A Dynamic Video Conferencing App in Links

Weston Everett

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

2022

Abstract

This project is focused on the development of a video calling interface for the web-focused functional programming language Links, which will then be used to implement a dynamic video-conferencing app to show its applications and explore new functionality in that space. To enable peer-to-peer video calling within the Links programming language, the Links Foreign Function Interface was used to access the WebRTC library developed by Google. This required finding solutions to multiple inter-language challenges such as Links lack of Promise support, timing issues, and custom object serialization. The developed API was then used to build an implementation of a video-calling interface, which was initially based on Gather.town due to its recent surge in popularity and novel features. However, the final project is more similar to apps like Zoom and Discord, with multiple clearly delineated conversations visible on a single server controlled by a simple mouse-based interface. In order to provide more control to individual users within a conversation, an interface was implemented to allow users to define what information (audio/video) they will send and receive for each call they are a part of. This allows for very dynamic and asymmetrical calls that are impossible on most platforms, as users can form “sub-conversations” that can still hear the larger conversation but only send audio to one another. The final system is robust under normal usage, and single-device local testing suggests a single-room capacity of 20-30 users with durations exceeding 16 hours.

Table of Contents

Chapter 1

Introduction

The topic "A Dynamic Video Conferencing App in Links" can cleanly be split into two parts: "A Dynamic Video Conferencing App..." and "...in Links" because they each bring their own collection of challenges. These will be initially approached separately, as they have almost entirely independent background material and context. They will then be approached together in the Implementation (Section 3), as the Links API is used to build the video conferencing app. On the video conferencing app side I used Gather.town [7] as an initial model due to its recent sudden increase in popularity, especially during the pandemic. I generally attempted to design a website that could capture the parts of Gather.town that make it useful and widespread, while controlling for emergent behavior within it that causes issues. On the Links programming language side, I built an API for web-calling in Links [13] using JavaScript's WebRTC [1] and Links Foreign Function Interface, then used it to implement the app that I designed.

1.1 "A Dynamic Video Conferencing App..."

As the title may suggest, the most visible goal of this project is to make a video conferencing app that mimics the functionality of popular apps, and to explore a similar but alternate set of features in an attempt to improve upon it. This aspect of the project is primarily motivated by a large trend towards Work-From-Home and online events seen during the recent pandemic, as this increase in popularity means more apps (and therefore more diversity of function) are likely to see actual use. Therefore, investigation into possible new features that could become part of other apps is warranted.

The main app that will be studied for this project is Gather.town, as it has recently seen a large surge in popularity, boasting millions of users on a site with uncommon UI and novel features. For those who are unfamiliar with Gather.town, it is a video calling service on the web where users are given small video game-like characters that they move around the map, automatically video-calling other users who are within a certain radius of them (this is explored further in the Background, section 2.1). A multitude of other similar sites were also considered, giving a wide variety of experimental features to take inspiration from. These extend from relatively lightweight interfaces such as Sococo's mouse based interface [9] to even full-VR like Mozilla Hubs [10].

The main quantifiable benefit of apps like Gather.town is the ease with which small sub-conversation may form, with small groups of people being able to wander away from a larger group to continue their own conversation without disturbing the others. Being able to see who is in which group is also important so that each user has the information necessary to know which group they'd like to join, which is somewhat captured by Gather.town by looking around the screen, although not fully as Gather.towns vision does not extend over all conversations. Other popular apps also include forms of this functionality as well, such as Zoom's "breakout rooms" [21], or Discord's multiple-channel servers [4]. This is a feature that apps of this style must mimic, as it is at the core of what makes many of these systems useful.

However, Gather.town also has features and emergent behavior that users take issue with, such as the "fuzzy boundaries" of the calls, where users are unable to tell who can or can't see/hear them. This is further exacerbated by the awkwardness around directly joining a conversation online, without the various non-verbal cues that the conversation is open to new people found in real life. Some of this is controlled by the use of the table objects in Gather.town which limit the conversation to only the people sitting in the specific seats. However, while at a table it is difficult to see other tables as the screen dims making it hard to recognize who is in other conversations. The tables are also slow to change between as users must use the WASD movement system (using the W,A,S, and D keys as arrow keys, often seen in video games) to walk between them. Another common problem is that people unfamiliar with computers or video games specifically have found themselves getting motion sickness caused by the WASD movement style and screen movement of Gather.town. To find a way to improve on Gather.town, these complaints are an important place to start.

These observations simplify the design goals for the video calling website to the following:

- Multiple conversation to choose from
- Convenient way to change conversations
- Clear call boundaries
- Easy way to signal you are "Open to talk"
- Simple, static interface

To fulfill these requirements, I designed a video calling app that captures the ease of swapping conversations through the use of a collection of "Rooms" on a website that users can freely swap between using a mouse-only interface. These rooms are directly analogous to Gather.towns tables, as there are clear boundaries of who is and isn't on call with you. However, it improves on them as a user can change between rooms with a single mouse click, and can see who is in all other rooms while in one. This makes it very similar to the "channels" in Discord, although the Discord channels default to audio-only. To attempt to address Gather.towns issue with the lack of non-verbal cues, I created a feature where a user can easily invite someone to join their room by selecting them and then the room they wish the user to go to. This feature is then generalized so that any user may select any group of users and then request they all go to a room, even if the selected group does not contain that user. A comparable feature exists in Zoom and Discord, although in those cases it is limited to a small subset of users who control the servers/meetings.

With the basic advantages and disadvantages of Gather.town addressed, I then extended the flexibility of Gather.towns separate conversations by generalizing the "Mute" interface common to many services (where a user may disable their Audio input device) by applying it to both audio and video, and allowing those rules to be applied to every connection independently. In other words, each user can modify every call they are a part of to decide if they will be sending audio, sending video, receiving audio, and/or receiving video. This granular control allows a user to, for example, whisper to a friend in the middle of another conversation without either of them leaving the room, completely disable their connection with someone they don't want to interact with, or even just stop one person from hearing them. In order to be conveniently usable, this granular control requires large-scale macro buttons such as one that stops their audio from going to anyone else (analogous to the usual "Mute" button). Therefore, a selection of the most common actions were implemented such as "Blind All" which stops any call from seeing a users video and "Whisper To" which changes the state so audio is only sent to a selected few.

1.2 "...in Links"

This project was implemented in Links, which is a functional programming language designed for the web. Links has advantages for building websites because of its "tierless design", which means that the vast majority of the code is in the Links language itself, that is then translated to the required other languages for the specific task. This allows many websites, including both client and server code, to be held in a single file. Links also has other advantages, such as being a strictly typed language, which can theoretically make it more predictable (as opposed to languages like JavaScript). However, Links lacks any direct support for video calling, which is why a central part of this project is to remedy that through the development of a video calling API for the language.

I built this video calling API complete with device selection, beginning and ending calls, and the ability to modify current calls by (for example) muting them. To do this, I relied on using Link's Foreign Function Interface as a way to access JavaScript code in Links, mostly making use of the WebRTC platform. This was made more difficult by the fact that Links does not support many of the key data types that WebRTC and JavaScript use, such as Promises (used in WebRTC due to it's asynchronous nature) and the custom data types for containing and processing WebRTCs objects. I got around these issues through a variety of methods, ranging from storing custom objects as Strings (so they can be passed through Links) to writing loops that essentially turn an asynchronous JavaScript function into a blocking function (which Links assumes they are). The final API is almost entirely peer-to-peer, although in its current form it requires a nominal server for the initial signalling process. However, because none of the large video and audio streams are sent through the server it is very lightweight and should scale for a large number of callers quite easily.

1.3 Goal and Results

The overarching goal of this project was to design a video-calling API for Links, then to use that API to implement a video conferencing app similar to apps such as Gather.town and Zoom. Ideally this was to involve some sort of extension beyond purely imitating some of their features.

I fulfilled these project goals by:

- Preparing a custom virtual machine with Ubuntu and port forwarding to allow local web hosting
- Installing Links on the virtual machine
- Designing, implementing, and testing a Links API for video calling
- Researching other video conferencing apps like Gather.town to understand common complaints
- Designing and implementing a video conferencing app to address those complaints
- Exploring different ways of expanding on existing video calling apps
- Designing and implementing individual call modification for the API and application
- Extensive reliability testing including multiple browser and device configurations, over various time frames and methods of access

The finished project functions reliably, with good audio and video calling, a consistent interface for interacting with the website, and unique functionality in the form of independent call management. In addition, the project is easy to expand upon to explore different useful macros, as the systems underneath are generally “complete” in that they allow all possible states. However, the final version of the project is relatively removed from the initial inspiration of Gather.town, adopting a different user architecture by trading the continuous movement and positioning of Gather.town for a set collection of rooms with clear boundaries more similar to apps like Zoom and Discord.

1.3.1 Report Organization

This report starts in Chapter 2 with a Background section that investigates other popular apps in the video conferencing space to learn from their successes and issues. Gather.town is mainly focused on to give a potential model that can be worked towards, although other similar apps are covered as well. The Background also looks at the documentation available for Links and covers the features that may be relevant for a project such as this one, in addition to a brief look for similar projects. Finally, the JavaScript API relied on for the video calling in this project, WebRTC, is explored.

The report continues by covering the way that the project’s video conferencing app and API was actually built in Chapter 3, titled “Design and Implementation”. This looks at the designed systems overall model, the implementation of the various components, the collection of features that were added, and some of the major challenges of the project.

The final product is then evaluated in Chapter 4, with an assessment both in terms of the experience a user may have with the system and that of a programmer attempting to use the API or expand on the video conferencing app. This is reinforced by a selection of tests, mainly focusing on the reliability and scalability of the project. This is then extended somewhat in the conclusion, Chapter 5, where a possible extension is discussed.

Chapter 2

Background Information

2.1 Gather.town Overview

For those unfamiliar with the website used as the initial model, Gather.town [7] is a website which allows a user to move around a map and speak to other users in the same "room", automatically beginning voice/video calls with anyone in their close proximity. This is done through a video game-like interface seen in figure 2.1. It also includes other features such as areas where your call is limited to a select group of people (such as a table where you are only talking to other people at the table or a podium where everyone in the room can hear you), objects that allow you to load documents for others to see, and a space/avatar designer for users to customize their avatars and surroundings. Other quality of life features such as automatically muting and ending your video while tabbed off the screen, the implementation of visual emotes (through emojis in speech boxes on screen), and automatic echo-suppression in calls helps improve the platform further. The website contains multiple public gathering spaces for various topics where anyone is allowed, although it is also possible to create private or more specialized spaces that require a link. For these private and public spaces, Gather.town allows several tiers of "role" so that the space can be managed and built according to the creator's liking, including roles like "Moderator" for managing any community that forms.

The application itself uses peer-to-peer connection over WebRTC (discussed in the WebRTC section) making it fairly cheap to run as extensive servers are unnecessary even though Gather.town advertises large events which create a large amount of traffic. Despite the peer-to-peer structure, it still allows a large number of simultaneous calls (up to 9 video tracks simultaneously, and many more audio tracks), suggesting that the framework it runs calls over is not overly strenuous on a network. Gather.town is also fairly robust over several different browsers and offers guidance on at least Google Chrome, Firefox, and Safari.

In theory, the design of Gather.town allows workplaces and events to emulate the feel of an in-person meeting as small groups of people can easily break away from a larger meeting to talk about a certain topic, and it is then easy to move between these sub-groups. It is also easy and intuitive to walk around the room and see conversations that are currently on-going. To reinforce this, Gather.town suggests designating areas for different topics using the room designer, which lets people wander over to conversations that are likely to be of interest to them. Gather.town also seems to want to attract people to use it as more of a place to hang out with friends, adding several games in the objects menu such as poker and popular web party games like Skribble.io and One Night Werewolf.

In practice, many of these benefits over traditional calls are realized, with the ability to see other conversations and change between them quickly featured chiefly among them. Many of the advertised features also work according to my own admittedly limited testing, although this is supported by a lack of complaints of broken features (beyond the usual FAQ's about how to use certain aspects of the site) on online reviews as well as their generally positive tone [22]. In addition, an academic paper intended for a conference showcasing a new video-conferencing platform similar to Gather.town specifically noted it as a success story, calling Gather.town " ...sufficiently scalable, reliable and safe, supports multimodal communications including live participant video and audio, whiteboards, chat, etc. and runs in a browser, which makes it compatible with almost any platform" [18], although they did not feel that it fulfilled all their requirements given that they developed their own version.

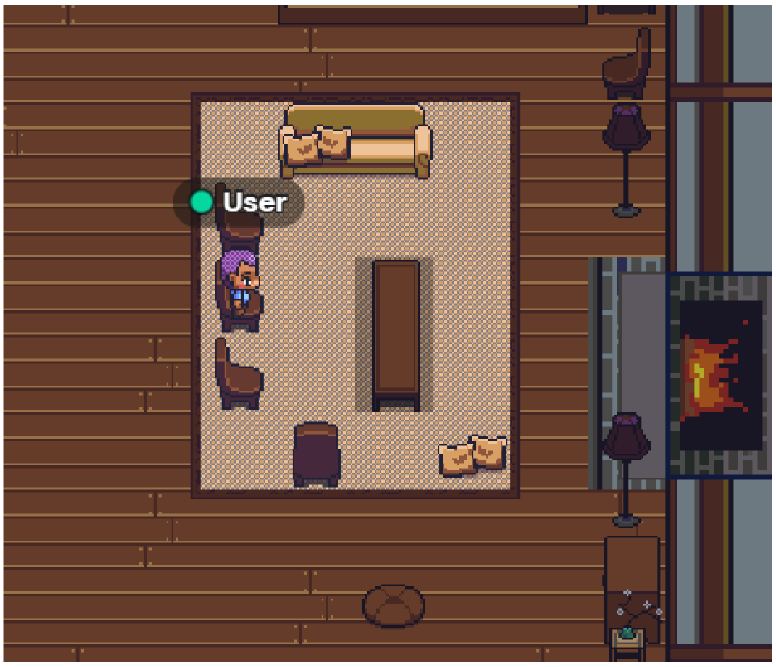

Figure 2.1: Gather.town Graphics

2.1.1 Gather.town Usage

In order to understand the strengths and weaknesses of a platform, it is also important to know how it is used and if it is expanding. For instance, Gather.town found a great deal of success in large events, with Gather.town alone holding events for over 10,000 companies and organizations [6], with similar competitors claiming smaller but still significant numbers. One example of such a conference can be seen in the Women in Machine Learning organization's annual "un-conference", which used Gather.town for the poster presentation portion of their event. The organizers found Gather.town to be "very joyful" and "very similar to in-person poster sessions and having real human interactions", although they encountered technical issues and found the platform to be somewhat expensive [17].

For educational purposes, a recent case study (although of only about 40 students) found that Gather.town was significantly better than more traditional video calling spaces, albeit still significantly worse than in person teaching [15]. Another smaller case study (16 students) involving two different university courses found a similar result [19], so it is therefore reasonable to conclude that Gather.town is effective in educational environments when compared to other online services. These successes and others like them are the main reason that the Gather.town style of app was focused on in this project.

It is harder to find evidence of the platform being used commonly for more casual gatherings (outside the conferencing and educational space) beyond a few expired Facebook game night groups from universities, which one can assume are likely to fade away even more once the pandemic is over. However, statistics on Gather.town usage [8] do suggest that its user base is increasing, with total monthly visits going from 2.6 million to about 4 million over the past 6 months. Although some of this spike can be attributed to the pandemic it also speaks to the advantages of the medium over traditional voice/video calls or chat rooms. Interestingly, one of the top referring pages to Gather.town is learn.ed.ac.uk, implying a considerable number of university-sponsored events.

2.1.2 Gather.town Issues

Although reviews and reports are generally positive, users of Gather.town also have complaints about the system [12]. For example, in Gather.town it is difficult to tell who can hear your conversation as it does not show a radius on the screen of where you can be heard and the usual way of telling if someone can hear you (their video feed appearing on your screen) does not happen if there are already a large number of people in the conversation, which can lead to privacy issues. Its binary system where a single pixel distance determines if a user is in a call or not also leads to multiple issues such as accidentally joining conversations if you walk too close, awkwardness about whether or not to join a conversation (as it provides no visual signals about how welcome new people are to a call), and a certain abruptness to entering a conversation as opposed to real life. In addition, some users less experienced with the control system (which uses the WASD set-up common in games) have found it difficult to use, and some have even reported nausea due to the way the screen moves.

Other user issues can be found in a more formal context in a paper from the Association of Internet Researchers [11]. The paper looks at the flaws of the platform compared to real life interaction such as lack of true eye contact inherent to video calling, the difficulty in "hovering" at the edge of a conversation as an instructor to see progress, and the stark unnatural feel of the breakout rooms, which have sharp borders and can suddenly close and eject everyone in them. Additionally, the team who wrote the paper also saw issues with lack of engagement with the platform, which is an issue as any vaguely social platform will require a critical mass of active users to function properly. Despite this, the authors did still have an overall positive experience with Gather.town.

2.1.3 Similar Applications

Although Gather.town is very successful, there are a large number of clones and competitors with similar but distinct products such as Sococo [9], Mozilla Hubs [10], and Remo [20], with all of these taking somewhat different approaches to the same idea. More distinct services like Zoom [21] and Discord [4] are also interesting to compare with Gather.town, as although there is not as clear an overlap in form, they have a similar function (online group video-calling).

For example, Sococo and Remo are both more abstract and accessible than Gather.town and are targeted more towards remote workplaces and events respectively. They implement features such as click to move to be more natural to those less familiar with technology and a static screen to avoid nausea. Additionally, Sococo implements simple dynamic avatars in order to quickly show what someone is doing (such as sharing their screen or in a video call) and "knocking" to ask permission to enter a conversation. Remo, with its focus on events, implements "business cards" to make it easy to connect with people met at their events and separate modes for presentations. However, implementing a simpler interface to be more accessible can come at the cost of making these platforms less immersive (especially as Sococo [9] and some like it have ditched most avatar customization beyond a name) which lessens the "feel" of an in-person event.

Zoom and Discord are even more abstract, using simple "Breakout Rooms" or "channels" respectively to allow users to subdivide into smaller conversations. These platforms are both widely used by the general public, with Zoom especially being practically inescapable (from personal experience) over the pandemic. Discord is specifically aimed at and marketed towards gaming communities, with its website saying it's for "...a school club, a gaming group, or a worldwide art community". It facilitates the growth of communities through long-term "servers" users can join with multiple "channels" which are always active. This design is very different than Zoom's, which has "meeting" time limits of 40 minutes for group calls on a basic plan and 24 hours for individual calls.

On the other hand, Mozilla Hubs takes almost the opposite approach with a full 3D VR-enabled setup with more complicated controls than Gather.town (using specific keyboard inputs) and spatial audio. It also includes Spoke, which is a custom room designer with a great deal of control. While this looks impressive and allows more flexibility than the more streamlined Sococo and Remo, it also comes at the cost of possibly alienating less technically-savvy people and increasing the barrier of entry.

Many of these platforms have found success due to the variety of features they offer, for instance Sococo has multiple large companies currently using (and paying) for their services such as JetBlue (A large American airline) [9]. Remo achieved similar successes with Northstar and several Chambers of Commerce's using their platform [20].

2.2 Links

2.2.1 Links Overview

Links is a functional programming language for web applications that uses a single source code to generate code for all tiers of a web application, as opposed to the more traditional method of using separate languages for database and client (often SQL and JavaScript/HTML/CSS respectively). Although Links is a relatively small academic language and therefore does not have the extensive set of libraries and API's of a more mature language, there are still some resources available. These are mostly in the form of a few example projects from the Links GitHub and documentation found on Links wiki [14].

2.2.2 Learning Links

As Links is a research language, it has relatively little documentation (compared to widely used commercial languages) to aid learning and programming within it. However, documentation for some of its features and systems does exist on the Links wiki [14], including the web server model mentioned earlier and client web page generation. The majority of functions aren't directly documented, but do have recorded type signatures in Links, which means that their function can be inferred through the use of their name, inputs, and outputs, with some experimentation to confirm required. Although this can slow down coding and lead to some non-optimal solutions it is still generally functional as programs can be tested quickly, typically by running the Links code on a port in a virtual machine and accessing that port via an IPv4 address. Individual functions may also be tested by command line for certain purposes.

Various example applications have been implemented in Links already that are useful for learning the language [13], which help to augment the documentation.

2.2.3 Relevant Features

Links has a web server model that will be used in this project, documentation for which may be found on the wiki [14], which allows Links to easily map URL's to Links functions and to call those functions using that mapping when the appropriate URL is accessed. Each function called using this routing table must then return a page to be displayed.

Links also allows the use of functions from JavaScript through the use of the Foreign Function Interface (FFI) which will be used in this project mainly to access WebRTC (see section 2.3). Although these functions are in JavaScript and therefore more difficult to access, they allow access to many functionalities that would otherwise be inaccessible in Links. In this project, JavaScript will be used initially to prototype video calling API before being replaced as much as possible by native Links code, leaving only the required portions in JavaScript. However, the Links Foreign Function interface lacks support for the asynchronous “Promises”, and Await statements used commonly in JavaScript, and does not have a direct way of dealing with the custom datatypes created by WebRTC. This creates integration problems that can be difficult to solve.

The Links Foreign Function Interface is a simple to use system where the programmer writes a function in JavaScript, then declares on the JavaScript side that the function will be used in Links. On the Links end the programmer effectively imports those functions in a module, where they can then be accessed by the rest of the program.

Other provided example code [5] details a way to have a server respond to new users by broadcasting a message to all current users, which will be useful for updating the clients with data required to interact with other clients, as well as a signalling method between individuals. This system of "processes" is explained fully in the original Links language paper [3], and is the system used for asynchronous programming within Links. The process system is composed of a collection of processes which each run at the same time as all other processes, each identified by a unique "process ID". The process IDs are used to send messages between the processes, which are received using a receive-case block that basically takes the top message out of the "Mailbox" that stores the received messages and parses it similar to switch statement. A diagram of this can be seen in Figure 2.2.

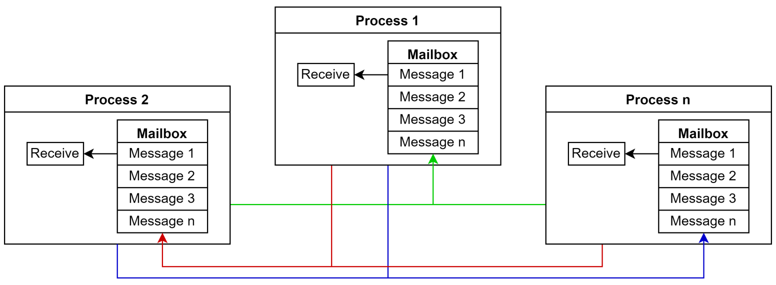

Figure 2.2: Processes with Mailbox Architecture

For example, the "Draggable List" example project [13] is intended to be a simple program with three lists that can be clicked and dragged to reorder. For each of these lists, the "main" process calls a function with the raw list as an input. This function then spawns a process that listens for messages (such as "MouseUp" and "MouseDown") and follows certain behaviors when they occur. Spawning the process returns a "process ID", which the function (on the main process) uses to send messages to the list-specific process as a reaction to events such as "l:onmouseup". As the main process calls this function for each list, each list has its own process denoted by a unique process ID to manage the events relevant to it. Used this way, processes allow the code to be more modular, with lower likelihood of conflict between different lists.

As of the start of this project Links had no video-calling programs, which is why part of this project is building one. However, that does not mean that the language has no previous work similar to a video-calling website to draw inspiration from. The most similar of these that I could find was a messaging app, which could be used as an example of a signalling system within Links. In addition, some of the example projects contain code that could be useful as an example for some implementations of a video calling app. One of these is an application which includes draggable lists (useful for implementing a form of mouse-based movement), and a different application with example code for keyboard-based movement.

2.3 WebRTC

WebRTC [1] (which stands for "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 Links Foreign Function Interface (described in 2.2.3).

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 users device, and the RTCPeerConnection which allows two different devices to make a peer-to-peer connection.

The MediaStream API allows "media capture" where in this case media is primarily the audio and video feed of a camera and microphone 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.

The API also has the functionality to request a list of all available devices, which can allow the user to choose which the application will use or allow the application to select one that best fits its parameters. The API also can control selection using its constraints, such as setting 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 were unplugged or a new one were to be plugged in. Once this event happens, it can then trigger the proper response.

The stream outputted 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.

The RTCPeerConnections 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 through a variety of different methods as WebRTC does not provide that functionality. However, as this project will be based around a website that all clients will be connected to, the simple solution is for the website to handle the signalling. 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.

Once the initial connection has occurred a client will store the connection as an object and an offer or accept behavior will be decided based on the configuration of the system (which is a design decision that needed to be made for this project). 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.

Additionally, even before a peer-to-peer connection is fully 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 as well to be ready to immediately receive data.

Chapter 3

Design and Implementation

3.1 Design Overview

3.1.1 User-Facing Design

As covered in the introduction, the basic requirements for the Video Calling app are the following:

- Multiple conversation to choose from
- Convenient way to change conversations
- Clear call boundaries
- Easy way to signal you are “Open to talk”
- Simple, static interface

To capture these requirements, a design that (to users) looks somewhat similar to apps like Discord was created, using fully separate ‘Rooms” to create multiple conversations with clear boundaries on a single server, and a mouse interface where users simply click the room they want to join. The requirement for easy non-verbal signalling was approached through the addition of an invite system, accessed through the same interface as swapping between rooms.

A mouse interface was chosen as it has been shown to be more intuitive than key-based interfaces for populations who are likely to both rely on video calling and more likely to have usability issues [2], and more efficient for selecting fields on a screen [16] (and by extension conversations). The architecture where the rooms are entirely separate was chosen to make call boundaries more clear, and make the interface simpler. This design may be seen in figure 3.3, although features of it will be explained in more detail later.

3.1.2 Functional Design

To implement this, along with the actual video calling within a “Room”, a functional design was created. The goal of the functional design was to minimize server load while still supporting the user-facing design. Therefore, a server-client model (shown in Figure 3.1) was designed to delegate how the responsibilities were shared between actors, and how these functional blocks would be organized.

This design contains a single Server process, which is responsible for tracking and coordinating the overall “master copy” of the state of the site with any number of Client processes. It must also be able to send and receive messages from Clients so that they may communicate. The design contains any number of Client processes, who hold a local state to display to their user through the UI, and are responsible for maintaining peer-to-peer calls with other Clients as necessary. These Clients must also be able to carry out requests given to them by the UI they are responsible for, including by sending requests to the server to change the overall state (for example switching rooms). Finally, the design allocates a UI process for each Client, that is responsible for displaying relevant information to the user and taking user commands to pass to the Client.

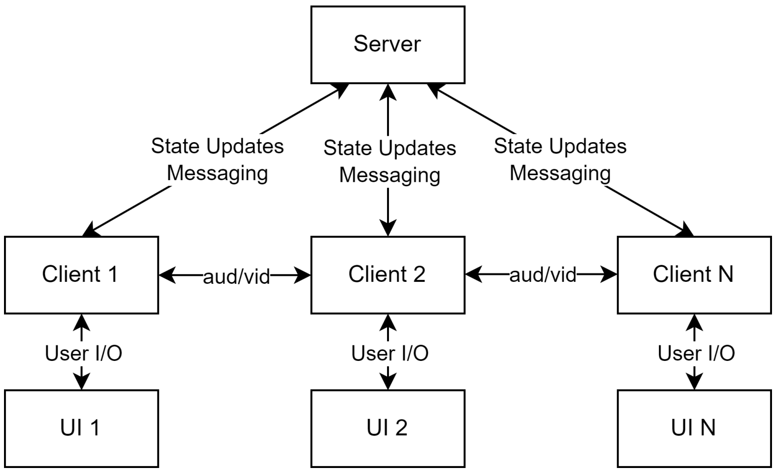

Figure 3.1: Server-Client Basic Model

This design has a few advantages, with the most important being that it requires only a relatively lightweight Server, because no large data streams like audio or video are passing through it. It also has a generally modular design, both as a single Client failing will have little impact on other Clients and as the number of calls can easily be expanded by simply adding more Clients. The centralized master state is also an advantage, as it provides an easily accessible default if a Client somehow has the wrong state.

However, this design also has some drawbacks. For example, the major drawback is that the decentralized calling model means that a traditional method of making conference calls less data-intensive client-side through multiplexing many audio feeds and only sending the final one is difficult here, as there is no central service to do the multiplexing. This can create a lower cap on the number of users than other methods might.

The centralized state model combined with the anonymity of the process mailbox also presents a potential security issue, as the Server cannot tell the source of a message. Because of this, if for example a Client could find some way of modifying the messages they were sending to the Server, they could modify the main state for everyone by sending move requests claiming to be from other Clients.

This model allows the addition of interesting features to expand on the functionality offered by popular video conferencing apps. A major expansion added to this design was the ability of each user to control the individual audio/video inputs/outputs of each of their conversations, which is entirely Client to Client and involves modifying the peer-to-peer calls between them. To make this more usable, macro-buttons for applying conditions to all calls at once (i.e. "Mute All") were added, using the existing infrastructure for communicating between UI and Client.

A core functionality not part of the initial requirements, Input Device Selection, was also necessary in order to allow each user to choose which audio and video devices they want to use. This is in order to avoid issues on multi-device systems. It required adding a "landing stage" to the Client where it requests this sort of information, but required no changes to the overall design.

3.1.3 Video-Conferencing API design

The API for video-calling was initially simple, designed to hold a collection of calls that could be accessed by a Client, and relying on the Links Foreign Function Interface to make the WebRTC functionality available. Beyond this, its functionality was generally built up naturally in response to needs of the other components of the project. For example, when adding call modification to the website an interface was needed to modify existing calls, so it was designed and created for this API.

3.2 User Interface Implementation

Practically any system that expects human interaction will require a functional User Interface for humans to interact with. As a major goal of the video conferencing app is to demonstrate the video-calling API working, the UI was kept minimalist to make future prototyping easier and avoid obfuscating features. Displaying the information in a more directly spatial manner (to mimic Gather.town in style as well as substance) would be a simple extension of the existing code, as it would be a purely aesthetic modification.

3.2.1 Landing Page and Device Selection

In order to properly communicate with other users, every user needs to have a name and unique ID, as well as valid input devices. Therefore, as this data is necessary before joining any calls, a "Landing Page" was set up which allows selection of devices and username, while a unique ID is chosen by the server on entry. The username entered is only used for the purposes of identification by other users, rather than by the Client, and therefore allowing duplicate names is not an issue on the functional side of things.

Although this may cause usability issues for users with the same names, it is assumed that a user will simply swap their name if the duplicates present too much of an issue.

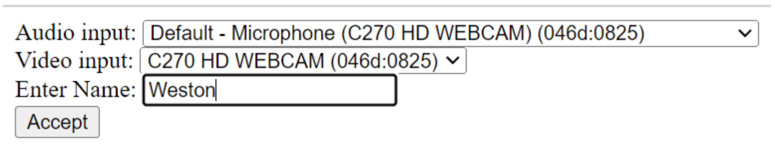
Audio input: Default - Microphone (C270 HD WEBCAM) (046d:0825)
Video input: C270 HD WEBCAM (046d:0825)
Enter Name: Weston

Figure 3.2: Landing Page

The device selection is handled in JavaScript, using WebRTC to access all of the relevant devices on the Users computer and using HTML to display this information in a drop-down menu. When the details are accepted, the currently-selected devices are then stored in JavaScript as part of a constraints object, which is passed to all peer-to-peer connections that are made. This does mean that all calls use the same devices as input, although this could be changed to allow call-specific device selection if a solution that does not overly impact the UI could be found

3.2.2 Main Page

Once the User has entered their information, they will be brought into the website proper as shown in Figure 3.3 (in a live call).

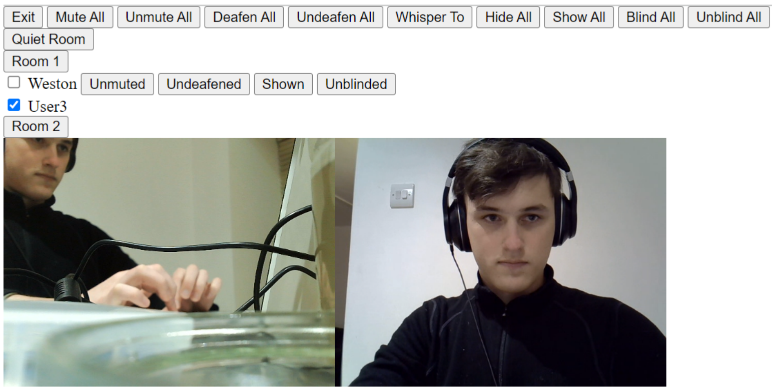

Figure 3.3: Website Main Page with 2 users

The user initially enters the “Quiet Room”, where no calls can be active, and may move to other rooms by clicking the relevant room button while they (the user) are selected. As is visible in figure 3.3, users can be selected using the checkbox next to their name. If a user other than you is selected when you click to change rooms, a request is sent to them that they may accept to follow your instruction. Any users who are in a room together automatically begin a peer-to-peer call between themselves which can then be modified by toggling the states that appear beside their name (which consist of “Unmuted”, “Undeafened”, “Shown”, and “Unblinded” by default). These states may also be manipulated through the macro buttons at the top of the page.

Almost all of the UI is written in HTML, but uses the same method as the Server-Client Communications system (described in 3.3) to pass signals to the Client when buttons are clicked or new options selected. This will be looked at in detail later but broadly allows the Client to receive the messages from the Process Mailbox messaging system described in the background.

3.3 Server Client Communications

Server Client Communication is done through the use of Process Mailboxes, which is a system where a process can be “sent” a message that will then be read out using a function in a first-in first-out manner. This is done using what is essentially a switch statement (although the Links API refers to it as a receive-case) which calls specific code depending on the type of message received. Therefore, the “heart” of both the Server and Clients is an infinite loop repeatedly reading the first message from the mailbox and reacting appropriately to it according to its type.

To send these messages, each client stores the “process ID” (basically the “address”) of the server, while the server contains a list of all of the different client process ID’s associated with the data about that Client. The UI then also stores the process ID of the Client connected to it so that messages may be passed between the UI process and the Client process.

The major challenge with using this system is first and foremost that it does not contain any way of recognizing if a message was not received by a mailbox, which means that if someone’s computer were to crash or the user was to exit the website by closing the page then the server would assume they were still there and continue to act as if they were. Although this does not crash the program it can still be mildly annoying as it is equivalent to a user leaving the browser open and leaving, with multiple “abandoned users” piling up over time. To completely avoid this, a liveness test would be required where the server queries all clients every so often to make sure there is a response. As this would add a large number of largely pointless messages, a simple “Exit” button was added for users to notify the server that they are leaving, which should be easily hooked up to server-side “not received” error catching when that functionality is implemented in this form of Links communications (as is planned). Although the “Exit” button does not catch crashes (as the error catching functionality will), it does avoid the accumulation of inactive clients under normal usage.

3.4 Client Implementation

The Client is responsible for updating the UI with the information received from the server, as well as passing requests from the user (through the buttons on the website) to either the server or stored state. To do this, the Client must store certain information to display for the user and to correctly control its peer-to-peer calls. Therefore, every Client must store the current state of the other users (i.e. their user ID’s, Names, and Locations), the actual connection objects for each current call (in JavaScript), and the connection rules for each other user (such as whether they should be muted). The Client also must be able to both send and receive messages to the server, and therefore stores the Servers process ID (see Server Client Communications). A general view of the Clients interactions can be seen in Figure 3.4.

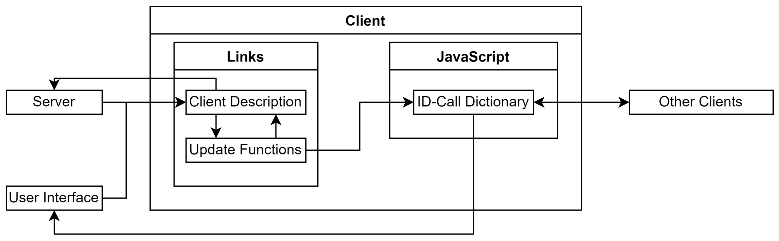

Figure 3.4: Client Process Diagram

3.4.1 Client State System

Each Client stores its own local state which is composed of a "ClientDescription" custom object for each other client containing the unique user ID (not process ID) of the associated user, the name of the user, and its location. This is then used to setup the display on the website page for the user to see, as well as to recognize when a call should be placed with a specific other Client.

This state is then updated when a relevant message is received from the Server or User Interface, with other supporting functions used to update the UI to the new state. Care must be taken to ensure that all Clients have the same state in order to avoid differences in expectations between Clients, however, constantly re-sending the same state (which could be fairly large) is expensive and redundant. Therefore, a system which typically only broadcasts the changes, but can send the full Server-state when necessary (such as when a new user needs a full state) was implemented.

This works by having the Server broadcast single "ClientDescription" objects whenever a user joins the server, changes rooms, or leaves, that the Client uses to overwrite the data it has stored on that Client in its state. However, the Client can also request a full state from the server if there is reason to think the Client State is compromised. These messages use different types so the Client receive-case loop (reading the Mailbox) can have different behavior depending on which is received.

3.4.2 Peer-to-peer Video Calling

In order to avoid routing large amount of information through the server, I implemented peer-to-peer video calling using JavaScript's WebRTC library, accessed using Link's Foreign Function Interface. However, to use this library multiple custom data types were required that Links can't easily handle, as well as a different type of asynchronous coding than the type Links allows: JavaScript's "listeners" and "Promises". Therefore, using this library required storing a significant fraction of the information in JavaScript instead of Links.

To do this each Client has its own JavaScript dictionary, using the unique ID of each other Client as the key for all the information about the call with that Client (as well as the call itself). This differs from the Links list objects used to store the rest of the

Client data mentioned earlier. The Client can pass any instructions for interacting with a specific call (such as beginning a new one or hanging up) to the JavaScript function by identifying the relevant call using its ID. The JavaScript ignores the Links code altogether for its output video/audio, writing video/audio streams directly to video objects in the HTML.

As described in the Background Chapter, each Client must go through certain states in order to complete a call through WebRTC. These states include an initialization, offer, and accept stage, and each require sending custom objects between the Clients (an overview of this is in Figure 3.5). The steps used for doing this may be found in the "WebRTC" Background section, with inter-client messaging used for the signalling process (described later in Server Message Types). However, the Links signalling used in this project (through processes and mailboxes) does not directly support the custom objects created, so the data must be serialized in JavaScript first by making use of JSON stringify to store the objects as Strings before sending them through the Server-Client messaging system to the corresponding Client.

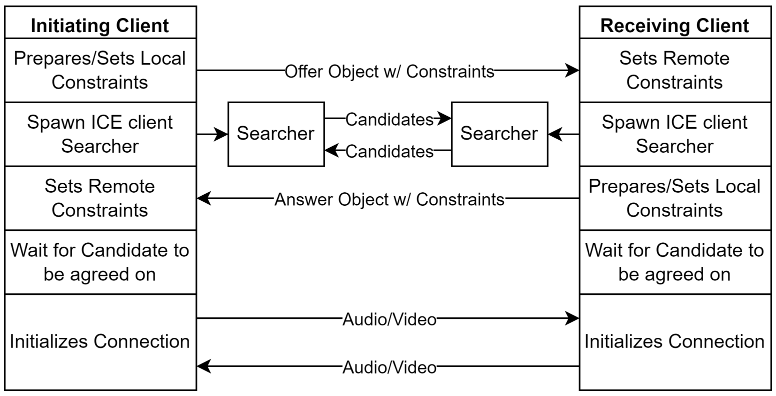
Initiating Client
Prepares/Sets Local Constraints
Spawn ICE client Searcher
Sets Remote Constraints
Wait for Candidate to be agreed on
Initializes Connection
Receiving Client
Sets Remote Constraints
Spawn ICE client Searcher
Prepares/Sets Local Constraints
Wait for Candidate to be agreed on
Initializes Connection
Offer Object w/ Constraints
Searcher
Candidates
Candidates
Searcher
Answer Object w/ Constraints
Audio/Video
Audio/Video

Figure 3.5: Call State Diagram

The mismatch between Links and JavaScript did cause some issues, for example when a call is starting up the two clients who wish to call must pass information between them, which is contained in a custom object Links does not support, or worse-still an asynchronous "Promise" object. Fixing this required manually coding what is essentially an "await" statement in Links by having the JavaScript functions write to a String variable with the relevant information while the Links code repeatedly reads that variable until something appears. This effectively makes the Links Foreign Function Interface function a "blocking" function that forces the program to wait for it to finish instead of simply returning a "Promise" that eventually will have the output. The asynchronous nature of WebRTC caused similar issues elsewhere, although these were handled in a similar way.

Another difficult problem introduced by the conversion between Links and JavaScript was timing issues, as the WebRTC library is typically used with listeners and JavaScript for signalling so the listeners can be added at the earliest possible time. However, this project is intended to rely on Links as much as possible and therefore Links signalling methods, which led to difficult-to-debug timing issues as the listeners had to be applied in reaction to calls from Links code. This was fixed by ensuring that the aspects of the call were added in an order which maximizes the difference between a listener being added to a JavaScript object and the possibility of that listener being required, which would not have been as large an issue using a purely JavaScript signalling system where they could be directly linked.

3.4.3 Connection Rules

Each of these connections must be managed so that an individual user may control the independent audio/video input/output per connection. This is done using a hybrid system relying on both Links and JavaScript, using Links to store the state of the connection rules with JavaScript functions to apply those rules to the connections themselves.

The Links portion of this is done by associating a "AudioVideoMods" object with each Client by wrapping a ClientDescription object (mentioned earlier) and an AudioVideoMods object under a new datatype, "UserClientRecord", and storing one of them for every Client. The AudioVideoMods is composed of a set of 4 booleans representing whether the audio/video inputs/outputs are activated or deactivated for each connection. The UI buttons to the right of each Clients name are used to control these states directly, as well as the macro buttons at the top of the page for changing many of them at once. This can then interact with other systems such as in the case of the "Whisper To" button which "Deafens" all Clients other than the selected ones as it interacts with the user selection system to provide functionality uncommon in calling apps.

It is important to note that each connection is controlled on both sides, so information only flows if both parties agree that it should. For example, if user A has disabled their audio input from user B, but user B has not disabled their audio output to user A, audio should not be sent from user B to A even though B's rules claim that it should. This is demonstrated in Figure 3.6, which shows that both users must both agree on each type of connection.

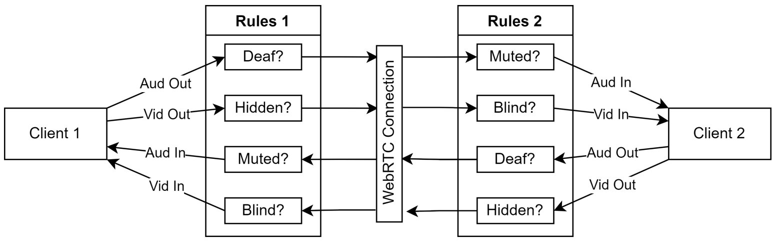

Figure 3.6: Connection Rules Diagram

The JavaScript functions which apply these rules to each connection are more direct, with functions that use the unique ID of the connection to access the WebRTC objects and enable or disable the channels as necessary. This is done whenever a state is changed and is persistent, so for example another user could not force someone to unmute them by leaving the room and rejoining. The approach in this case was mainly taken as protection against malicious users, and could be further improved by making the default ruleset changeable by a user, although this has not yet been implemented.

3.4.4 Client Message Types

The Client generally receives messages from one of two sources, either the UI or the Server. These can be viewed differently, as the former is a direct response to the controlling users input while the latter is generally a response to other users input. The messages from the UI are either the initialization procedure or mid-function updates, and the messages from the Server are either modifications of existing states or supporting communications.

The initialization procedure messages pass the name to the Server and retrieve the generated ID. To make things simpler, the selected devices are read directly from the HTML by JavaScript as opposed to going through the messaging system (as that is how they are written to as well). Once all initialization procedure messages have been sent, the Client replaces the Landing Page with the Main Page so that the user may interact with others.

UI mid-function updates are things like "Migrate" which occurs when a user clicks on one of the room buttons. These pass user-caused events to the Client for handling, which can vary in response from something simple like muting a single connection (toggling a boolean and calling an update function) to Migrate, which collects data on everyone who was selected, then passes messages to all of them requesting a room change.

Server state modifications are when the Server passes a "StateUpdate" request which usually consists of a single users new information. The Client then changes the relevant ClientDescription to the new information, sets the displayed state (through an update function) as well as passing all necessary commands for calls to be either begun or ended. These also include messages like adding a new Client to the Server or the removal of a Client.

Support Communications includes things like requests for others to go to a separate room (as it does not change the state but just displays a request) and the messages associated with the video calling process. These messages are again handled by specific routines that always occur when a message of the specific type is received, allowing the compartmentalization of certain behaviors such as the states of the video call initialization.

3.5 Server Implementation

The Server is responsible for tracking the master state and passing necessary state updates to Clients, on-boarding new Clients, and passing messages between clients. It is also responsible for ensuring that all Clients are uniquely distinguishable from one another to allow correct functioning of updates and state-tracking. All functionality in the Server is in response to specific messages the Server receives (similar to the Client) which all come from Clients. This uses the Process Mailbox structure described earlier. A basic overview of the server can be seen in Figure 3.7.

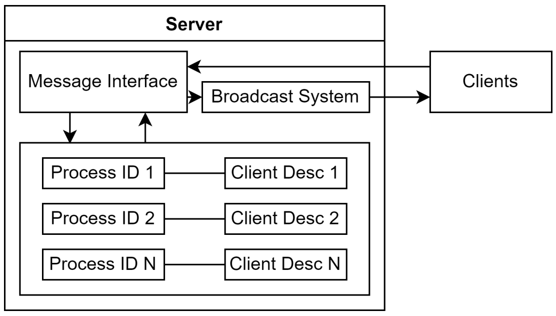

Figure 3.7: Server Process Diagram

3.5.1 Server ID generation

In order to distinguish between Clients, each must have unique identifying information in the form of (in this case) a unique ID number. This is generated by the Server sequentially as a part of the on-boarding procedure before the user may interact with other users. The generated ID is then communicated back to the Client, which stores it as part of its own information for use with error checking (for example catching if a message was not intended for it). The server also stores this ID associated with other Client-provided information covered earlier such as the provided name, the Clients process ID, and the current location of that Client.

Originally, this ID generation was done on the Client-side by choosing a random 6 digit number, however despite being simpler to implement this runs into the error of ID-collision, albeit very rarely. As the new form works sequentially, this is impossible in the current use-case. To avoid possible security issues or impersonation problems, the Server's stored IDs may not be changed by any message from the Clients and are permanently associated with the Client after the initial generation.

3.5.2 Server State System

As the Server only must track the basic ClientDescription (the name, ID, and location) of each client and the process ID associated with it, the "State" is simpler than in the Client. This is good, as the State must be processed almost every time any message is received by the Server. For example, if Client A wanted to send a message to Client B, Client A would send a "Send" message to the Server coupled with the ID of Client B and the message. The Server would then have to filter the State for the data corresponding to the provided ID before passing the message along. As the Server must do this for every message, it will be processing its state much more often than a Client and therefore that state must be lightweight. Partly for that reason, the Server stores no information about Connection Rules (as the client does), limiting the state to only what is strictly necessary. Ideally, the Server would store its state as a dictionary rather than an unsorted list, reducing lookup time from $O(n)$ to $O(1)$, however that is currently unnecessary as other bottlenecks will be hit much earlier (such as the data limitation on video sent between Clients).

This State also must be changed depending on the messages the Server receives. For example if a new user joins the associated Client will send a message to the Server requesting to join, which the Server must respond to by adding that Clients information to the State and generating an ID (in addition to updating all other clients with the new State). Stored Clients are also mutable, as the individual ClientDescription (specifically the room number) must be modified if the Client wishes to change rooms.

3.5.3 Server Message Types

The messages the Server expects can broadly be classified into a few types; on-boarding and initialization messages, state change messages, and inter-Client messages. Each of these groups are generally associated with separate functionality and have different requirements.

Inter-Client messages are the simplest and the most common, as they are messages that the server must receive from one client and send to another (or multiple others), and do not require modification of the internal state. These are generally done by searching the internal state by the ID's of the recipients for the associated process ID's, then broadcasting the requested message to them. These messages are used for sending requests between clients such as a request to join a different room or to do the signalling associated with initializing peer-to-peer calling.

Initialization and on-boarding messages are the subset of messages that are required for setting up a new Client, and involve adding to the internal state, as well as processing the internal state for sending. This is necessary as the new client will require a copy of the state in order to, for example, know what other users are on call and where they are. These also make use of ID generation as mentioned earlier.

Finally, state change messages are messages such as "Room Change" that require the server to modify it's internal state without adding new Clients. These messages generally require filtering the list for the Clients whose data must be changed, and broadcasting a message containing the change to the rest of the Clients.

Chapter 4

Evaluation

4.1 User Experience

4.1.1 Experience Overview

The video calling website is now fully functional, including multiple rooms with distinct calls within each, a simple way of moving between rooms, and an interface that allows complete control over data both sent and received by a user. The last of these allows the formation of smaller, ad hoc calls between a subset of the users in a room. Even asymmetrical calls are enabled, for example a call in which one user sends only audio but their call-mates send audio and video. This control over what you both receive and send extends to device selection before any call takes place, so you could specifically choose which individual camera and microphone you wish to stream from, as well as selecting the name you wish to appear by on the website proper.

All data rules on calls is perpetual, so if a user for example mutes someone who they don't want to hear from, then happen to be in a different room with them later the mute will remain between calls. This is useful as malicious users cannot "wipe" others settings on their connections by leaving the call and rejoining.

Generally speaking the website is robust, allowing a considerable number of people to connect without issue, and continuing to function even if left running for extended periods. The Exit functionality (see Server Client Communication) allows the server to continue functioning without issue with an arbitrarily high number of people entering and leaving the server, as their data is completely removed. Due to its reliance on WebRTC, the project is most reliable on Chrome, but is mostly functional on a variety of other browsers such as Firefox (albeit with some slight glitches). While the project was originally designed for a personal computer with attached webcam and microphone, it is somewhat device-agnostic in that it also works for phones in mixed computer-phone calls (assuming the phone has a suitable browser).

The most major issue a user may encounter is that a Client may crash if the user swaps between calls fast enough that the asynchronous JavaScript code is still in the process of hanging up when they attempt to begin a new call with the same ID. Although this crashes the Client, it does not affect the Server so the other users will be unaffected. This can likely be fixed by applying the pseudo-blocking solution used in call initialization, although this has not been done at time of writing.

The other common issue found is when the site is first accessed in a new browser online (using https), when the device selection simply displays “microphone 1” or “camera 1” instead of the full name of the device. This is caused by the program not having permission to access information about those devices until it actually tries to use them directly, and the problem disappears after the user allows access to microphone and camera the first time. Although the selection process still works correctly, and the selected devices work once the main site is entered it could cause an issue for people with multiple cameras/microphones, as they would have no idea which is selected. This could be fixed by forcing a permissions check on all audio and video devices before entering the page.

In a test call with a few new users, the basics of the user interface was picked up fairly quickly, with the new users quickly understanding how to swap between rooms and modify call rules on an individual basis. However, some of the interactions between systems required explanation such as the functioning of the “Whisper to” button mentioned earlier that applies a certain rule to everyone selected and another rule to everyone else. Systems where results weren’t immediately apparent to the user, such as the join request system, also took slightly longer to understand as they required interaction with at least one other user to see the results.

4.1.2 Gather.town Comparison

When compared against its initial inspiration, Gather.town, this project trades away some of the immersion offered by the video game-like format for a more abstract spatial metaphor with clear rules on who can see and hear you. This minimalist design allows the user more control over the actual calls they are in, which makes the smaller “sub-conversations” more flexible through the use of the rules available. This allows the website to mimic real life conversations and interactions in a way that Gather.town’s interface is unable to do, even with just the existing macros and individual modifications.

A hypothetical example of this project better modeling the intricacy of real life events can be seen in the simple situation of the platform being used to deliver a lecture to a group of students. The user who is delivering the talk will use the “Mute all” button to avoid disruption but they may continue keeping an eye on the still-active cameras for hands being raised so that they could selectively unmute to hear questions. Each individual student would then deafen and mute most people other than the lecturer, but leave the connections to perhaps a few friends untouched so that no one could hear them but their friends, and they could only hear the lecturer and the friends. Assuming others in the lecture follow suit this effectively creates “sub-conversations” within the larger conversation akin to whispering to a friend in a lecture hall. To mimic this in Gather.town you would require a specially prepared set of rooms/tables for each sub-conversation and a “spotlight tile” allowing the presenter to speak to everyone regardless of their location. Even with the preparation, the Gather.town equivalent is much less efficient at splitting or shrinking subgroups (as it would require yet more tables and moving around the screen), and is completely unable of creating complicated asymmetrical subgroups (such as a student being part of multiple subgroups simultaneously). However, using this project the situation can be created quickly by everyone joining a room and each clicking a few buttons to unmute/undeafen their chosen few.

An example of this sort of complex conversation can be seen in Figure 4.1 where a new lecturer, User 1, gives a lecture to a group of students, Users 2-6. However as the lecturer is new, they have an advisor, User 7, listening to the lecture and giving them advice that cannot be heard by anyone else. The students wish to talk among themselves, asking questions of each other and making comments, but do not want to interrupt the lecture so set their rules so they can only be heard (or seen) by who they wish to be. This example system, if not impossible to set up in other popular applications, would at least be very impractical and require extensive fore-planning.

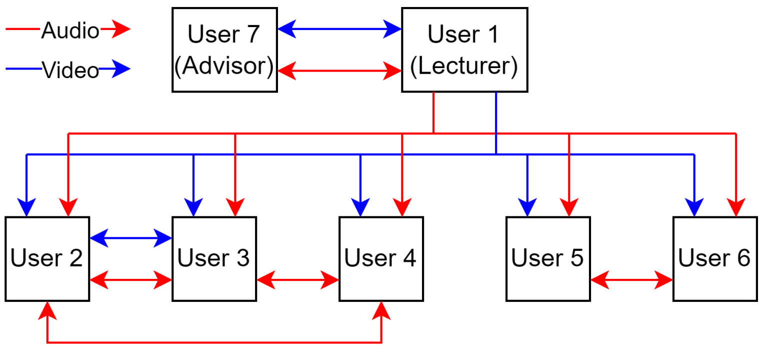

Figure 4.1: Example Complex Call Diagram

After the lecture, the teacher could then use the “Request Room Change” interface to ask groups of students and/or tutors to join specific rooms to go over the material, without having to go themselves. Again, while in Gather.town this would require something like a long bulletin where people find the tables assigned them and go to them, in this project, the Lecturer can select the students they want in a group, and click on the room they should go to in order to send out automated requests.

4.2 Technical Results

The website is written in a mixture of Links, HTML, and JavaScript, with approximately 80% being Links, and most of the rest being JavaScript functions to expose WebRTC functionality to Links. Theoretically, this Links-WebRTC functionality should be possible to generalize for usage in other Links applications.

The way that the application is designed allows for additional features such as dynamic rooms (so that users may add new rooms should all rooms be full), or the expansion of the existing room request functionality to a complete chat-box using almost purely existing functionality. No issues have been found with load on the server end during stress testing, implying that the system can support large Clint populations with many rooms and calls.

The completely exposed control over the aspects of a call and the convenient interface for modifying these calls allow a designer to freely test new ideas, for example the "Whisper to" macro button implemented which makes it so that only a selected group can hear anything the user says. However, the current project does not have ubiquitous type signatures on functions (although some exist), which would be somewhat time consuming to implement as it requires declaring all possible messages that could be sent between processes (when process ID's are passed as arguments to functions). This makes some errors fairly opaque for future programmers, which would make modification more difficult.

4.3 Testing

To attempt to discover any bugs a variety of tests were performed, varying from single-computer multi-browser tests on a local network to testing with multiple users over the internet. This also served to demonstrate possible future bottlenecks.

Generally features were initially tested using Single-Device Local Testing, which was essentially running the code on a port of a computer than accessing the website through that port multiple times on the same computer. However, most features were eventually tested through tests such as Multi-Device Non-Local testing, which involved accessing the website completely over the internet from multiple sources.

4.3.1 Local Testing

The vast majority of testing was done purely through local internet, as running tests on the open internet is slower and requires more setup, due to video and audio device access requiring an https site.

4.3.1.1 Single-Device Local Testing

In order to approximate an upper bound on the number of concurrent calls, a simple test was used. Essentially, the site was hosted locally, then tabs are connected to it until the system begins to lag. This will eventually happen as every single call requires a port going through ICE servers to make the peer-to-peer connection work, even if both of those peers are the same computer. Each tab in this test will require a number ports equal to one less than the number of tabs already existing, meaning that the required ports on a single computer for a test of n tabs is $n \times (n-1)$. In a call where only one user is on each device, the same one port per other user in the call is necessary. However, as the computer only has to support one tab, only $n-1$ ports are required. This means that on a single computer if 4 tabs may be used concurrently that implies this sample laptop can handle $(4) \times (4-1) = 12$ concurrent ports, which is a maximum call size of $12+1 = 13$ people.

The test described was repeated several times: very slight delays began with 5 active users, and the system entirely broke down at 6 active users. This implies that the bottleneck on single-computer ports is between $(5) \times (5-1) = 20$ and $(6) \times (6-1) = 30$ users in a single room. This could possibly be increased by dropping resolution of the video being sent, but the current resolution of 320×240 is a good minimum resolution for video calls so going below that value is unlikely in practical video-calling use cases. This could also be improved by multiplexing using a video server, but this would limit the individual call control currently offered and would likely require more information be passed through the server.

In addition, Single-Device Local Testing was the initial test used to ensure that all the features and website worked, as it is the most convenient way of testing if, for example, connection rules are properly maintained after switching rooms.

4.3.1.2 Multi-Device Local Testing

To ensure the project continued to work when not solely on a single computer (essentially to make sure that different devices with different webcams/resolutions still worked), the project was tested a few times by hosting the site on a local computer, then attaching to the port directly with other devices using the IP address and port number. This was only done a few times as it is unlikely to catch new bugs, but did show that 5 different devices could connect and function well on the same call.

4.3.1.3 Mobile-Device Local Testing

The project was also tested on phones, using the local hosting to expose a port with the project and then connecting manually using the IP address. This worked with some phone-based browsers although others had issues with sending video over multiple ports at once, specifically Safari on an iPhone (which seems to limit audio/video feed to one output at a time). However, the single video call that did go through seemed to work, and the project worked without issue on android phones using other browsers.

4.3.1.4 Multi-Input Testing

In order to test the Device selection process, the project was locally hosted, and a computer was hooked up to multiple different USB camera devices. This computer then joined the website each with a different camera selected and joined a call with the other example cameras. This worked and caused no issues on Google Chrome, however on Firefox multiple cameras to select are displayed even if only one is connected (likely due to WebRTC not fully working with Firefox), this is not a fatal error however, as the selected device does still work and allows the user to join calls and communicate.

4.3.1.5 Time Based Testing

To make sure that the Server did not accumulate errors or excess data over time and slow down, a local test was held with three users. The project was again hosted on a local device and users were prepared on two devices, with two users on one device and one on the other. The calls were all muted and deafened so the the computers could still be comfortably used and were then left running for three hours (with the devices used consistently enough they did not enter “sleep mode” or turn off). After the three hours, the calls were then tested and functioned as expected, with clear audio and video being used. This implies that a call can last at least as long as other services advertise for free (often up to 3-4 hours maximum) and possibly much longer.

After this test, the server was left running for another 16 hours before being tested again with new Clients. Again, the system worked entirely as expected, with multiple new Clients being capable of joining the server and beginning group calls with one another.

4.3.2 Non-Local Testing

This test involved having multiple people in different places connect to the website (while hosted as an https site) as part of a demonstration and presentation. Although this showed that it functioned under the right circumstances it also exposed an issue where the project was unable to access a camera if another process (such as Zoom) was already accessing it. This led to a variety of issues with calls, as no video stream was sent and errors thrown within the program due to the WebRTC conflicts went unhandled. Other than that, it did show that working over an https server to different places worked similarly to local hosting, and that once successfully accessed the website worked as expected.

The issue that caused these errors was then handled through the addition of device selection, which gives people the option of selecting an alternate device before joining the server.

With the fix in place, the project was used to host a short call (approximately 15 minutes) with four other students with experience programming attempting to break the website (with some small issues found and fixed), a two hour call with two people, and a four hour DnD session of four people which made use of several of the macros such as “Whisper to”. The latter experiments went smoothly, showing the project functions reasonably well under these common use cases.

Chapter 5

Conclusions

5.1 Goals

This project had two major goals: to build an API for video calling in Links and to apply that API to build a video conferencing app which improved on the popular examples in some way. While somewhat distinct, these are necessarily linked as the functionality of the API constrains the possible features of the product built using it.

5.1.1 Links Video API

In order for the video calling API to be usable, it required a few core features. At the most basic level, it had to allow two users to send audio and video data to one another. However, it also had to be able to distinguish between existing calls so that multiple calls could be held simultaneously, end calls that were over, and have a way of outputting the data to the user.

This API was to depend on peer-to-peer calling as much as possible to decentralize the processing load on the video server. For convenience, I used the Links Foreign Function Interface to access JavaScripts WebRTC functionality.

5.1.2 Video Conferencing App

Making a video conferencing app that improves upon the popular video calling options is more open-ended, as there are various specifications this could follow. What apps of this type require to be useful is inherently subjective, so I took the approach of inspecting a specific app, Gather.town, then attempting to identify the key benefits of its design that sets it apart from many other “traditional” video calling apps. I believe the largest of these differences is the flexibility of the call system, which allows conversations to naturally break into smaller groups then combine again later, as it is a major common feature that Gather.town and other similar app share. This is the feature that I focus on when referring to these video-calling apps as “Dynamic”. Key to this is the ability for each user to be able to choose to enter a variety of different calls and change to a different call later. Therefore, for the purposes of this project, a “Dynamic

Video Conferencing App" was viewed as a video-calling website where a large number of users could take part in a selection of different conversations, and change between them easily.

Improving on apps like Gather.town is also quite subjective, so I went through the common complaints in the reviews of Gather.town as a starting point. This was useful, as it allowed me to approach the general goal of "improving on popular video calling apps" as addressing key/common issues users had with Gather.town, namely its unclear conversation boundaries, its propensity to cause vertigo in some users, and the awkwardness of joining a new conversation within the website.

5.2 Evaluation

5.2.1 Links Video API

The video calling API is functional and generally robust, despite the challenges of integrating asynchronous JavaScript functionality with Links, which has no support for dealing with Promises for example. The API meets all of the stated goals by allowing peer-to-peer calling of multiple user simultaneously, and writing the data to video objects in HTML. It is also generally reliable in common use cases, as seen in the variety of tests run on the implemented platform. Currently though, that reliability does depend on the correct usage of the API, as many of the functions require immediately calling a separate function to block until the first returns results. For the API to be generalized this should be automated for easy usage, as otherwise an unwary programmer could hit many timing issues such as the call-switching bug currently in the project. This will be covered more in the Extension section.

In addition to meeting the basic criteria for a functioning video API, the API also includes extra features. The first of these is the implementation of Device Selection for the user, which searches the computer for the devices that could be used for audio and video calls and allows the user to select which they intend to use. This is important as people with multiple cameras or microphones may wish to use specific ones for calling. This also avoids the most major issue with the API, which is that the API is unable to access devices that are already being used for other processes. That is an issue innate to the browser and outside code used though, so cannot be addressed without a workaround such as using other devices. The solution implemented here also has some issues, mainly that the project cannot access the names of the attached devices before permissions are given. However, after the project is first used, the error goes away as the site then has permissions. The API also allows all audio and video aspects of a call to be toggled, for example limiting a call to audio only or a call where one user sends audio and video while the other sends only audio.

5.2.2 Video Conferencing App

The video conferencing app meets the basic specifications provided for it, with a customizable number of rooms available for users to enter for various calls to mimic the ad hoc conversations of Gather.town. It uses an interface more similar to apps like

Discord or Zoom, with the main website using simple buttons separating each room and a generally simplistic interface using only mouse-clicks to make the site more accessible. This simplified interface also directly addresses the Gather.towns issue of not knowing who is able to hear you by providing a simple list of audio/video connections. The issue of awkwardness in joining conversations is somewhat addressed with the implementation of a simple invitation system, where you can invite anyone to any room with two clicks. Ideally this would be further addressed by adding an "open to new users" flag on rooms that can be set by people within them, although this has not yet been added.

The app was also expanded past the original goals, using a "Landing Page" for device selection so that users may select any video and audio input they'd like before entering any calls. In addition, a unique (to my knowledge) connection rule functionality was added, where each user has control of what data they send and receive to each individual call they are on. This may be viewed as building on the common "Mute" function seen on most calling programs which disables audio input from one other caller. With this functionality, it is possible to make any individual user unable to see and/or hear you, and yourself unable to see and/or hear them. This functionality allows significantly greater flexibility in the creation of individual calls Gather.town by giving ultimate individual control to the user, and allows complex asymmetrical conversations to occur.

Although there were a variety of additional features and modifications that were not added due to time constraints, the most important thing that is missing is a more graphical/spatial interface (in the vein of Sococo [9]) than the current text-based solution as it would aid immersion for some users. However, this would be a purely aesthetic modification, as all the functionality required for a simple one is already implemented. Additionally, certain bugs remain in the project such as the crashes when calls are quickly changed between and the lack of device specification in the device selection when the project is first accessed. Although solutions for these are known, they have not yet been implemented.

5.3 Challenges

This project involved various challenges, but first and foremost among them was the use of Links, as it is a small language with varied levels of documentation and an uncommon structure. Although this structure had advantages, the alternate approach made the project difficult to begin as simply getting used to the language took a significant amount of time. When actually writing code, Links can be difficult to debug, especially when compared to larger languages with extensive debuggers and countless tools and examples already available. This did get easier with time though, especially once the structure and quirks of the language became more natural. In addition, while Links has a Foreign Function Interface that works for JavaScript, it lacks support for Promises (and generally the type of asynchronous programming used in JavaScript), which required a significant amount of effort to work around due to the asynchronous nature of WebRTC. Lack of support for other custom datatypes was also an issue and required serializing WebRTC's data before passing it through Links, although dealing with highly-specific custom datatypes is perhaps something that should not be expected of a language.

Another major challenge was attempting to improve upon apps like Gather.town (a multi-million dollar, long term project) within about 20 weeks of part-time work. The approach that was taken for this challenge was to focus on very specific things (i.e. call control and ease of use), and rely on system design to address issues as opposed to the high levels of polish that time and money can afford.

Finally, choosing the basic design of the project was somewhat complicated, as different designs allow for different functionality. For example, the design chosen with direct peer-to-peer connections overseen by a central server allows a large amount of control over individual channels by a user, and allows the server to be very small as it does not have to manage any actual audio or video. However this comes at the cost of the sheer capacity of users in a single call that other designs may allow as it requires every individual user in a call has a separate port for each other user. As this had to be chosen before any testing or true work could be done, priorities had to be chosen before any feedback was available.

5.4 MIP2 Extension

As this project is only the first part of a Masters project, there is another equivalent amount of time to extend the project. This extension will focus on generalizing and expanding the Video Calling API to be capable of the required functionality for any program where calling is necessary. Initially this will involve wrapping the existing calling and call modification code so that each is a distinct blocking function and the intricacies of the inner system can be ignored by a potential future programmer.

For example, instead of having to understand how the initialization and eventual calling system works (seen in figure 3.5) a programmer should be able to simply call two functions on each client to begin a video call between them. This should also take constraints such as where the video and audio will be written to, which must be as broad and modifiable as possible.

To further generalize the system, all call modifications and functions available to WebRTC in JavaScript, such as variable and asymmetric resolution, should be wrapped and made available through easy to use Links functions. Additionally, live screen sharing options enabled by WebRTC will be added, allowing the development of more complete video conferencing apps. This may include file sharing as well, as WebRTC allows arbitrary data to be sent over its service.

Finally all functions that are part of this API must be well documented, with exact inputs, responses, and errors to make sure that it is as easy to use and debug as possible.

This can then be evaluated by replacing the API used in this project by the new version, which should be simple if the generalization was done well. The more telling evaluation would be modifying the code of the Links video-calling app created by another student this year, which relies on a different base code structure and a different style of video calling website, as if that project can be easily modified to use a new API it suggests the API is flexible. The API should then be used to build a few simple apps of different types to show the different ways it can be used, such as a mixed video-calling and pure audio app.

Bibliography

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

[2] Jyh-Rong Chou and Shih-Wen Hsiao. “A usability study on human–computer interface for middle-aged learners”. In: *Computers in Human Behavior* 23.4 (2007), pp. 2040–2063. ISSN: 0747-5632. DOI: https://doi.org/10.1016/j.chb.2006.02.011. URL: https://www.sciencedirect.com/science/article/pii/S0747563206000124.

[3] 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.

[4] *Discord: Your place to talk and hang out*. URL: https://discord.com/.

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

[6] *Gather for conferences*. URL: https://www.gather.town/conferences.

[7] *Gather.town*. URL: https://gather.town/app.

[8] *Gather.town Traffic Analytics amp; Market Share*. URL: https://www.similarweb.com/website/gather.town/.

[9] *Home - Sococo: Online Workplace for distributed teams*. URL: https://www.sococo.com/.

[10] *Hubs by Mozilla*. URL: https://hubs.mozilla.com/.

[11] Naomi Jane Jacobs and Joseph Lindley. “ROOM FOR IMPROVEMENT IN THE VIDEO CONFERENCING ‘SPACE’”. In: *AoIR Selected Papers of Internet Research* 2021 (Sept. 2021). DOI: 10.5210/spir.v2021i0.12188. URL: https://journals.uic.edu/ojs/index.php/spir/article/view/12188.

[12] *Launch HN: Gather.town (YC S19) – spatial video chat for remote teams: Hacker news*. URL: https://news.ycombinator.com/item?id=25039370.

[13] *Links*. URL: https://links-lang.org/.

[14] Links-Lang. *Wiki · Links-Lang/Links Wiki*. URL: https://github.com/links-lang/links/wiki.

[15] Colin McClure and Paul Williams. “Gather.town: An opportunity for self-paced learning in a synchronous, distance-learning environment”. In: *Compass: Journal of Learning and Teaching* 14.2 (2021). ISSN: 2044-0081. DOI: 10.21100/

compass.v14i2.1232. URL: https://journals.gre.ac.uk/index.php/compass/article/view/1232.

[16] A.F. Norcio and J. Stanley. “Adaptive human-computer interfaces: a literature survey and perspective”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 19.2 (1989), pp. 399–408. DOI: 10.1109/21.31042.

[17] Mandana Samiei et al. “Convening during COVID-19: Lessons learnt from organizing virtual workshops in 2020”. In: *CoRR* abs/2012.01191 (2020). arXiv: 2012.01191. URL: https://arxiv.org/abs/2012.01191.

[18] Alexei V. Samsonovich and Arthur A. Chubarov. “Virtual Convention Center: A Socially Emotional Online/VR Conference Platform”. In: *Brain-Inspired Cognitive Architectures for Artificial Intelligence: BICA*AI 2020*. Ed. by Alexei V. Samsonovich, Ricardo R. Gudwin, and Alexandre da Silva Simões. Cham: Springer International Publishing, 2021, pp. 435–445. ISBN: 978-3-030-65596-9.

[19] Bernhard Standl, Thomas Kühn, and Nadine Schlomske-Bodenstein. “Student-Collaboration in Online Computer Science Courses – An Explorative Case Study”. In: 11 (Oct. 2021), pp. 87–104. DOI: 10.3991/ijep.v11i5.22413.

[20] *The Virtual Experience Platform*. URL: https://remo.co/.

[21] *Video conferencing, cloud phone, webinars, chat, virtual events: Zoom*. URL: https://zoom.us/.

[22] Phillip Wang et al. *Gather town reviews*. Mar. 2022. URL: https://www.producthunt.com/posts/gather-town/reviews.