Background Chapter

Lewis Raeburn

November 5, 2021

In this chapter, I will discuss the key concepts and software technology involved in this project - implementing a suitable webcam API for Links (https://links-lang.org/, 2005) and using that to build a dynamic video chat web application. Such software includes the web-development-targeted functional programming language Links (ibid.), as well as the more well-known programming language JavaScript (Pluralsight, 1995), and finally, the peer to peer real time communication interface, WebRTC (Google, 2011). Some of the fundamental concepts and ideas include web development, distributed actor-style concurrency (https://en.wikipedia.org/wiki/Actor_model, 2021), and communicating to peers over the internet.

As well as introducing these systems and ideas, I will comment on various previous web systems, as well as web applications around today which have built on some of the concepts and technology I mentioned, and which are similar to the goal artifact of this project.

Web Development Familiarity

Knowledge of the way in which web sites are designed and implemented is key to developing a well-functioning and aesthetically appealing web site. Here I mention the term "web site" instead of the term "web page" because of how much more the former encompasses than the latter. I will come back to this near the end of this sub topic.

Creating a simple web page which does nothing but display some lines and text requires knowledge of only one computer language - the markup language, HTML (WHATWG, 1993). To introduce colour and a little bit of dynamism (e.g. changing the colour of text upon hovering over it) is still possible in HTML only, but much easier if a style sheet language, such as CSS (W3C, 1996), is used as well. Finally, if I wish to manipulate the web page more intricately, or require complex logic to allow a user to carry out a task, then it would be appropriate to make use of the programming language, JavaScript (Pluralsight, 1995), within a JavaScript file, along with the HTML and CSS files. However, a good understanding of these three elements would not be enough to implement a dynamic video chat application which is the goal artifact of this project. Knowledge of WebRTC (Google, 2011) - which is discussed later in a sub topic of it's own - and how to create a web server to act as an intermediary between the clients would also be essential.

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

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

Links and Web Development Without Tiers

What tiers refer to here are the three components which make up a web system. Typically, a web server - the middle tier (programmed in Java, JavaScript, PHP, etc.) - is implemented to deal with client's requests by generating web pages and graphical user interfaces - the top tier (programmed in HTML, CSS, JavaScript) - to send back to the client. The server might also need to access data from a database through queries - the bottom tier (programmed in SQL, XQuery, etc.). Consequently, a highly-skilled full-stack web developer will have expertise in at least several different computer languages.

The functional programming language, Links, allows web developers to design and implement web sites which incorporate a web server, dynamic graphical user interfaces, and queries to databases, all within a single language. Thus, if a programmer new to web development wishes to build their own web site from scratch, it may be of more ease for them to learn one programming language, Links, instead of learning several. This paper (Cooper, 2006) demonstrates several examples of web pages implemented in Links, and goes into more detail as to why Links is convenient for web development.

Despite the wide variety of web development functionality Links offers, it does not support video streaming or have video support of any kind. In this day and age, software applications in which users interact and communicate are expected to have functionality for streaming videos live. Therefore, implementing an API for Links to support video live streaming provides Links programmers extra capability in developing their systems.

On the same note, due to the lack of support for video handling in Links, I resorted to programming in JavaScript and using Links' foreign function interface (Fowler, 2017) to access the JavaScript code.

WebRTC API for Video Calls

In order to implement an API in Links to support live video streaming, I had to research into how live streaming is supported in web applications, typically. Generally, the MediaDevices[LINK] and WebRTC[LINK] web API's are used; MediaDevices is required in order to access the client's microphone and webcam, and WebRTC is vital in enabling rapid transfer of media streams between the clients. Briefly, WebRTC is a software technology which enables web applications to stream audio and video media between browsers without necessarily needing an intermediary web server.

Before diving straight into Links and attempting to implement a WebRTC API from scratch, it is important to understand deeply the process of setting up a WebRTC connection between clients and having them exchange video streams. I created a document[LINK] on GitHub which explains the details of how a WebRTC connection is set up between peers and also contains an example of a program which uses WebRTC to initiated a video call between multiple peers - the same example of which originates here[LINK]. In writing this document I was able to demonstrate to myself that I thoroughly understood the process and that I could program my own WebRTC video call application in JavaScript.

To summarise the process; firstly, there must exist a server which acts as an intermediary to allow the users to exchange connection and media details. Once this has occurred, and the users know about the best way to connect to each other, they initiate a WebRTC peer connection between each other. It is through this connection they transfer their video streams.

Programming a WebRTC program in JavaScript first made it much easier to implement the same program in Links, since I could use the JavaScript program as a useful reference, and also use the foreign function interface to call JavaScript functions and so make use of existing functionality. However, as explained earlier, I also required a web server to act as a medium for clients to exchange connection and media details. In the JavaScript implementation, I programmed a server using "Node.js" which would broadcast any messages received to all connected clients. In the Links implementation, I had to take a different approach because Links has it's own client-server model. With the guidance of a developer of Links, I came across a communication model which Links supports known as distributed actor-style concurrency[LINK].

This web page[LINK] explains the concept well, but I will summarise the important parts. A communication system which utilises this model comprises of several independent units of computation, or processes, which are referred to as "actors". Messages sent from the program to an actor are asynchronous and so the program will not halt to wait for a response. Actors typically wait for a message and use pattern matching to decide what to do when it receives one. It is distributed in the sense that an actor does not have to run locally - an actor can be run on a different machine and still receive messages, as long as the sender knows how to get messages to it.

In order to implement the WebRTC program in Links, it was important I made use of the actor model. I programmed the application so that, upon running it, a server process (actor) is spawned which waits for messages to arrive from client processes (actors). When a client connects to the server, the program spawns a client actor which represents this specific client, and then this client actor sends a message to the server actor. The server actor then keeps a record of this actor and sends a message to all client actors it has a record of to let them know a new client has joined. By implementing this client actor to pattern match on different types of messages, I made it possible for the clients to talk to each other through the server and set up a WebRTC peer connection.

In completing this, I extended Links to support live video streaming and functionality to allow a developer to access local audio and video media.

Existing Work on Dynamic Video Chat

The main inspiration of this project originates within the most well-known application of it's kind, "gather.town"[LINK]. In a nutshell, the web system is a web-conferencing web site which allows users to communicate over video call, but with the added capability of being able to "walk around a room" virtually and video call with those close enough only. It has been recognised as a platform which makes "virtual parties fun"[LINK] and has hosted events which turned out to be a "masterclass"[LINK]. Despite this, I, and many others[LINK], believe there are still aspects of it which could be improved to make for a better experience for various types of users.

In this thread[LINK], multiple users who have used gather.town make comments both constructively and critically. One of the problems gather.town has is the awkwardness to start conversion with strangers - when someone gets close to another user, their faces pop up on each other's screens immediately and there is no way to "read body language" to tell if a user is open to conversation. In short, it is difficult to "break the ice" with new people.

An intuitive solution to this which I could implement in my web application would be to highlight characters as a certain colour to show they want to talk. As well as this, my program could simulate "open/closed shoulders" to indicate whether a user or group is open for conversation. A further help for this would be to let users use emojis to represent their current mood. To tackle the problem of a user's face suddenly appearing on another user's screen would be to broadcast audio further than video so that a user can figure out the best moment to join a conversation.

Another problem users had is that the character jumps from one square to another and the background changes with it. This causes these users to experience nausea and dizziness. On a similar note, users who have not played computer games are not familiar with using WASD or arrow keys to move and so movement is trickier for these users.

To solve this, I could implement smooth movement - pixel by pixel, for example - and keep the background still. As well as having the arrow keys as an option, the user could move by clicking areas of the room or clicking in the direction they want to travel. If a room is larger than the user's screen, then the user could drag the screen to reveal other areas of the room.

It is also not clear whether a user's camera feed is visible to other users nearby. A user's camera feed could be displayed on their screen, but there is no clear indication that it is visible to anyone else.

A way I could solve this is by annotating the user's local camera feed with text, such as "Public" or "Private" to indicate whether their camera feed can be seen by others. A small "i" icon could be included beside the annotation which can be hovered over to read a small description of the meaning of the annotation.

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

I believe I could implement my system in such a way that the feature of breakout rooms seems much more natural to participants and teachers who make use of it.

References

Cooper, Lindley (2006). *Links: Web Programming Without Tiers*. URL: `https://links-lang.org/papers/links-fmco06.pdf`. (accessed: 01.08.2021).

Fowler, Simon (2017). *JavaScript FFI.*
URL: `https://github.com/links-lang/links/wiki/JavaScript-FFI`.
(accessed: 10.09.2021).

Google (2011). *Website for WebRTC.* URL: `https://webrtc.org/`.
(accessed: 20.08.2021).

https://en.wikipedia.org/wiki/Actor_model (Oct. 2021).
Distributed actor-style concurrency for messaging.
URL: `https://en.wikipedia.org/wiki/Actor_model`.
(accessed: 01.10.2021).

https://links-lang.org/ (2005). *Links Website.*
URL: `https://links-lang.org/`. (accessed: 05.09.2021).

Pluralsight (Dec. 1995). *JavaScript programming language.*
URL: `https://www.javascript.com/`. (accessed: 15.07.2021).

W3C (Dec. 1996). *CSS Docs.*
URL: `https://developer.mozilla.org/en-US/docs/Web/CSS`.
(accessed: 15.07.2021).

WHATWG (1993). *HTML Standard.*
URL: `https://html.spec.whatwg.org/multipage/`.
(accessed: 15.07.2021).