconnect (which often takes a couple seconds). This solution to the bug is possible due to the decentralization of certain functions covered earlier (Section 4.2).

I used this application to test the potential call capacity of a system relying on the API by creating a large number of clients on a single device, then beginning calls between them until the video/audio streams began to break down. This testing was done in a similar way to the previous project [7], which noted performance degradation with an estimated 20 single-end calls, with a full breakdown occurring by 30 single-end calls at the latest.

When testing this application, performance degradation began to appear when there were 14 calls. As both endpoints of these calls were on a single device, we double this to give us an estimate of 28 single-end calls, which is roughly comparable to the earlier project. It is interesting to note here that performance degradation appeared to get worse over time, with the 28 single-end call benchmark used here not showing performance degradation until approximately a minute in. This limit is potentially linked to the available computational power of the computer running the calls, as the systems task manager showed large amounts of CPU power being used by the processes.

5.1.2 Input Selection

I built the next sample application to evaluate and test the input selection functionality of the API. To do this, each CallClient is allocated a series of buttons allowing its sources to be overwritten during live calls. This app can be seen in Figure 5.2 in which a single user has two CallClients, each with a different video source. These CallClients are calling each other, and therefore both video feeds are displayed.

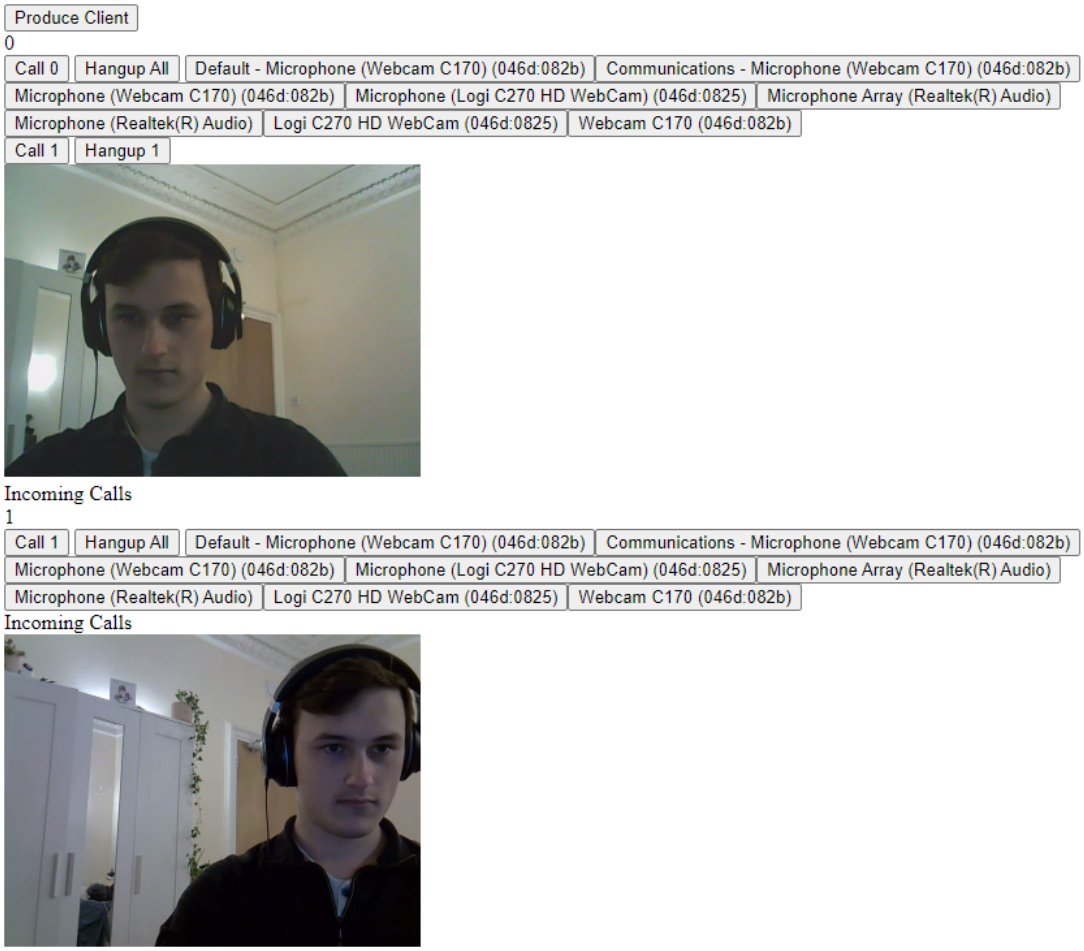

Figure 5.2: Input Selection: Two Video Feeds

This sample application consists of about 150 lines of Links code, with a significant proportion of the increase (relative to the last sample app) due to the increased code structure required to clearly identify the different CallClients. These 150 lines include 10 lines calling the API in order to build the video-calling functionality.

The sample application allows the evaluation of the input selection component of the API. By comparing the available devices (the names on the displayed buttons) with the available devices on the computer, I confirmed that the API displayed the correct devices with recognizable names. Additionally, as the application is able to correctly read from and display two different video devices simultaneously on the same device,

the selection works as expected. Finally, I used this application to show that the calling API would work for CallClients whether or not they had access to audio and/or video sources. The CallClients were still being able to send anything they had access to and receive anything sent by any connected CallClients. This was shown by creating a call similar to the one shown, but with one CallClient providing only video and the other providing only audio.

There are also some important drawbacks to this system. First, while testing with non-Chrome browsers the API is unable to access information about connected devices without directly using them (as otherwise the browser never asks for permission). This means that the *getSources* function of the API is unreliable outside of Chrome, as it relies on getting information about connected devices before accessing them. Although full audio/video control would not be available the API could still attach sources to CallClients using the *anyAudioSource* and *anyVideoSource* functions, albeit with less control. This could be somewhat addressed by adding a method of providing constraints to the API rather than selecting by device ID (which is protected by permission constraints). Finally, sources do not currently live-update on existing calls when changed, although this would be relatively easy to modify the API to do.

5.1.3 Call Manipulation

I built the final sample application to evaluate the call manipulation aspects of the API. This is application is very similar to the app described in Section 5.1.2. However, instead of buttons for selecting a specific source, buttons for manipulating the video/audio of connections were added. This app can be seen in Figure 5.3, where two clients on a single device are calling each other, although one has "hidden" the other by disabling the incoming video of the call.

This app consists of about 160 lines of Links code, including 18 API function calls (primarily repeats to hook up the mute/deafen/etc buttons).

Testing this app shows that the call manipulation functions work as expected, with the ability to correctly modify all available calls. When a call missing (for example) a video stream receives instructions to enable or disable video streams, the API simply does nothing. This is the desired behavior as otherwise a potential programmer would have to consider the current sources of a call and could easily cause runtime errors by muting non-existent audio streams.

5.2 Re-implementation of Existing Systems

I re-implemented three existing applications in order to test this API on larger applications with varying designs. These applications were the "Actor-Based Calling" project [7] (covered in 2.1.2.2) which is an Actor-based Discord/Zoom-like project, the "MVU Gather.town clone"[14] (covered in 2.1.3.2), and the "MVU Hybrid" project

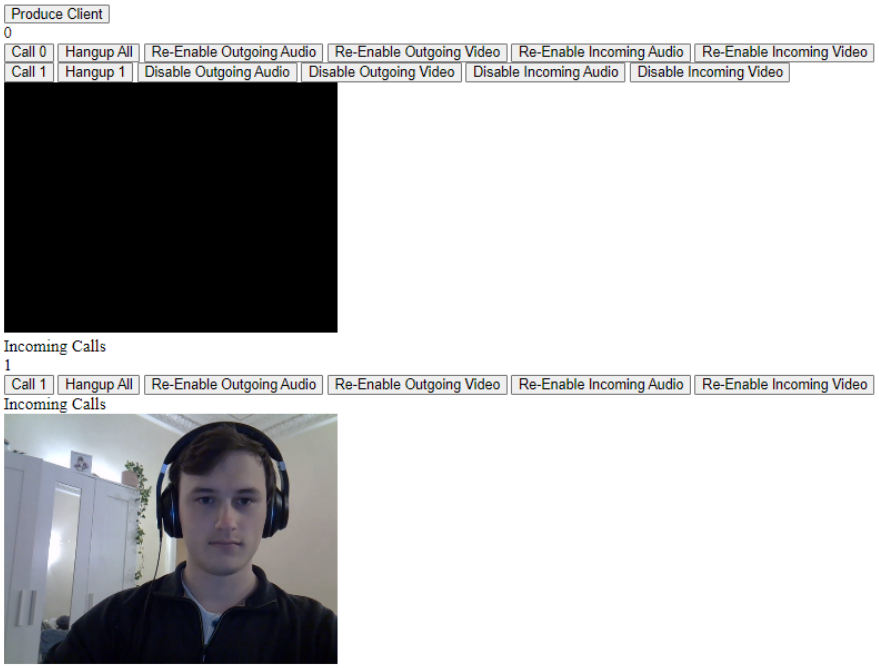

Figure 5.3: Call Manipulation: Hidden Call

which is a new MVU app that is essentially a hybrid of the other two applications[13] (also covered in 2.1.3.2).

Across the board, re-implementing other applications using this API reduced the code required significantly (about 40% fewer lines) and offered an easy implementation of additional features at a lower cost. However, the style of direct conversion I used to modify these apps had issues when attempting to convert apps that blended their calling functionality with other functions, or relied on JavaScript code keyed to specific HTML IDs.

5.2.1 "Actor-Based Calling" Re-implementation

First, I re-implemented the video-calling app I produced for my previous dissertation. This was unsurprisingly the easiest and smoothest re-implementation, as I was already familiar with the project and had designed the new API partially based on what I had learned implementing the original project.

Last year, I wrote approximately 765 lines of Links code and 300 lines of JavaScript code to implement the complete project. When doing a relatively shallow (only rewriting when absolutely necessary) re-implementation of this project I had to write about 630 lines of Links code and no JavaScript, a reduction of 435 lines total (or about 40% of the project). This could likely be increased further if a full rewrite was done as there is likely redundant information and functions designed for calling left in the code. Additionally, by replacing the original code with my new API, I added new features such as no longer requiring both audio and video sources and a more robust

calling experience (as the quickly hanging up and recalling bug no longer happened). The new API also enabled extensions such users in multiple rooms or routing/labelling calls according to the call's source.

5.2.2 'MVU Gather.town clone" Re-implementation

I then re-implemented the "MVU Gather.town clone" application (covered in Section 2.1.3.2. This application uses MVU to create relatively smooth movement, and applies proximity-based calling as opposed to discrete rooms. Re-implementing this app proved to be particularly challenging due to the app's heavy reliance on JavaScript and CSS code, a relatively fuzzy line between calling code and structural/feature code, and a reliance on specific HTML IDs throughout the code. For an app like this to use the proposed API well, it would likely be best to write it from scratch rather than attempting to redirect the existing code's dependencies.

However, I did eventually convert the application to make use of the proposed API for calling, although certain features such as taking a picture to represent each user had to be disabled due to existing issues (likely hardware-linked), and errors were often shown. The original code required approximately 970 lines of Links code, 610 of JavaScript, and some CSS that is not considered as it was disabled in the re-implementation. The re-implemented code required about 610 lines of Links code and about 150 lines of JavaScript, representing approximately a 50% decrease in necessary code. This figure is relatively imprecise, as it is likely code would have to be added or removed to fix the current errors.

5.2.3 "MVU Hybrid" Re-implementation

Finally, I re-implemented an MVU-based app that is essentially a hybrid between the earlier two apps. This app overlays a spatial metaphor on discrete rooms to form call rooms, creating multiple views of the same underlying system. This app proved significantly easier to convert than the Gather.town clone, with a comparable level of difficulty to the Actor-based app. This is primarily because the app was built mostly in Links and clearly separates the calling functionality from the other features of the app.

The original app required the programmer to write about 560 lines of Links code and about 190 lines of JavaScript. The re-implementation using the proposed API required the programmer to write about 450 lines of Links code and no JavaScript, representing a reduction of about 40% of the code. This re-implementation would also allow the relatively easy addition of features like audio/video source selection and live-call manipulation (muting/deafening/etc) to the project.

Converting this project using the proposed API provides evidence that the API can work relatively seamlessly in an MVU format as well as evidence that it can work in other programmer's code.

5.3 Cognitive Dimension Framework Evaluation

An alternative way of evaluating an API is to use the Cognitive Dimensions Framework (see section 2.3) to judge the API over a set of different aspects, which helps provide a more structured evaluation of quality. Therefore, this section will go through the 12 aspects of the Cognitive Dimensions Framework and evaluate the API with them when possible.

Abstraction Gradient: The API is typically quite abstract, especially when used concerning calls as a great deal of detail is omitted (such as the individual stages of a call or functioning of WebRTC which are abstracted by the CallClient structure). However, some exceptions do exist to allow finer control where necessary, such as the *getSources* and *setSources* functions that directly access and display available devices.

Closeness of Mapping: The structure of the program maps closely to the structures associated with video-calling. For example, each individual CallClient can be conceptualized as a call endpoint familiar to most such as a mobile phone. Similar to a mobile phone, each CallClient reads from one audio and video source, can call multiple other people simultaneously (through group calls), and can temporarily disable any audio or video connection it has. Additionally, any one person (or User/Client) can choose to use multiple different mobile phones simultaneously, each with their own audio/video and own calls.

Consistency: Most aspects of the API are consistent, with for example the functions *callClient* and *hangup* accepting the same ordering of IDs and applying the expected functionality from that view. However, there are some parts of the API that trade off consistency for other benefits. One of these is the local and non-local groupings of functions, where functions like *callClient*, *hangup*, and *hangupAll* can affect any process from any other process, while other functions like *getSources* only get sources from the local machine. Although there are benefits to doing things this way (simplifying implementation and use of functions that do not require non-locality) it does come at the cost of some consistency.

Diffuseness: In this case, diffuseness is taken as the number of separate concepts required to use the base functionality of the API. To make use of this API, the following independent actions must be taken: an HTML location must be created to receive a call, a CallClient must be created with a reference to the HTML location, the intended audio and video sources must be selected and applied to the CallClient, and finally, the CallClients ID must be passed to another client for a call to begin. Although this program could be less diffuse by, for example, integrating source selection into CallClient creation by automatically selecting any available source, this would come at the cost of available functionality, because then specific sources could no longer be specified. Therefore, the API is relatively concise, with no simple ways to make it more concise without sacrificing other aspects.

Error-proneness: The API was designed to minimize Error-proneness as much as possible by removing access to the underlying systems except through the provided functions. For example, CallClients within the API are referenced by a generated ID that only the API server process can de-reference as it prevents the programmer from sending CallClients unintended messages. Errors are still possible, but the number of ways to make an error using this API have been constrained as much as possible without damaging functionality.

Hidden Dependencies: Technically speaking, Google Chrome is a hidden dependency of certain functions in the API (mentioned earlier in Section 5.1.2). Other than this the VidAPI JavaScript files could be considered a hidden dependency as they need to be added to a folder the app has access to, although this is simply part of the installation instructions. Otherwise, the API is only reliant on the current Links language and JavaScript/WebRTC release.

Premature Commitment: This facet isn't really applicable to this API as all information is available while programming.

Progressive Evaluation: Some components of progressive evaluation are present as there are either console messages displayed when a CallClient goes through a "step" or something is displayed to the screen (such as a call still displaying the video even if audio was improperly connected). This allows a programmer to see that they are getting closer to a functional project even if their implementation currently has issues.

Role Expressiveness: Role expressiveness is relatively difficult to evaluate in an unbiased manner as it is particularly opinion-based, and would likely require some form of user survey to come to a well-supported conclusion. However, the names of functions and the provided API are consistent and say what each function does, with the provided documentation 4 giving further clarification.

Secondary Notation: This API largely does not have a secondary notation beyond the names/inputs of the functions.

Viscosity: As the function calls of the API are relatively simple and independent (meaning that information provided in previous function calls does not change what information should be provided in future function calls), existing applications should be reasonably easy to modify, especially for small changes like changing the default write location of a function. For example, if a programmer designs the application created throughout Section 3, but decided to implement different logic for selecting an audio/video source, they could modify only the code associated with selecting sources without breaking any other code written using the API.

Visibility: Visibility does not really apply for this API, although the closeness of mapping to other calling devices familiar to users such as a phone should provide some suggestion for functional ordering.

Chapter 6

Conclusions

6.1 Goals

This projects main goal was to extend the existing functionality from last year's project to create a video-calling API that could be used in as many Links frameworks as possible. This API should then be able to handle the video-calling aspect of any application. The primary factors to consider when building this API were the expressiveness of the API (how many different video-calling activities could be produced), and the conciseness of the API (how many functions the API consists of).

This API is necessary as currently whenever a video-calling application is built using Links the programmer must understand how to properly use WebRTC, then find a way to integrate WebRTC into their Links program using the Foreign Function Interface. This is a large amount of effort that is likely tangential to their goal of creating a specific application in Links, and provides countless opportunities to make mistakes and introduce bugs. Instead, this API should provide a way of abstracting away any knowledge of WebRTC and producing video-calling applications using a simple, easy to understand API.

6.1.1 API Expressiveness

To be expressive the API should be able to "express" a wide variety of concepts and situations. For example, the same API should be usable by a programmer wishing to create a complex video-calling app with many overlapping calls and devices, as well as a programmer wishing to make a simple app that calls one other predefined person. The API should also work whether the Links programmer chooses to use an Actor-Based design or an MVU one, and should not require workarounds in either case.

6.1.2 API Conciseness

The "conciseness" of the API can be considered in two main ways. The first of these is in terms of the amount of code a programmer has to write to produce functional video-calling apps. Clearly, this is a very important aspect of a usable API, as no one would use an API that increases the amount of code you need to write to accomplish a particular task.

The other way to consider the "conciseness" of the API is to look at the number of expressions in the API, whether through directly counting the functions that the API provides or through looking at the "orthogonality" of the functions made available. Here, the "orthogonality" of a function is viewed as what the function does that no other function available in the API does, avoiding unnecessary redundancy. This form of conciseness is desirable as fewer functions available to the programmer means there are fewer functions that the programmer needs to become familiar with.

6.2 Evaluation

6.2.1 Resulting API

The final API consists of approximately 20 functions, generally organized into 4 categories: CallClient initialization, calling, active-call manipulation, and state-checking and auxiliary functions. The available CallClient initialization functions enable the API to create "CallClients" and select up to one audio and one video feed each. These CallClients can then be connected in calls using the calling functions such as *callClient*, which control which CallClients are connected at any given time. Finally, these connections can be further modified using the active-call manipulation functions, allowing programmers to introduce functionality like "Mute".

Through a collection of example applications built with this API, the API has been shown to be robust (assuming Google Chrome is used), with no known bugs after extensive testing. However, other browsers can cause permissions issues when using this API, such as Firefox which disables the *getSources* function. This is probably because *getSources* requests information about all potential devices, not just a single specific device. The API can scale to up to around 28 simultaneous active video-audio calls, which is comparable if not superior to the API built for last year's project (20-30 calls).

This API was evaluated using the Cognitive Dimensions Framework to attempt to judge different facets of it separately. This analysis can be found in Section 5.3.

6.2.2 Project Re-implementation

To truly test how useful the API can be, and achieve quantitative metrics of its performance, the API was used to re-implement 3 existing Links video-calling applications. These projects include an Actor-based app similar to Zoom and Discord [7], an MVU

app emulating Gather.town with spatial-calling [14], and finally an MVU app that is essentially a hybrid of the previous two [13].

Re-implementation worked well on the Actor-based app and the MVU hybrid app, reducing required programmer code by at least 40% in both apps while allowing asymmetric calls and easing further feature expansion (such as reading multiple video devices simultaneously). This proves that the API can work well in both Actor-based and MVU environments, while substantially decreasing programmer workload.

However, re-implementation of the MVU Gather.town app was only a partial success, as the original app relied heavily on JavaScript and had multiple internal dependencies between the calling functionality and a variety of other functions. After an intensive rewrite, the application worked to a limited extent, reducing lines of code by about 50%. However, this rewritten application was very bug prone, and a full rewrite of the application from scratch would likely be necessary to produce a stable app using the proposed API. This was not done, as the re-implementation of the hybrid app showed that MVU continuous-movement apps were possible with the API.

6.3 Challenges

Similarly to the last project, one challenge was integrating Links and JavaScript functionality, as Links and JavaScript use asynchronous models (JavaScript promises vs Links processes). This applies more generally to Links as well, as it can be difficult to find documentation of the available functions. This was significantly less of an issue than it was last year primarily due to experience, as this is my second year working with Links.

When designing the API, the primary challenges consisted of coming up with a working design that would allow users to hold multiple calls over many devices simultaneously. This was further complicated as the functions could not block execution, as this could then block overall program execution if a future programmer called one in their main process. This challenge was approached by separating out a great deal of functionality into specific processes (referred to in the report as “CallClients”). However, this introduced its own issues. For example, the JavaScript memory is shared between all CallClients associated with one user/browser, which means the shared memory had to be carefully managed to keep the different CallClients separated.

Within the API, what functionality was made available to the programmer had to be carefully balanced, as a variety of design goals conflicted. For example, the API should be expressive, encouraging a large number of functions that each do relatively little to give the programmer as much control as possible. Contradicting this, the API should also be concise and relatively abstract, implying that there should be fairly few functions.

Finally, one of the largest challenges of the project was converting other's projects to use the produced API, as these are academic projects produced by other students and are therefore not necessarily standardized or easy to understand. Additionally, these programs often went about aspects of video-calling in slightly different ways, which required tracking down to match the functionality.

6.4 Future Work

Should this project be expanded further, there are a few potential avenues that could be explored. The first of these is an extension allowing programmers to specify input devices by abstract constraints such as "Any camera with dimensions 1024 by 768", as this could avoid the non-Chrome issues mentioned earlier. Ideally, when new constraints are fed to the API these could then update the sources of any active call, as currently the constraints are only checked on the beginning of a new call.

A more ambitious extension would be integrating WebRTC arbitrary data streaming capabilities to allow activities like file transfers and screen sharing through this API. Although this is somewhat beyond the scope of a video-calling API, it would still be very useful in many video-calling applications, especially screen sharing which is reasonably ubiquitous.

Bibliography

[1] URL: https://links-lang.org.

[2] URL: https://webrtc.org/.

[3] Chi-Feng Chou. "Functional reactive animation in SVG for the web via Links". In: 2011. URL: https://links-lang.org/papers/mscs/chifeng.pdf.

[4] Steven Clarke. "Describing and measuring API usability with the cognitive dimensions". In: *Cognitive Dimensions of Notations 10th Anniversary Workshop*. Vol. 16. Citeseer. 2005.

[5] Ezra Cooper et al. "Links: Web Programming Without Tiers". In: *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*. Ed. by Frank S. de Boer et al. Vol. 4709. Lecture Notes in Computer Science. Springer, 2006, pp. 266–296. DOI: 10.1007/978-3-540-74792-5_12. URL: https://doi.org/10.1007/978-3-540-74792-5%5C_12.

[6] Conal Elliott and Paul Hudak. "Functional Reactive Animation". In: *International Conference on Functional Programming*. 1997. URL: http://conal.net/papers/icfp97/.

[7] Weston Everett. "A Dynamic Video Conferencing App in Links". In: 2022. URL: https://links-lang.org/papers/undergrads/ug4_20222813.pdf.

[8] Simon Fowler. *Distrib.links*. URL: https://gist.github.com/SimonJF/cac22ed65c43c452a6fbde52213bb425.

[9] Simon Fowler. *Distribution*. https://github.com/links-lang/links/wiki/Distribution. 2017.

[10] Simon Fowler. "Model-View-Update-Communicate: Session Types Meet the Elm Architecture". In: *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Ed. by Robert Hirschfeld and Tobias Pape. Vol. 166. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 14:1–14:28. ISBN: 978-3-95977-154-2. DOI: 10.4230/LIPIcs.ECOOP.2020.14. URL: https://drops.dagstuhl.de/opus/volltexte/2020/13171.

[11] Cory Gackenheimer. "What Is React?" In: *Introduction to React*. Berkeley, CA: Apress, 2015, pp. 1–20. ISBN: 978-1-4842-1245-5. DOI: 10.1007/978-1-4842-1245-5_1. URL: https://doi.org/10.1007/978-1-4842-1245-5_1.

[12] T.R.G. Green and M. Petre. "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework". In: *Journal of Visual Languages Computing* 7.2 (1996), pp. 131–174. ISSN: 1045-926X. DOI: https://

doi.org/10.1006/jvlc.1996.0009. URL: https://www.sciencedirect.com/science/article/pii/S1045926X96900099.

[13] Caitlyn McDougall. *alternative-views*. https://github.com/caitlin-mcdougall/links_project/tree/alternative-views. 2023.

[14] Lewis Raeburn. "A Spatial Metaphor for Dynamic Video Calling in Links". In: 2022. URL: https://links-lang.org/papers/undergrads/ug4_20223144.pdf.

[15] Lewis Raeburn. *dynamic-video*. https://github.com/L3R4/dynamic-video. 2022.

[16] Trygve Mikjel H Reenskaug. "The original MVC reports". In: (1979).

[17] Jeffrey Stylos and Brad Myers. "Mapping the Space of API Design Decisions". In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. 2007, pp. 50–60. DOI: 10.1109/VLHCC.2007.44.

[18] Zhanyong Wan and Paul Hudak. "Functional Reactive Programming from First Principles". In: *SIGPLAN Not.* 35.5 (May 2000), pp. 242–252. ISSN: 0362-1340. DOI: 10.1145/358438.349331. URL: https://doi.org/10.1145/358438.349331.

Appendix A

Links Code

A.1 Basic Links Server

```
fun broadcast(clients, msg) server {
  switch (clients) {
    case [] -> ()
    case clientPid ! msg ->
      clientPid ! ServerBroadcast(clientPid, msg);
      broadcast(clients, msg)
  }
}

fun serverLoop(clients) server {
  receive {
    case Register(clientPid) ->
      serverLoop(clientPid :: clients)

    case Broadcast(msg) {
      broadcast(clients, msg);
      serverLoop(clients)
    }

    case Send(clientPid, msg) {
      clientPid ! ServerBroadcast(clientPid, msg);
      serverLoop(clients)
    }
  }
}

var serverPid = spawn { serverLoop([]) };

##Client and mainPage() go here
```

```
fun main() {
  addStaticRoute("/js", "js", [("js", "text/javascript")]);
  addRoute("/", fun(_) {mainPage()});

  serveWebSockets();
  servePages()
}
```

A.2 Basic Video-Calling App

```
import VidAPI;

##Server
fun broadcast(clients, msg) server {
    switch (clients) {
        case [] -> ()
        case clientPid::clients ->
            clientPid ! ServerMessage(msg);
            broadcast(clients, msg)
    }
}

fun serverLoop(clients) server {

  receive {

    case Register(clientPid) ->
      serverLoop(clientPid :: clients)

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

    case Send(clientPid, msg) ->
      clientPid ! ServerMessage(msg);
      serverLoop(clients)

    }
}

var serverPid = spawn { serverLoop([]) };

##Client
fun clientLoop(myID) {
  receive {
      case ServerMessage(id) ->
```

```
        if(id <> myID){
          VidAPI.callClient(myID, id);
          clientLoop(myID)
        } else {
          clientLoop(myID)
        }
      }
  }

fun initializeClient() {
  serverPid ! Register(self());
  var myID = VidAPI.startClient("defaultWriteLocation");
  var (audioSources, videoSources) = VidAPI.getSources();
  VidAPI.setSources(myID, hd(audioSources), hd(videoSources));
  serverPid ! Broadcast(myID);
  clientLoop(myID)
}

fun mainPage() {

  var clientPid = spawnClient {initializeClient()};

  page
    <html>
      <div id = "defaultWriteLocation"/>
    </html>
}

fun main() {
    # Spawns a process on the server which keeps track of all clients
    # Registers the "mainPage" route
    addStaticRoute("/js", "js", [("js", "text/javascript")]);
    addRoute("/", fun(_) { mainPage() });
    # Starts the server and distribution
    serveWebsockets();
    servePages()
}
main()
```