
# Web-based conferencing system for audio/video/chat communications with screen recording. CONFIDENTIAL

by

Pierce Lowe

This Report is submitted in partial fulfilment of the requirements of the Honours Degree in Electrical and Electronic Engineering (DT021A) of the Dublin Institute of Technology

May 23rd, 2022

Supervisor: Dr Miroslaw Narbutt

School of Electrical and Electronic Engineering

# Abstract

Since the beginning of the Covid 19 pandemic, millions of people around the world have had to work remotely and as such the demand for remote communication platforms has drastically increased in the last 2 years with a focus on video conferencing applications. There are many different platforms which provide such a service and although many of the native desktop clients use a proprietary solution, their web based counterparts are all based on WebRTC technology, a real time peer-to-peer to communication technology built for the web. This project aims to look at WebRTC technology and its various protocols more closely and from this, build a web based video conferencing application using WebRTC.

When it came to building the application, a number of components were required; a HTTPS and signalling server, browser client and a STUN and TURN server. The HTTPS server was built using Node JS and the Express server middleware and the signalling server with WebSockets (Socket.io) and the RTCMultiConnection server library. The client was built was with the React JavaScript front end framework, Styled Components library and the RTCMultiConnection client library. Finally the STUN and TURN server were setup on a remote server using the distributed coturn binary for Debian. The result was a fully functioning web based voice and video conferencing application with a text chat and screen sharing and recording functionality. The final application worked as intended across a number of devices and browsers.

# Table of Contents

# 1 Introduction

Since the widespread outbreak of the Covid 19 virus in 2020, much of the communication between people, especially in the work place and in education, was forced to be carried out remotely, usually through voice and video calling over the web using platforms such as Microsoft Teams, Zoom, and Skype to name just a few. According to the Central Statistics Office (CSO) as of November 2021 in Ireland, prior to the pandemic just 23% of employees who participated in the 'Our Lives Online' survey had worked remotely, while at the time of the survey 80% had now worked remotely at some point [1]. Although all restrictions limiting the physical meeting of people have now been lifted, many jobs are still being done remotely, with reference to the same 'Our Lives Online' Survey, 65% of the respondents continued to work remotely as of November 2021 [1].

Each of the previously named web conferencing platforms are built on variations of the same real time communication technology: Web Real Time Communication (WebRTC). This project aims to develop a web based video conferencing application, built using WebRTC to allow users to carry out remote voice and video calls in their browser. This report outlines the research, design, and development processes of the project.

## *1.1 Objectives*

The objectives of this project are to carry out a comprehensive investigation into the technology and Application Programming Interfaces (APIs) behind WebRTC and to create a web based video conferencing application using WebRTC. The application will implement a number of features, including voice and video calling, as well as a text chat, screen sharing and screen recording. Unlike some other services, users will not require an account to use the application, as authentication and authorisation are out of scope of the project. The project also aims to build a responsive graphical user interface (GUI) to allow the application to be device agnostic.

## *1.2 Ethics*

### 1.2.1 Security and Privacy

There are a number of ethical considerations to be made regarding the design and development of this project. Firstly, user data and privacy is an extremely important aspect of software development and failure to take it into account can lead to many issues. With users able to record a call for later playback, parties must to be aware of the data being recorded and in most

cases give explicit consent to the recording. There is however, a personal and household activities exemption in GDPR law that states video processing in a private environment is allowed. For user assurance and clarity, users will be informed of any recording both when a user begins and ends recording a meeting.

Naturally, as a web-based application it is susceptible to attacks, however the use of HTTPS on the server side should be sufficient as no sensitive data is being sent to or stored on the server, in the event of an authentication or third party authorisation system, further protections would need to be put in place.

### 1.2.2 Professional

There is also professional responsibility surrounding code plagiarism. Software development often includes research into problems that have been solved in many other cases with solutions shared by various authors and as such can if any code or approach to a problem is sourced from or contributed to by a third party, correct attribution must be given to the author, as well as complying with any licences put on the open-sourced code such as MIT, BSD or GPL licenses. The same applies to background information and research throughout this report. Any text or report documents that have been used as a source of information must be cited with correct IEEE referencing.

## *1.3 What is WebRTC*

WebRTC, first released in 2011, is an open-source technology developed by Google's Chrome team for real-time voice and video communication in the browser [2]. Today it's been standardised by the Internet Engineering Task Force (IETF) and is supported by all modern browsers as a collection of native JavaScript (JS) Application Programming Interfaces (APIs) and also supporting native clients such as Android and iOS applications through their respective WebRTC libraries [3]. WebRTC is based on a peer-to-peer connection model, primarily for the exchanging of voice and video data, but generic data transfer is also supported through the WebRTC data channels [3]. Due to the nature of the internet connecting billions of devices, each one sitting behind networks with various configurations and firewall rules, a WebRTC session takes a number of steps and follows multiple protocols to test these network rules and establish a peer-to-peer connection. This process will be reviewed in more detail in the Background section below.

# 2 Background

This section will look at the background of WebRTC technology, the process of connecting two remote peers, as well as looking at the considerations made when selecting the appropriate tools and technologies for building the application.

## *2.1 Establishing a Peer-to-Peer Connection*

To establish a peer-to-peer connection the clients to be connected need to be known. Typically this is achieved by the clients visiting the same website with a unique token in the URL; in the context of video conferencing, this is typically known as a "room" or "meeting" ID. When two clients visit the URL containing this token, the connection process begins with getting the peer's session descriptions.

### 2.1.1 SDP and JSEP

The session description for each peer is obtained through the Session Description Protocol (SDP) and JavaScript Session Establishment Protocol (JSEP). SDP itself doesn't send any information, rather it collects information about the client, this includes network metrics, media types, codecs, resolution, etc. SDP is not only used in WebRTC and has been around since 1998 [4] however for use in WebRTC it has been incorporated into JSEP. JSEP, is used to initiate the connection between peers once each user's session description has been obtained. It is not involved in any signalling; it is only concerned with the exchange of session description data through the use of the offer/answer model (see IETF request for comments (RFC) 3264 for more information on the offer/answer model) and is used to transfer the data collected by the SDP to the other client [5].

The JSEP process is as follows: The initiator of the connection creates a session description object which it stores locally containing the initiators session description. An offer containing this session data is sent to the peer through the chosen signalling channel. If the peer answers the offer, they store the initiators session description while also creating and storing their own. This is then sent back to the initiator and at this point the establishment of a communication channel between the peers begins.

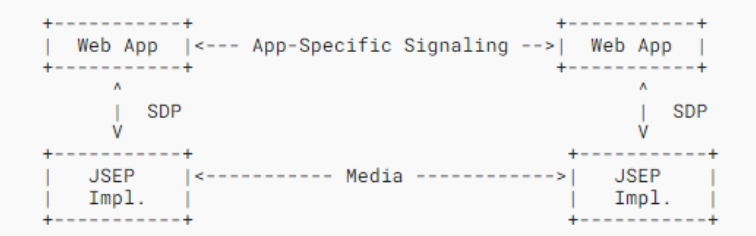



*Figure 1: Overview of JSEP [5]*

### 2.1.2 Signalling

To create a communication channel connecting two clients, information needs to be exchanged to determine how the clients can be connected, what data to share and the configuration for the data exchange. There is no set standard for this within the WebRTC standard; it is left to the developer to decide how to handle these session establishment messages. This is known as signalling and is carried out by a signalling server. This can be done in many ways; a naïve but simple approach is server polling. This involves the client constantly polling the web server with requests for any updated information and the server responding with the updated information it has received. A more efficient approach to this is done by utilising server-side events (SSE), although this requires the use of a secondary communication channel for the server to emit messages to the client [6].

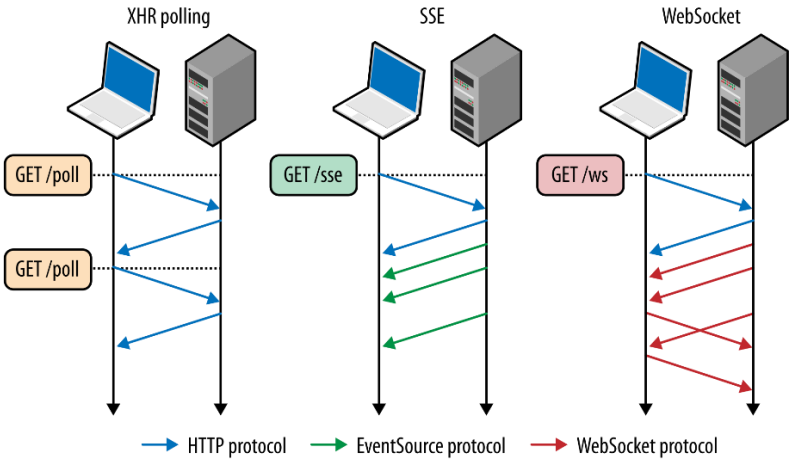



*Figure 2: Illustration of HTTP polling vs Server Side Events vs WebSocket communications [8]*

Recently Google's Cloud Firestore service has become quite a popular solution for use as a signalling server due to its focus on flexible database structures and the ability to push real-time database updates to clients through the use of listeners. The official WebRTC documentation now also includes details on setting up a WebRTC signalling server using Cloud Firestore [7]. However, another method of signalling that has been popular for quite a long time is using WebSockets. WebSockets utilise HTTP to establish a connection with the server,

then keep the TCP connection alive after the request has ended. This TCP channel is then used for two-way communication between a server and client and in terms of signalling this allows a client to send a message to the server, the server can then handle the data and push the information down to another client which also has a WebSocket connection with the server [8].

### 2.1.3 NAT

Once a method of signalling has been chosen the information exchange can begin. Firstly, the two peers must know the IP address of the other to establish a connection. Most network connected devices are used behind a private network. The issue with this is that most private networks use Network Address Translation (NAT). NAT is done by the router on a network and is the act of translating the internet protocol (IP) addresses of incoming and outgoing network packets between local and global address spaces. Each device on the network is given an internal, private IP address that maps to a unique public host address. The public host address is simply the routers public IP address, on various ports [9]. Each machine on the network is only aware of its private address, not its public one. If this private IP address is exchanged with a peer on another network, it will not be possible to establish the connection since this IP address is scoped to the client's local network. To circumvent this issue, the Session Traversal Utilities for NAT (STUN) protocol was introduced [10].

### 2.1.4 Protocols

#### 2.1.4.1 STUN

STUN is a utility allowing a devices to make a request to an endpoint, requesting its assigned public host (IP address and port) [10][11]. In most cases, STUN solves the issue that NATs introduce, however in some NAT configurations, the router assigns a new port for each connection. This means the port assigned to the device when contacting the STUN server would could not be used for the peer-to-peer connection. This means STUN was not a fully solution for NAT traversal. To solve this problem another protocol was then introduced; Traversal Using Relays around NAT (TURN) [10].

#### 2.1.4.2 TURN

TURN is a protocol which can be used to relay data from one peer to another by passing it through an intermediary server. This circumvents the issue of a NAT selecting a new port for each connection. However, the use of TURN completely removes the peer-to-peer nature of

the technology to solve a problem only faced in a minority of situations. Prior to the introduction to TURN, there was no way to attempt to establish a connection using one method and fall back to another in the case of an issue e.g. using STUN with TURN as a fallback. It was “all or nothing”. This brought about the introduction of the Interactive Connectivity Establishment (ICE) protocol around the same time as TURN [10][13].

#### 2.1.4.3 ICE

The aim of ICE is to try all possible pairs of address candidates (an IP address, port and transport protocol) between 2 devices to find a way to connect the two peers. This is done in a specific order to find the best method for making the connection [13]. The first candidates obtained are the private network candidates, followed by the candidates received from the STUN server and finally the candidate returned by a TURN server [13]. These candidates are then exchanged via the signalling server and candidate pairs are constructed in order of priority. The pairs are then tested by each client by making a STUN request to the peer's candidates, once a STUN request is successfully made on a candidate by both clients, the connectivity method has been determined.

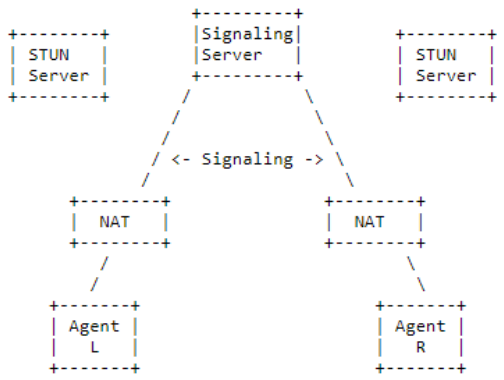



*Figure 3: Signalling Process, Network Topology [13]*

## *2.2 WebRTC in Practice*

To abstract away from these low level protocols, WebRTC in the browser context is built upon a number of different APIs specified by the W3C WebRTC working group and includes a large number of interfaces, types and events covering various aspects of the WebRTC connection and communication processes. There are three core WebRTC browser APIs which abstract away much of the interfacing with low level protocols: MediaStreams, RTCPeerConnection and RTCDataChannel APIs [14].

### 2.2.1 MediaStreams API

As well as finding a suitable communication path, the media to be shared by each peer must also be obtained. This is achieved through the use of the MediaStreams API which is designed to allow the browser access a client's input and output (I/O) media devices, such as web cameras and microphones. The primary function of this API in WebRTC is the *getUserMedia* function, it requests access to a user's local media devices and creates a MediaStream object composed of multiple media tracks. This object can then be used to create Hypertext Markup Language (HTML) audio or video elements to be displayed on a page in the local browser or can be passed to the RTCPeerConnection API to be streamed to a remote peer. The API also provides a number of events that can be listened for at both the MediaStream and MediaStreamTrack levels, such as listening for the addition or removal of tracks or updating the state of a track [14][15].

### 2.2.2 RTCPeerConnection API

The RTCPeerConnection API represents a single WebRTC peer-to-peer connection. It handles the majority of WebRTC interactions, from gathering ICE candidates and getting local and remote session descriptions for JSEP, to adding and removing video streams from the WebRTC communication channels [16].

### 2.2.3 RTCDataChannel API

The RTCDataChannel API is used to handle the WebRTC data channels, used for two way transfer of generic data between peers. An RTCDataChannel object holds a reference to an RTCPeerConnection object, while a single RTCPeerConnection object can have many individual data channels associated with it [17].

## *2.3 Considerations*

To achieve the goals of this project there were many considerations to be made regarding the tools that will be used to build the application. This section will look at the considerations made when deciding on the technology used in this project.

### 2.3.1 WebRTC

As previously discussed there are a number of WebRTC JavaScript browser APIs that can be used to manage WebRTC connections, however once the need for multiple simultaneous connections is introduced, the use of these APIs becomes more complicated and difficult to

manage. As a result many wrappers exist to abstract away some of these complications. Due to the limited timeframe of the project and there being a wide variety of wrappers available, a wrapper was opted to be used for this project. Some of these wrappers only included functions for handling the WebRTC API calls such as peer.js, however, many of these wrappers also handle the MediaStream API calls as well, which further abstracts the complexity of establishing a WebRTC connection, these libraries include simple-peer, lib-jitsi-meet and RTCMultiConnection.

As mentioned, one of the objectives of the application is to allow multiple users to connect simultaneously, there are multiple approaches to this achieve this, namely mesh, star and routing topologies [18]. Of the three, mesh topology is the only one which does not remove the peer-to-peer nature of system while the other two relay the data through a central server.

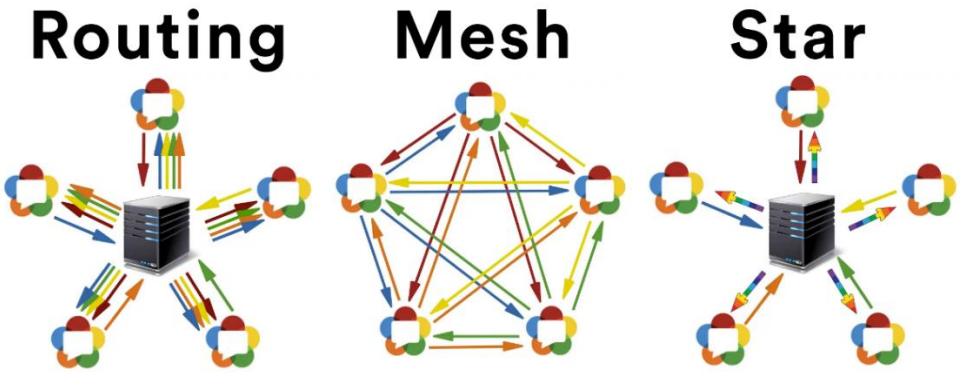



*Figure 4: Comparison of routing, mesh and star WebRTC topologies*

Both lib-jitsi-meet and RTCMultiConnection support mesh topology handling in their libraries and so these seemed the most suitable for this project. The RTCMultiConnection library was chosen for the task based on a recommendation from the project supervisor.

The library written and published by Muaz Khan, provides a wrapper around many of the WebRTC functions, as well as providing a number of helper functions outside the scope of WebRTC, such as the addition of a file selection handler, recording and writing streams to a file, among others. It also includes a signalling server library to be used on a server alongside the client library. Using the two libraries together allows it to handle all the signalling required when establishing a connection, making the process of getting an initial working application a much simpler process.

### 2.3.2 Frontend

Nowadays it is rare that a web application will be developed with “vanilla” HTML, CSS and JavaScript. A JavaScript/TypeScript framework is typically used to build the application, which is then “minified” (compiled) to single HTML, CSS and JavaScript files which are then served to the client. There are a plethora of frameworks available each with their own advantages. The most popular of these, although by no means the only options, are Angular, developed by Google, React developed by Meta and Vue maintained by the core Vue team.

For a standard HTML, CSS, JS website, it is composed of static files on a webserver which are served to the client usually in their final state, however JavaScript can be used to modify the page content on the client. An alternate approach to this is client side rendering (CSR) which is used by each of the three frameworks mentioned above use. They can also be configured to use Server Side Rendering (SSR) or Static Site Generation (SSG), typically through the use of a “meta-framework” such as NEXT which is an SSR framework or Gatsby an SSG framework both of which are built on top of the React framework.

#### 2.3.2.1 CSR vs SSR vs SSG

Client side rendering means the client is served a boilerplate HTML file along with a minified JavaScript file. The code contained in the JavaScript file is used to render all the content on to the page within the browser client, this requires the browser to support JavaScript which is virtually not a problem nowadays however more complex or poorly optimised CSR sites require more device resources to render content and can be slow on less powerful devices. SSR is similar to CSR, in that a page is only rendered as it is requested by the client however the content is rendered on the server before being served down to the client. This reduces load on the client, improves search engine optimisation (SEO), and offers faster page load times. However it requires a full page reload when navigating to a new page unlike CSR sites. SSG is similar to SSR with the main difference being an SSG site is rendered at build time instead of at call time.

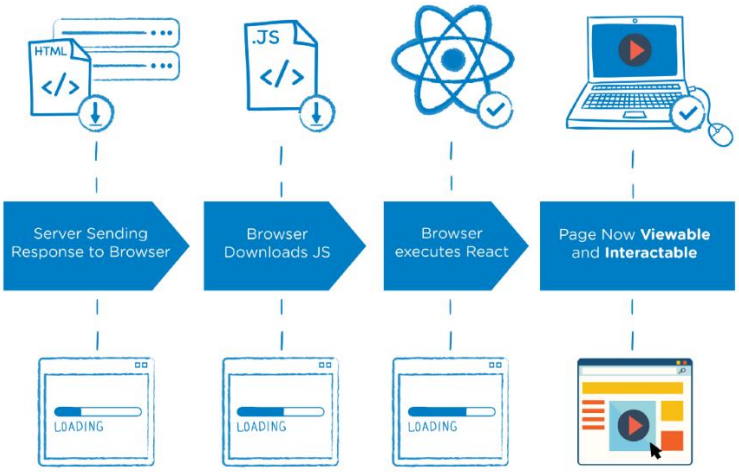



*Figure 5: Client Side Rendering Flowchart [19]*

Due to the nature of the web application largely consisting of video streams and chat messages, means that the majority of content data on the pages will be different every time a user visits and will need to be updated live on the page. This lead to the decision to use a CSR framework. Each of the frameworks have their design paradigms and concepts to be considered when choosing one over the other however they all largely supply tools to achieve the same goal of a stateful CSR web application. React was chosen as the front end framework for this project due to having some previous experience with the framework and it is by far the most popular of the three frameworks [20]. It is also actively being used to develop other WebRTC browser applications such as Microsoft Teams, Meet Jitsi and Discord.

#### 2.3.2.2 React

React follows a stateful, component based model. Components are essentially the building blocks of a React site, they are “independent, reusable pieces” [21] of code that can be classes or functions which return a React object. They can take in a number of properties and maintain their own state between page renders. Components can maintain state across renders due to the React Virtual Document Object Model (DOM). The use of the Virtual DOM allows for React to manipulate specific elements of a page as needed without updating the real DOM then when a page is re-rendered it updates the DOM to match the virtual one stored by React [22].

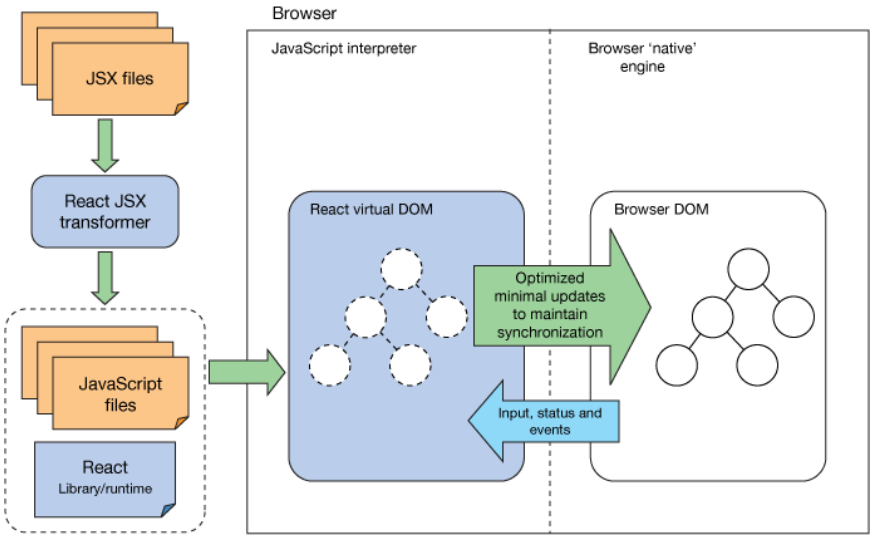



*Figure 6: React JS Framework Architecture [22]*

React re-renders are triggered when any of a components properties or states change. A re-render only changes that specific component on the page, while leaving the rest untouched. All data in that re-rendered component is reset unless it is marked as stateful or is a property passed by the parent in which case it will maintain its current value [23].

The issue with the Virtual DOM is that is that native DOM API calls cannot be made. This can be an issue with utilising native DOM libraries such as the RTCMultiConnection library. Some of the functions in the library reference and create native DOM elements, which cannot be used directly in a React environment. To overcome this, React has added "refs" which hold a reference to native DOM instances of React objects. This allows native DOM API calls to be made on that element via the reference [24].

#### 2.3.2.3 Styled Components

Aside from using React to handle the page structure and user interactions, native CSS is required to be used within React, this is a perfectly viable option however there are plenty of powerful CSS pre-processors available such as SASS, LESS and PostCSS which allow the developer to write CSS in a custom syntax, unique to that pre-processor which is then compiled into native CSS [25]. These are powerful tools however, these approaches require a completely new syntax to be learnt and in a React application where the HTML and JavaScript is tightly coupled, the CSS remains separated from the application. An increasingly popular approach to more tightly couple CSS into an application is known as "CSS in JS". There are multiple libraries that implement this functionality, such as Styled Components and JSS. These allow for CSS to be written inside JS facilitating for cleaner syntax, updating styles based on the

applications state, as well as nested CSS syntax for referencing child elements and pseudo selectors. A third alternative to CSS is the use of CSS frameworks such as Bootstrap or Tailwind CSS which provide complete component styles for the likes of buttons, navigation bars, dropdown menus, etc. However, these frameworks offer much less control to the developer in terms of the visual style of the application and so for this project a "CSS in JS" library was opted for; namely Styled Components, simply due to its popularity compared to other "CSS in JS" libraries.

### 2.3.3 Server Side

On the server side, three things are needed. One: a signalling server to establish a connection between peers; two: a webserver to serve files to the client; and three: a STUN and TURN server, although Google does provide a number of free STUN servers for use by anyone; and a TURN server is not strictly necessary for WebRTC to work.

Any language can be used on the server however as mentioned previously the RTCMultiConnection library includes a signalling server library which is designed to be used in tandem with the client library. It is quite robust and handles a wide range of signalling events including all events needed to establish a WebRTC connection between 2 peers. The library is written in JavaScript for use with Node.js, an open-source, JavaScript runtime environment built on top of Chrome's V8 JavaScript engine [26]. This limited the choice of server side language to JavaScript running with the Node runtime.

#### 2.3.3.1 Node

Node was released in 2009 and as mentioned, runs on the V8 engine. It is modelled as an event-driven, single threaded non-blocking I/O runtime environment [26]. The single threaded non-blocking nature allows the programmer to write asynchronous code without the need for manual thread management. This model is implemented as the Node event loop. When a function is invoked it is place on the V8 engine's call stack which holds a reference to the function. The function is then executed within Node's internal C++ thread pool. Once the function has been executed, the function reference on the call stack is replaced with the return value, which is then popped off the stack on the next iteration of the loop [26]. Once the call stack is empty i.e. there are no more functions to be executed, the process will then exit.

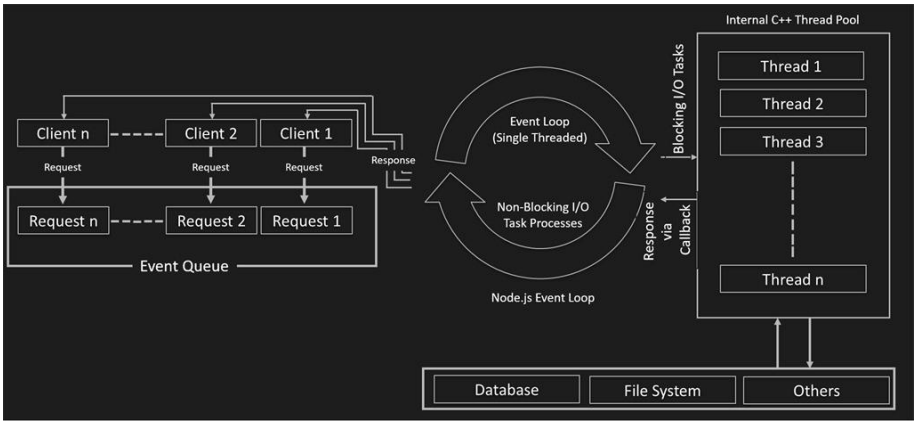



*Figure 7: Node JS Event Loop [26]*

### 2.3.3.2 HTTPS and Signalling Server

Node has a standard HTTP library that can be used to establish a HTTP/S server for responding to requests and serving files. However, as with everything, there are a number of alternatives to simplify the process. The industry standard for HTTP/S server development with Node is the Express framework. Express has become synonymous with Node and is actively maintained and very well documented with a lot of third-party Node frameworks built on top of Express such as Keystone CMS and Sails MVC [27]. For these reasons, it was chosen as the server framework of choice.

Express offers a much cleaner and more human readable syntax for managing HTTP requests than the standard library without abstracting away too much of the control offered by Node. Due to not abstracting away much detail and only maintaining a thin layer above Node, it does not add much overhead on top of running a HTTPS server built with Node's standard library.

The signalling server library uses WebSockets and so this decided on the signalling technology to be used. It is intended to work with the popular, multi-language, WebSocket library: Socket.io. It takes a Socket.io object as its only parameter to be constructed and handles all incoming WebRTC signalling messages through the socket, as well as allowing for custom events to be sent to a single peer or broadcast to all peers.

### 2.3.3.3 STUN and TURN

As mentioned previously a STUN and TURN server was to be configured for this project, developing one from scratch however, was not within the project scope. The most common solution is the coturn TURN server project. It is a free and open source STUN and TURN

server written in C which is distributed as a compiled binary, a Docker image, or can be built from source [29].

The project is based off rfc5766-turn-server however, it implements far more TURN specifications that were added in later IETF RFCs as well as implementing STUN server and ICE specifications as per their respective RFCs [29]

# 3 Methodology

This section of the report will take a look into the development process of this project. The development process can be broken down into two sections, one being research and the second being the application development itself. This report will break them down into two distinct sections although they were conducted simultaneously at various times throughout the project.

## *3.1 Field Research*

### 3.1.1 Overview

Before beginning any development, the website requirements were laid out to promote a structured development process. Minimum requirements were as follows: a landing page to allow users to setup meetings and the page where the meeting will be conducted. A review of similar applications was also carried out to examine various approaches. One of them being Jitsi Meet: a WebRTC based video conferencing application by the Jitsi team. Unlike most video conferencing applications it requires no account to use, a user is only required to enter a unique room name to start a meeting. They are then redirected to a webpage where they can enter a display name and configure their media devices. Once ready, they can click a button to start or join a meeting. Anyone who then visits the URL can join the meeting and after requesting to join, a peer-to-peer connection is made following the procedure discussed in the Background section. Following this the peers begin exchanging audio and video streams.

This user experience is quite intuitive and user friendly. It also provides enough control to the user to adjust their environment before joining. Alternatively, both Microsoft Teams and Zoom require the user to have an account to be able create a meeting. With Teams, once the meeting is prepared the user is met with a similar join meeting screen allowing the user to change their media settings before joining, while Zoom does not and so media settings must be configured in the application settings instead. This page is extremely helpful from a user perspective as it allows them to ensure everything is set up correctly before joining.

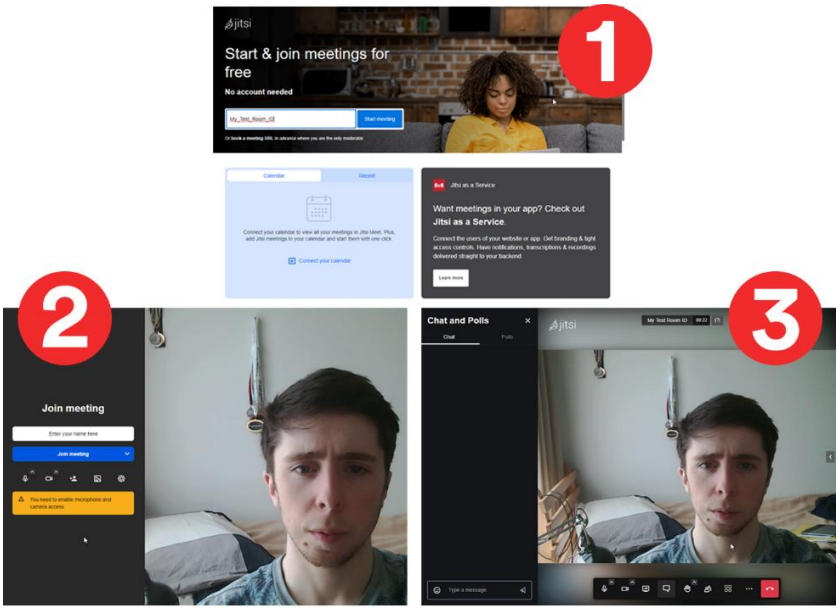



*Figure 8: Process for joining a Meet call: 1. Landing page where user enters room name; 2. Pre-meeting page where user configures settings; 3. User has entered call*

Meet is built with React and examining the routing on the Meet website, the pre-meeting screen is the at the same URL endpoint as the meeting itself. This can be achieved through a state-based design to determine which layout to render for a given route, however a meeting entry page such as the ones discussed is not a necessity for the application to function and as such will not be implemented in this application.

### 3.1.2 Landing Page

Reviewing landing pages on a number of popular online service websites (See Appendix A) found that it is common practice to keep a landing page quite simple without overloading the user with information. Typically a landing page consists of a name or logo in the top left corner of the page, with various options lined across the top of the page. If the page is longer than the height of the browser window these options typically stick to the top of the screen as a quick access, navigation bar (navbar) as a user scrolls down the page. The main content of the page usually includes a title, a tag line if appropriate, along with a button or form to take the user further into the site. Some sites have more information further down the page while others restrict it to the size of the browser view only containing necessary information.

Looking at the Meet site again, although a user can create and join a meeting with any form of authentication, on the landing page an option to authorise with either a Microsoft or Google account is provided. Authorising with these services allows a user to schedule meetings with the calendar services provided by Microsoft and Google. An option for third party authorisation

was considered as an extra for this project, however building a meeting scheduling system was out of the scope of the project and so was not a priority. If authorisation support was implemented, it could have been used to automatically set a user's display name when joining a meeting by reading the username of the authorised account.

### 3.1.3 Room Page

The minimum requirements for a room page were an area to display the video streams in a responsive grid that adapts to the browser window size and devices screen size, an area to display text chat messages and a chat input box and an area to hold function buttons such as mute microphone, disable camera, leave call, etc. A common layout found in many video streaming services, is to have the video stream(s) in the middle of the scene with the chat taking up a small portion at the side, filling the full vertical height of the page. In most popular video conferencing applications, function buttons are typically placed in a toolbar positioned at the top or bottom of the window. This layout is extremely popular as it positions the most important content centrally on the screen with lesser content taking up a smaller, less intrusive area.

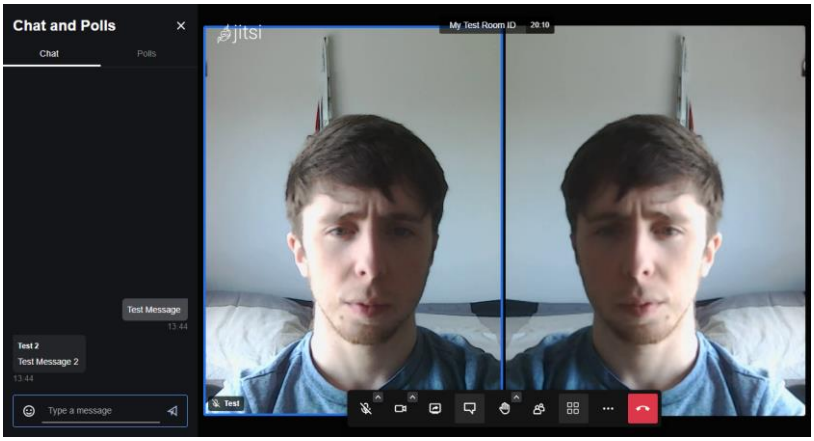



*Figure 9: Jitsi Meet meeting page*

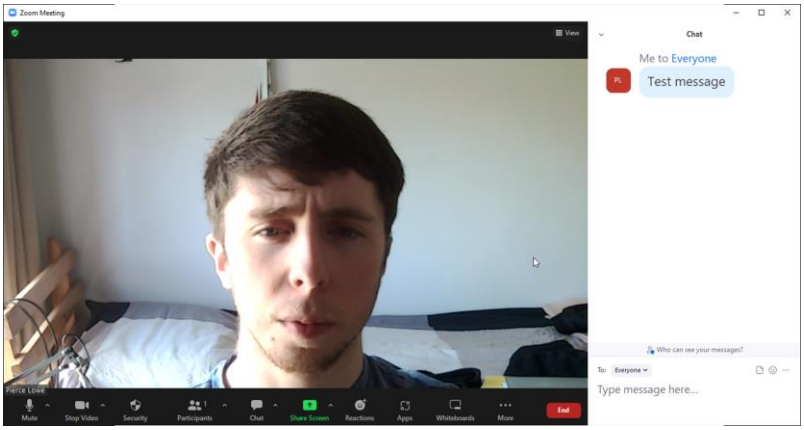



*Figure 10: Zoom meeting page*

As can be seen from the above images, both follow the same basic layout as discussed although with some slight positional variations. Also note, in the Jitsi Meet image, with a multiuser meeting where there is insufficient space to fit both streams on screen, to avoid introducing empty vertical space, the aspect ratio of the streams is disregarded, and the streams are cropped. This only occurs until the browser window becomes too narrow at which point the streams are stacked vertically rather than horizontally. As the window grows wider eventually the streams maintain their aspect ratio rather than being cropped. The nature of this responsive grid is common to most video conferencing applications. Implementation of such a system is non-trivial as logic needs to be implemented to handle when the streams should be cropped, stacked, or display at their native aspect ratio. Several approaches were tested to implement this functionality which will be discussed later.

Finally looking at the chat location, both applications place the chat to the side of the window with the chat input box at the bottom, however the two implementations placed their chat areas at opposite sides of the window and scroll the messages in opposite directions. Looking at other chat implementations such as with Microsoft Teams, Twitch and YouTube the chat box is typically to the right as in the Zoom implementation, with the newest chats scrolling from bottom to top, as in the Meet implementation.

## *3.2 Development Process*

The following section will examine the implementation of the server and client side code for the project. First it will look at how the HTTP server and the signalling server were built, followed by the client application and finally the setup of the STUN and TURN server.

Firstly, both the RTCMultiConnection client and server libraries included files for a demo application written with plain JS. Before starting any development, this demo was setup in the lab and tested with multiple clients connected to the server and the connection establishment process using the library was examined, as well as looking at the impact of video performance as the number of streams increased, this was expected to occur due to the use of mesh topology, which required more bandwidth as the number of streams increased, than both start and routing topologies.

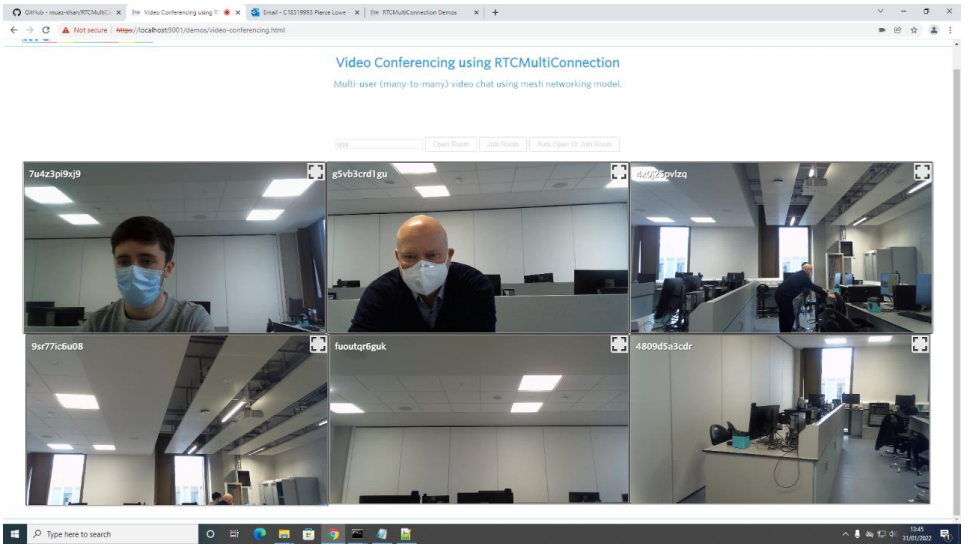



*Figure 11: Using the RTCMultiConnection Client and Server Demo*

What was found from this initial testing, was that after 12 connected peers, clients began dropping connections between peers before the number of streams reached a point where the media quality degraded or browser window began to slow. With such a high number of peers connected before connection issues began to occur, it was concluded that media stream performance was not something to be concerned with in the context of this project. The first step after testing the working demo, was to build the HTTPS and signalling server to serve the demo client application.

### 3.2.1 HTTPS and Signalling Server

As previously discussed in the Background section, the signalling server was implemented using the RTCMultiConnection server library and Socket.io, while the HTTPS server would be built on top of the Express server framework. Building the Express server was a trivial task as it required some basic configuration, enabling a single route for GET requests to the React application, as all other page routing is handled on the client by React. Setting up the signalling server with Socket.io was also straight forward, as it was mostly handled by the RTCMultiConnection server library.

At the top of the file any necessary modules are imported, following that a number of server configuration options are setup, such as the servers listening port, security certificates for HTTPS and signalling server configuration data. After that the server is configured to trust proxies, this is enabled in the likely case of the Express server would be hosted behind a reverse proxy such as Nginx. Cross origin resource sharing (CORS) is also enabled to allow the client to load resources from foreign domains and finally the React application's static build directory

is made publicly available. A get request handler is then set up for all routes to serve the index.html file from the build directory since, as previously mentioned, all page routing is handled by React on the client. Finally a HTTPS server object is created passing the security certificates previously defined along with the Express server object. The HTTPS server then listens on the defined port with a listener setup using the RTCMultiConnection server object to handle signalling. (See Appendix B)

Next the Socket.io object is created and configured with the HTTPS server and enabling CORS and requests from all origins. A connection event listener is then created on the Socket.io object. When a connection event is received the socket connection is added to the RTCMultiConnection server object to handle the signalling process. Finally, the socket is set up to handle and emit incoming custom events to peers which are not built into the signalling server library. (See Appendix B)

### 3.2.2 Client

Once the signalling server was successfully set up and working with the demo application, client side development started. Before any work started on building out the application, an initial test was setup to use the RTCMultiConnection library with React. This was done due to the requirement to use React "refs" to be able to use native JS libraries. The application was built without a usable interface, simply a button in the centre of the page that, when clicked, opens a WebRTC communication channel. As can be seen in Figure 12, from the React developer tools icon in the top right corner of the screen, the site is a compiled React application being served by the Express server. The console log messages indicate that the client makes a connection with the Socket.io signalling server, gets the audio and video media devices of the client machine and starts a "room" through the RTCMultiConnection library which listens for other connections on the same endpoint, at which point the signalling process begins between the server and two clients.

Once this was tested and working development of the main application started. The process of building each of the pages followed was largely the same process: design the page layout; scaffold the structure of the page with HTML elements; style the elements with the previously discussed Styled Components library; add functionality to each of the elements; test the result and repeat to adjust the style and functionality as required. This worked for the initial stages of each page, however as the pages became more complicated this became more loosely followed and changes were simply made as needed.

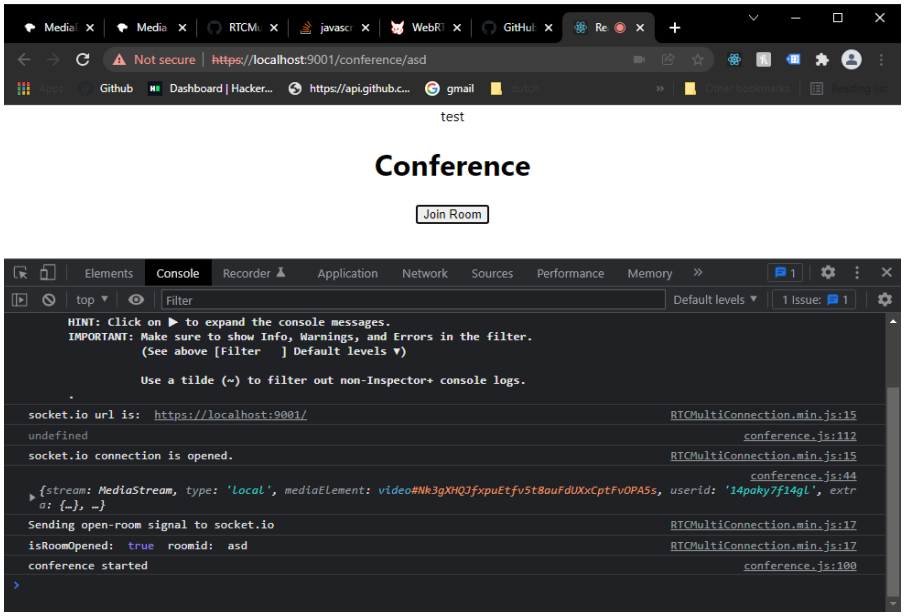



*Figure 12: Test React application using RTCMultiConnection library*

### 3.2.2.1 Landing Page

Before writing any code both the landing page and meeting page were laid out. The meeting page was straight forward as there was not much that could be changed from the typical layout of an online meeting room as discussed in the Background section. However, the landing page required more thought and iteration. The objective of project was not web-design and so the design process will not be discussed here in detail, however it can be found in the project logbook. The planned design for the page can be seen in Appendix C. Once laid out the implementation was started. It was relatively straight forward, the React application was initialised using Facebook's Create React App command line interface (CLI). This sets up the basic folder structure and installs the minimum dependencies for the project. From there, directories for components, data and styles were added to better organise the files. The landing page was then put together with plain HTML elements following the planned approach as laid out in Appendix C.

A number of global and "mixin" styles were first defined, mixin styles are commonly used partial styles that can be mixed with other styles to complete a style for a CSS class, such as setting the display attribute to flex, then aligning and justifying items to the centre, etc. (See Appendix E). After that, the styled components were constructed and replaced the default HTML elements that were initially used (See Appendix G). The styles were then adjusted until the landing page looked as intended. The initial implementation of the landing page had a "features" section further down the page however this was later removed as it was found to be

unnecessary for this project. As a result, the landing page footer was brought up to the bottom of the screen and the entire landing page contained in a single view (See Appendix K).

In terms of landing page functionality the only thing to be implemented was reading the text inputs and redirecting the user to the room URL. The room ID and display name inputs updated two stateful strings as they were changed. When the join button is clicked the user is redirected to the /room/{room-id} endpoint. If no room ID has been entered by the user, a random room ID is generated as an alphanumeric string. If no display name is provided the user is prompted to enter a display name on entering the room. However if a name is provided on the landing page, it is read and stored in session storage and linked to the room ID, this allows a user to use different names in different rooms and using session storage instead of local storage allows a user to have different names in the same room across different tabs or browser windows (i.e. outside the current session) (See Appendix G). When the user is redirected to the room page, the routing is handled by the React Router module, which has its routes setup in App.js (See Appendix F)

#### 3.2.2.2 Joining a Room

As mentioned, if a user has not entered a display name when trying to enter a room they are prompted to do so as they enter the room. From the discussion in the Background section, it is common to have a pre-meeting screen where the user can adjust settings. Provided more time on the project, this would be the ideal approach and using the *enumerateDevices* function as part of the MediaStreams API, the user could select their desired media device to capture. However as this was not being added to the project within the given time frame the only input needed was a display name for the user. As such, there was no need for a full intermediary page layout to handle this, instead a prompt was made using the native browser window API function: “prompt”, which creates a small input prompt on a page for the user to enter text data.

#### 3.2.2.3 Room Page

The process for building the room page largely followed the same process as with the landing page, however the functionality of it required a lot more iteration. The initial design of the room page was quite straight forward and can be seen in Appendix D. It followed the same structure as previously discussed with the main video grid taking up the majority of the screen, the chat window to the side and the toolbar to the bottom of the page.

Once again the page was initially scaffolded with standard HTML elements before replacing them with Styled Components. The first of the main components made was the toolbar this was done as it was the only item with fixed content. The toolbar is contained in an area which is positioned at the bottom of the window and spans the width of the video grid and the toolbar itself is centred within this area. This was done as it allows the toolbar to be hidden when not in use. If a user wishes to interact with it, then hovering over the containing area pulls it up from the bottom of the screen. This gives more screen real estate to the information the user needs to see i.e. the streams. The toolbar itself was then created as a simple styled div component containing an unordered list (See Appendix H). Each of the list items were also styled to grow slightly when hovering to indicate their interactive nature. The changing style on hover, especially in terms of pulling the toolbar up from off screen was made significantly easier through the use of Styled Components.

After the visuals were complete for the toolbar, an area for the chat was set aside and was made responsive by filling up the full screen or being completely hidden once the browser window became too narrow. This was then left aside until later and work started on the responsive video grid. There were many approaches taken to achieve the desired effect. As such development on this aspect was started early on, however work moved on to the other aspects of the project before coming back to work on it later. This report will treat it is one continuous process.

There were a few factors to consider which affected the layout of the grid:

1. The number of streams
2. The size of the grid area i.e. the size of the browser window - chat area
3. Should the stream be cropped or maintain its aspect ratio?

The first approach to this problem was to use CSS Flexbox to handle the streams as it is designed around flexible adjustable layouts. However it became an issue when attempting to set the CSS rules determining when to break onto a new line or when to crop the stream to a more narrow view. The issue with Flexbox is that it's commonly used for distributing items along a single axis rather than in a full grid area although can be used to fill an area with the "flex-wrap" property. The next step was to use CSS Grid instead as it is more suitable for grid layouts and did work to a certain extent with using the auto-fill and repeat properties, however again limitations arose once it came to determining when to break onto a new line or crop the streams. Finally a mix of Grid and Flexbox were used together, following a technique shown by Jen Simmons a Member of the CSS working group [30]. This almost worked as desired

however required a minimum width to be explicitly set to determine the breaking point rather than being able to determine it based off the number of streams and the grid aspect ratio. At this point it was clear that it would not be possible to achieve the required nature solely using CSS, since information about the current width and height of the grid and number of child elements in the grid needed to be known. Using the browser inspector on the Jitsi Meet application, it was seen that the video stream layouts were using inline CSS width and height properties that were changing dynamically; suggesting it was also achieved using JS and not just CSS. To simplify the development of the responsive grid, a blank HTML file was created with the stream listeners for the RTCMultiConnection added creating a basic WebRTC application. This provided an isolated environment removing some of Reacts complexity and allowing development to focus solely on the grids responsiveness. CSS Grid layout was kept as the base CSS layout and JS was used to set the number of columns and rows the grid should contain. The main challenge with this was determining the rules for when to update the various states:

1. Increase or decrease rows and columns
2. Invert rows and columns i.e. switch from horizontal to vertical layout
3. Crop stream or maintain original aspect ratio

The first of these was approached by first drawing up a table with the number of rows and columns needed for $x$ number of streams.

| **Streams** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Columns** | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 4 |
| **Rows** | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 |

*Table 1: Column and Row count for various numbers of streams*

From Table 1 it can be seen, that when the number of streams is equal to the number of columns squared, the number of columns is then increased on the next stream added. The table doesn't show enough row increments but once the number of streams reaches 12 the rows increment again to 4. From this pattern the number of rows should increment on streams 2, 6, 12, 20… the difference between each of these is 4, 6, 8, etc. This can also be calculated by the lowest number of rows multiplied by the number of columns that is greater than or equal to the total number of streams. This was implemented in code as follows:

```
for (let i = 1; i ≤ tempCols; i++) {
    if (streamObjs.length ≤ i * tempCols) {
        tempRows = i;
        break;
    }
}
```

*Figure 13: Calculating the number of rows in the grid*

Where *tempCols* is the new number of columns, *tempRows* is the new number of rows and *streamObjs* is an array of stream objects. For each new stream the number of rows is calculated from 1 each time, however the number of streams the application can handle will reach its limit sooner than the number of rows becomes greater than 5 and so iterating from 1 each time will have a negligible impact on performance (See Appendix I).

The next task of determining when to invert and revert the row and column count was much simpler and is determined by if the grid area is narrower than a 4:3 aspect ratio. Initially it a 1:1 aspect ratio was used, however that allowed the streams to become too narrow before inverting the layout. Finally determining when to crop or maintain stream aspect ratio followed a similar process to the layout inversion rule: When the width of the video div becomes less than the height, the contained stream should be cropped, but once the width is greater than the height, the stream should maintain its original aspect ratio.

After establishing these rules in a separate environment using plain JS and CSS, it was translated to work with React, however this brought with it a number of issues. Firstly, these rules required the width and height of the grid and its children to be known. Because React uses a virtual DOM the dimensions of components are not accessible directly through the React component itself, rather a reference to the actual DOM object for the grid has to be kept to be able to reference it and its children's dimensions. Whenever a new stream was added or the browser window was resized a check had to be made to decide if the layout should change. This was done by storing the streams in a stateful array, as well as the number columns and rows as stateful integers. When the streams array was updated it triggered a React hook which then updated the rows and columns causing a re-render of the grid. This took quite some time to determine which properties should be stored React references and which should be stateful, as well as determining which function calls to make on these changes.

There was some difficulty surrounding this because if a function in a React component is to be called on a state variable change it must be wrapped in a *useEffect* hook. These *useEffect* hook functions are called on the first load and then can be set to be called again on every re-render,

on a state change or not at all. In this case the grid update was to be called on the streams array being updated. There isn't a problem with this until another state value is referenced within the function. A *useEffect* hook has a dependency array, which must include any stateful variables the hook depends on. The function is called on each dependency change. Any state values referenced in the hook must be added to the dependency array to ensure future function calls have the most up to date value. This became an issue when requiring to read the column and row values and also update them within the same function, causing maximum depth recursion thus raising an exception. To circumvent this issue the rows and columns were copied to temporary variables within the function and the call to update the stateful values was only made if these temporary values differed from the current values (See Appendix I).

While working on the responsiveness of the video grid, the WebRTC functionality was being added alongside it. The use of the RTCMultiConnection library meant that a large part of the work on the RTC aspect of the client was setting up event callbacks. Before any event handling though, the user must create a WebRTC connection with a peer. This is also handled by the library, in which an *openorjoin* function is called which opens a new "room" if it doesn't exist or joins an existing room if it does. The function opens a WebSocket with the signalling server, when another client visits the same room address they also make a WebSocket connection with the server. At this point the signalling process to connect the peers begins. The *openorjoin* function is called within a *useEffect* hook which only depends on the room ID, this ID is only set once, when the page is first loads and so the client only attempts to join once (See Appendix I). Also, inside the *useEffect* hook are event listeners for new streams being added, the user enabling or disabling their camera and for incoming messages. There were a number of issues encountered when listening for new streams such as a new stream being added twice and on leaving the room the video would be removed however the video controls were left behind. These were tackled as they arose and were circumvented by adding a number of checks when adding and removing streams.

As the RTCMultiConnection library is designed for native JS applications on a new stream being sent over the RTC video channel the library constructs a native DOM video element that can be added to the DOM by the developer without requiring them to construct their own. Since a reference to the grid was being kept the video element created by the library could have been simply appended to the grid using the reference, however this would also require the developer to listen for a peer disconnecting and then search for the relevant video element in the grid to remove it. Using React provides a much simpler approach to this. A stateful array of stream

objects were kept. These objects contained a React component equivalent of a HTML video tag, the media stream itself, the randomly generated user ID, the username and their muted state. In another *useEffect* hook which depended on the video streams array, the video tags in each stream object had their source attribute set to their media stream. This function then checked if the tag already had a stream associated with it and if not one was added. Finally the array was added as a child of the grid with the array mapped to return the React video components wrapped within a containing styled div used to display the associated username over the stream when a cursor hovers over it. When the user "mutes" or "unmutes" their camera, the source attribute is simply added or removed from the video component. When the source attribute is empty i.e. the camera is disabled, the video simply displays the poster image which was set to a generic grey image with a silhouette of a person (See Appendix I).

Finally taking a look at the chat section, this was built in a similar manner to the responsive grid by building it in an isolated environment removing the complications of the other systems. This was quite straight forward, it required registering an event listener for new messages, when a new message arrives, a message object is constructed by placing the authors name and the message data into a span tag, the spans are appended to an array of messages with Flexbox being used to style the messages as a column ordered with newest messages at the bottom. The conversion to the React application for this was quite straight forward. The message listener was registered in the room ID *useEffect* hook, so it is only registered the first page load. The listener adds incoming messages to a stateful array of styled span components containing the message data. This array was then added as a child of the chat area div. React then unpacks the array and adds each element as a child of the chat area. Just as in the isolated tests, the chat area uses Flexbox to order and display the messages from bottom to top (See Appendix J).

Other than the camera and microphone toggles, leave room and copy link buttons, there were also toolbar buttons for screen share and recording. Implementing screen share required making new call to *getUserMedia* for screen type media. This prompts the user to select a screen or window to share which in turns creates a new stream object, the screen video track then replaces the track in the stream being sent to the connected peers and the client's local stream. The previous camera stream is then restored when the screen share ends using the copy of the stream element stored in the streams array.

The screen recording implementation was quite straight forward. It is invoked in the same way as screen sharing. The selected window to record is then captured and written to a file on a client's machine for later playback, with a message being sent to all peers whenever recording

is started or stopped. The audio for the recording was captured by parsing all local and remote streams, selecting their audio tracks and adding them to the screen stream object (See Appendix I).

### 3.2.3 STUN and TURN Server

The STUN and TURN server did not require much work to setup. The coturn precompiled binary had to be installed and the server configuration edited before running the coturn service. Although there were a lot of different configuration options that could be set, only a small selection of these were of concern for establishing a STUN and TURN server for a WebRTC application, such as the listening ports for each protocol, which protocols to enable and the authentication configuration which is required for TURN servers to be used with WebRTC (See Appendix B). However due to the limited testing environment without a web server, the STUN and TURN servers were never utilised, as suitable ICE candidate pairs could be found using each peers private IP addresses as they remained on the same network.

## *3.3 Final testing*

For the final testing scenario, 6 clients were set up in a lab with web cameras and microphones on a private network. Unfortunately due to a security breach on an old webserver the required ports on the network could not be opened, hence the limitation to testing on a private network. One machine acted as the server while the rest were clients and all joined a single room. While in the room a number of tests were conducted. The clients tested that:

1. Each client received audio and video streams from all peers
2. Mute and unmute for both camera and microphone worked
3. Screen recording worked as intended and could be played back
4. Screen sharing worked across peers

The first two tests ran as expected, however initially there were issues with screen recording and sharing. Screen recording failed to capture audio at first but was later fixed by simply adding each of the audio tracks to the stream object being captured as described in the Room Page section above. Screen sharing also only shared the stream to peers but did not display on the local client, this was also promptly fixed by replacing the local stream as well as the remote streams with the screen media track. The results from these tests can be seen in Appendix K. There were however, two limitations encountered.

Screen recording would not capture the audio stream of peers that connected after the recording started, this could be solved by updating the *onstream* function to handle this, but would require the recording function to be moved from the toolbar to the room component and passed as a property to the toolbar so the room component has access to the recording stream for updating. A second limitation was also encountered, where refreshing a room page would cause the other peers to lose the stream of the refreshing peer. Looking at the console logs, the stream event seemed to be missing a user ID after the refresh causing the issue. Although within the limited testing time the cause of the stream to be missing an ID could not be found. Leaving and re-joining the room fixes this issue.

# 4 Conclusions

This section will reflect on the outcomes and any conclusions that can be made from the project, as well as considering any changes that could be made to improve the final result and where the future of WebRTC technology is headed.

## *4.1 Changes*

Throughout the duration of this project a lot of choices were made between the design of the application and the implementation of the WebRTC event handling, to the tools and libraries used for development. A lot of time was spent with these tools and working on the design, as a result, flaws and areas for improvement were certainly found along the way.

### 4.1.1 Design and Implementation

In terms of issues with the design and implementation the biggest issue was with planning the approach to development. As was discussed previously, the development of the responsive grid and chat functionality were carried out in isolation, without any frameworks and was later rewritten for the React. This approach essentially required building the application twice. As mentioned in the Background section, the React JavaScript framework was used for its expandability and modularity. Due to its component based design, the various sections to the room page should have been developed as isolated components within React without the need to rewrite the same code logic twice. However, this was done due to uncertainty of the rules and algorithms to be implemented regarding the responsiveness as well as a lack of experience with React. This uncertainty on two different aspects of the problem meant developing it externally removing one of the aspects of uncertainty allowed the problem to be approach one step at a time. This could mostly have avoided through better preparation.

As a result of the aforementioned unfamiliarity with React, the code structure could also be improved to fit a more component based style. The use of Styled Components allowed for many of the areas of the room to be large, nested Styled Components. Which in this case worked, as the children of these components were generated by RTCMultiConnection events firing and not React logic. However, rather than inserting a styled div for each stream object, each could have been a React component with a stream source and a username and ID as stateful attributes and used to build each stream object. Similarly this could have been done with messages allowing for more control over messages and future expandability such as embedding links and images in the chat.

### 4.1.2 Tools and Libraries

The main frameworks and tools used on this project such as Express, React and coturn were all suitable for this project and as discussed, issues with these tools was largely down to lack of experience. However, the library used for interfacing with the WebRTC APIs, RTCMultiConnection, was suitable in the early stages of the project, however as it came into its later stages of development the limitations of the library began to show, especially when it came to screen sharing as some of the built in functions did not work and had to be rewritten manually. The Jitsi Meet application used for reference, open-sourced its WebRTC wrapper as lib-jitsi-meet and reviewing its documentation showed a very robust and more frequently maintained library. The founder of Jitsi who maintain the library, Emil Ivov, is also an active contributor to the IETF RFCs regarding WebRTC standards and so if the project was redone, the lib-jitsi-meet library would be the WebRTC wrapper of choice.

## *4.2 Further development*

With time to further develop the application, a number of other features could be added. One such feature being room moderation. This would require keeping a store of who is a room moderator both on the client and server. The client needs to know, as it should update the user's view to display moderation controls. However the server also needs to be aware, since a client could spoof the data to the signalling server to disconnect a user from all peers and so would need to be verified by the server before executing the function. The person to open the room would be set as the moderator with any other room moderators created by the existing moderator(s). If a moderator right clicks on a user's stream a menu could be displayed by creating a div with its top left corner at the mouse position, providing the user with a number of options such as disconnecting or muting a user. This could then send a custom event to the

signalling server to be verified and if allowed, emitted to the selected peer carrying out the requested function on their client.

Provided the use of a more component based design as discussed above, controls for the streams layout could also have been added, this would be easier to achieve treating the video grid as a separate component as quite a lot of logic would be involved with it. It would ideally allow a user to focus on a specific stream while pushing other streams to a list at the side of the window, or in grid mode be given the choice to maintain aspect ratio or crop streams.

Finally authorisation with various services could also have been added. Most online services support authorisation through OAuth 2.0. As it's the industry standard authorisation protocol there are also many JS libraries which implement the OAuth 2.0 authorisation flow for both the client and the server. Discussing the OAuth 2.0 authorisation flow is out of scope of this report however more can be read about it at https://oauth.net/2/. This authorisation could have been used to get a user's name from the service, removing the need to enter a username when joining a room. A scheduling system linked to the authorisation services could also be added given significantly more time.

## *4.3 Future of WebRTC*

Prior to the Covid 19 pandemic future developments in the area of WebRTC were focused on developing more efficient media codecs for more reliable operation in adverse network conditions or pushing the media quality higher in better conditions. However over the course of the pandemic the use WebRTC based applications accelerated and was deployed in many new use cases pushing the development of WebRTC further. It was no longer being used only for office meetings but now in areas such as healthcare, education, live arts events and more. The W3C published a document in November 2021 outlining some of these pushes for new innovations in WebRTC [31]. These include: support for streaming digital rights managed (DRM) content, simplifying the signalling process and reduced ICE checks for low power IoT applications, automatic connection reestablishment for long term connection applications and manipulating or processing media data before transmission, among many other suggestions. These innovations would allow for many new applications of WebRTC, such as in the areas of machine learning, WebRTC streaming from IoT devices, and media effects such as live captions and background removal.

# 5 References


[1] Central Statistics Office, “Pulse Survey - Our Lives Online - Remote Work”, Ireland, 2021. [Online]. Accessed: https://www.cso.ie/en/releasesandpublications/fp/fp-psolo/pulsesurvey-ourlivesonline-remoteworknovember2021/workingremotely/. [Accessed: May 01, 2022]

[2] H. Alvestrand, “Google release of WebRTC source code”, *w3.org*, 2011. [Online]. Available: https://lists.w3.org/Archives/Public/public-webrtc/2011May/0022.html. [Accessed: May 02, 2022].

[3] WebRTC, “Real-time communication for the web”, *webrtc.org*, 2022. [Online]. Available: https://webrtc.org/. [Accessed: May 01, 2022].

[4] *SDP: Session Description Protocol*, RFC 2327, April 1998. [Online]. Available: https://www.ietf.org/rfc/rfc2327.html

[5] *JavaScript Session Establishment Protocol (JSEP)*, RFC 8829, January 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8829.html.

[6] R. Manson, “A More Technical Introduction to Web-based Real-Time Communication,” in *Getting started with WebRTC*, Birmingham, UK: Packt Publishing, 2013.

[7] WebRTC, “Firebase + WebRTC Codelab”, *webrtc.org*, 2020. [Online]. Available: https://webrtc.org/getting-started/firebase-rtc-codelab. [Accessed: May 02, 2022].

[8] I. Grigorik, “WebSocket” in *High-performance browser networking*. Sebastopol, CA: O'Reilly, 2013.

[9] M. Luis and R. Vigil, “Network Address Translation (NAT) FAQ”, Cisco, 2020. [Online]. Available: https://www.cisco.com/c/en/us/support/docs/ip/network-address-translation-nat/26704-nat-faq-00.html. [Accessed: May 01, 2022].

[10] E. Ivov, “ICE always tastes better when it trickles!”, webrtcHacks, 2013. [Online]. Available: https://webrtchacks.com/trickle-ice/ [Accessed: May 01, 2022].

[11] *Session Traversal Utilities for NAT (STUN)*, RFC 8489, February 2020. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc5389.

[12] *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*, RFC 8656, February 2020. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8656.

[13] *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*, RFC 8445, July 2018. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc8445.

[14] I. Grigorik, “WebRTC” in *High-performance browser networking*. Sebastopol, CA: O'Reilly, 2013.

[15] Mozilla Developer Network, “Media Capture and Streams API (Media Stream)”, *mozilla.org*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Media_Streams_API. [Accessed: May 03, 2022].

[16] Mozilla Developer Network, “RTCPeerConnection”, *mozilla.org*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection. [Accessed: May 03, 2022].

[17] Mozilla Developer Network, “RTCDataChannel”, *mozilla.org*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/ RTCDataChannel. [Accessed: May 03, 2022].

[18] Ant Media, “WebRTC Server Explained: What is It? Everything You Need To Know!”, *antmedia.io*, 2021. [Online]. Available: https://antmedia.io/webrtc-servers. [Accessed: May 04, 2022].

[19] C. Söhlemann, "SSR or CSR - what is better for Progressive Web App?", *kruschecompany.com*, 2022. [Online]. Available: https://kruschecompany.com/ssr-or-csr-for-progressive-web-app. [Accessed: May 19, 2022].

[20] T. Krotoff, “Front-end frameworks popularity”, *github.com*, 2021. [Online]. Available: https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190. [Accessed: May 06, 2022].

[21] React, “Components and Props”, *reactjs.org*. [Online]. Available: https://reactjs.org/docs/components-and-props.html. [Accessed: May 06, 2022].

[22] S. Li, "React: Create maintainable, high-performance UI components", *ibm.com*, 2015. [Online]. Available: https://developer.ibm.com/tutorials/wa-react-intro. [Accessed: May 19, 2022].

[23] React, “State and Lifecycle”, *reactjs.org*. [Online]. Available: https://reactjs.org/docs/state-and-lifecycle.html. [Accessed: May 06, 2022].

[24] React, “Refs and the DOM”, *reactjs.org*. [Online]. Available: https://reactjs.org/docs/refs-and-the-dom.html. [Accessed: May 06, 2022].

[25] Mozilla Developer Network, “CSS prepocessor”, *mozilla.org*. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/CSS_preprocessor [Accessed: May 09, 2022]

[26] P. Kelly. (2021). Introduction to Gateway and NodeJS Environment [PowerPoint slides] Available: https://brightspace.tudublin.ie.

[27] Express, “Frameworks built on Express”, *expressjs.com*, 2017. [Online]. Available: https://expressjs.com/en/resources/frameworks.html. [Accessed: May 12, 2022].

[28] M. Mészáros *et al.*, *coturn Manpages*. (2021). [Online]. Available: https://manpages.debian.org/unstable/coturn/coturn.1.en.html. [Accessed: May 12, 2022]

[29] coturn, “coturn TURN server project”, *github.com*. [Online]. Available: https://github.com/coturn/coturn. [Accessed: May 12, 2022]

[30] J. Simmons, "Example of Nesting Flexbox and Grid", *jensimmons.com*, 2017. [Online]. Available: https://labs.jensimmons.com/2017/03-009.html. [Accessed: April 02, 2022].

[31] W3C, “WebRTC Next Version Use Cases”, *w3.org*, 2021. [Online]. Available: https://www.w3.org/TR/webrtc-nv-use-cases. [Accessed: May 16, 2022].

# 6 Appendices

## *Appendix A. Online Services Landing Pages*

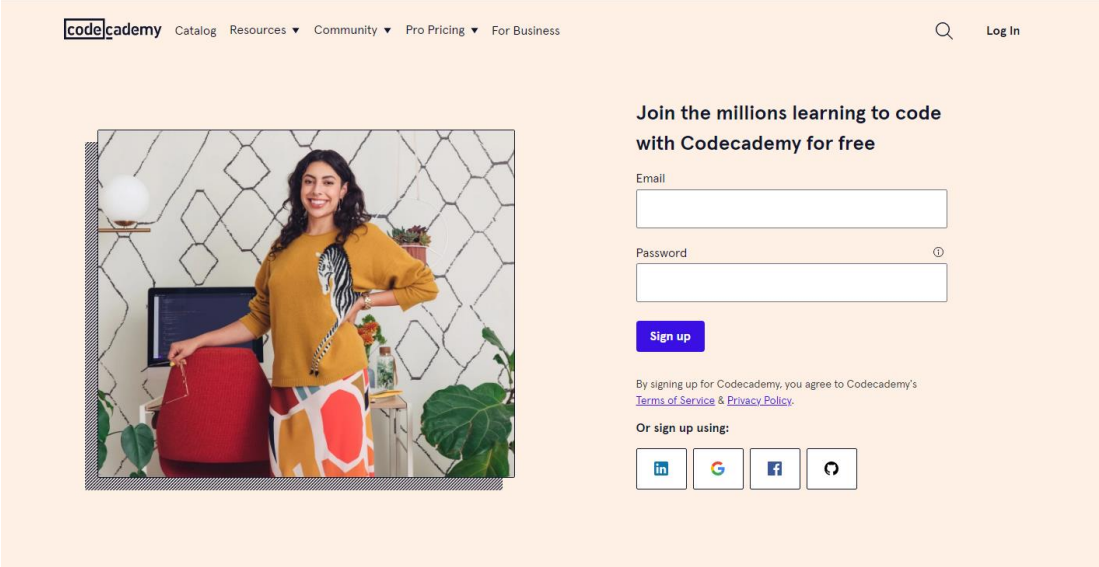



*Figure 14: Code Academy Landing Page*

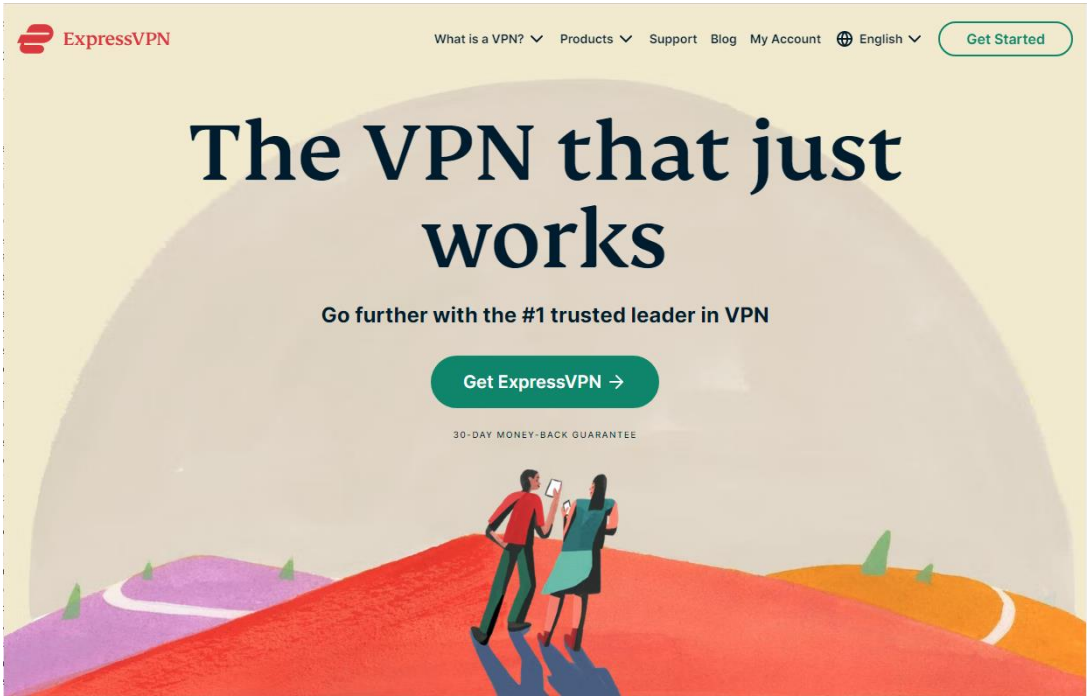



*Figure 15: ExpressVPN landing page*

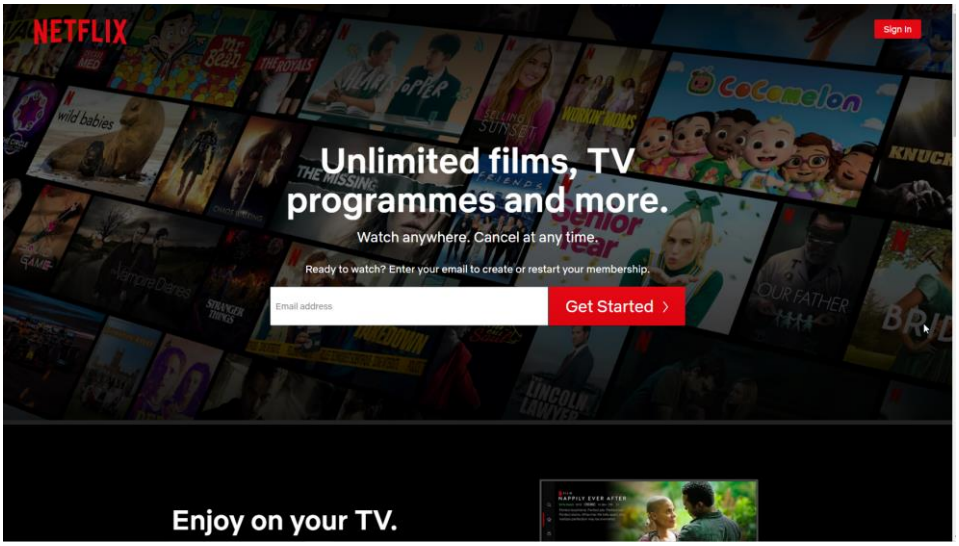



*Figure 16: Netflix landing page*

## *Appendix B. Server Implementation*

```
{
  "socketURL": "/",
  "socketMessageEvent": "RTCMultiConnection-Message",
  "socketCustomEvent": "RTCMultiConnection-Custom-Message",
  "enableLogs": "false",
  "autoRebootServerOnFailure": "false",
  "isUseHTTPs": "true",
  "sslKey": "./fake-keys/privatekey.pem",
  "sslCert": "./fake-keys/certificate.pem",
  "adminUserName": "username",
  "adminPassword": "password"
}      You, 3 months ago • express app to host rtcmulticonnec
```

*Figure 17: RTCMultiConnection server configuration file*

etc > turnserver.conf

```
listening-port=3478
tls-listening-port=5349
alt-listening-port=3479
alt-tls-listening-port=5350
verbose
fingerprint
lt-cred-mech
server-name=pieloaf.com
user=webrtcFYP:webrtcGuestPassword
realm=pieloaf.com
cert=/etc/letsencrypt/live/pieloaf.com/cert.pem
pkey=/etc/letsencrypt/live/pieloaf.com/privkey.pem
cipher-list="DEFAULT"
log-file=/var/log/coturn/turnserver.log
simple-log
TURNSERVER_ENABLED=1
```

*Figure 18: Turn server configuration file: turnserver.conf*

```
// importing required modules
const express = require('express');
const app = express();
const https = require('https');
const fs = require('fs');
const cors = require('cors');
const io = require('socket.io');
const RTCMultiConnectionServer = require('rtcmulticonnection-server');

const PORT = 9001; // setting port variable

// colour text for console logs
const BASH_COLORS_HELPER = RTCMultiConnectionServer.BASH_COLORS_HELPER;

// loading socket io settings from config file
var config = RTCMultiConnectionServer.
    getValuesFromConfigJson({ config: 'config.json' });

// setting bash terminal settings from config
config = RTCMultiConnectionServer.
    getBashParameters(config, BASH_COLORS_HELPER);

// setting server ssl certificate and key
const ServerOptions = {
    key: fs.readFileSync(config.sslKey),
    cert: fs.readFileSync(config.sslCert)
};

// used in the case of server sitting behind a proxy
app.enable('trust proxy');

// setting cors (cross origin resource sharing)
// used to allow requests from other domains
app.use(cors());

// setting express making static build directory public
app.use(express.static('build'));

// on any get request, send index.html from build directory
app.get('/*', (req, res) => {
    res.sendFile('index.html', { root: __dirname + '/build' });
})

// create the https server
var server = https.createServer(ServerOptions, app)

// passing server object and config to RTCMultiConnectionServer
RTCMultiConnectionServer.beforeHttpListen(server, config);

// start server lisening on port
server = server.listen(PORT, function () {
    // setting a lisener function to be handled by RTCMultiConnectionServer
    RTCMultiConnectionServer.afterHttpListen(server, config);
});

// on server error log the error
server.on('error', (err) => { console.log(err) })

// setting socket io to listen on server with cors and any origin
io(server, { cors: true, origins: '*:*' }).
    // on connection event
    on('connection', function (socket) {

        // add socket io object to RTCMultiConnectionServer
        RTCMultiConnectionServer.addSocket(socket, config);

        // allowing custom events to be emitted
        const params = socket.handshake.query;
        if (!params.socketCustomEvent) {
            params.socketCustomEvent = 'custom-message';
        }

        // on customs event
        socket.on(params.socketCustomEvent, function (message) {
            // broadcast to all clients the event type and message
            socket.broadcast.emit(params.socketCustomEvent, message);
        });
    });
```

*Figure 19: HTTPS and signalling server code*

## *Appendix C. Landing Page Layout and Planning*

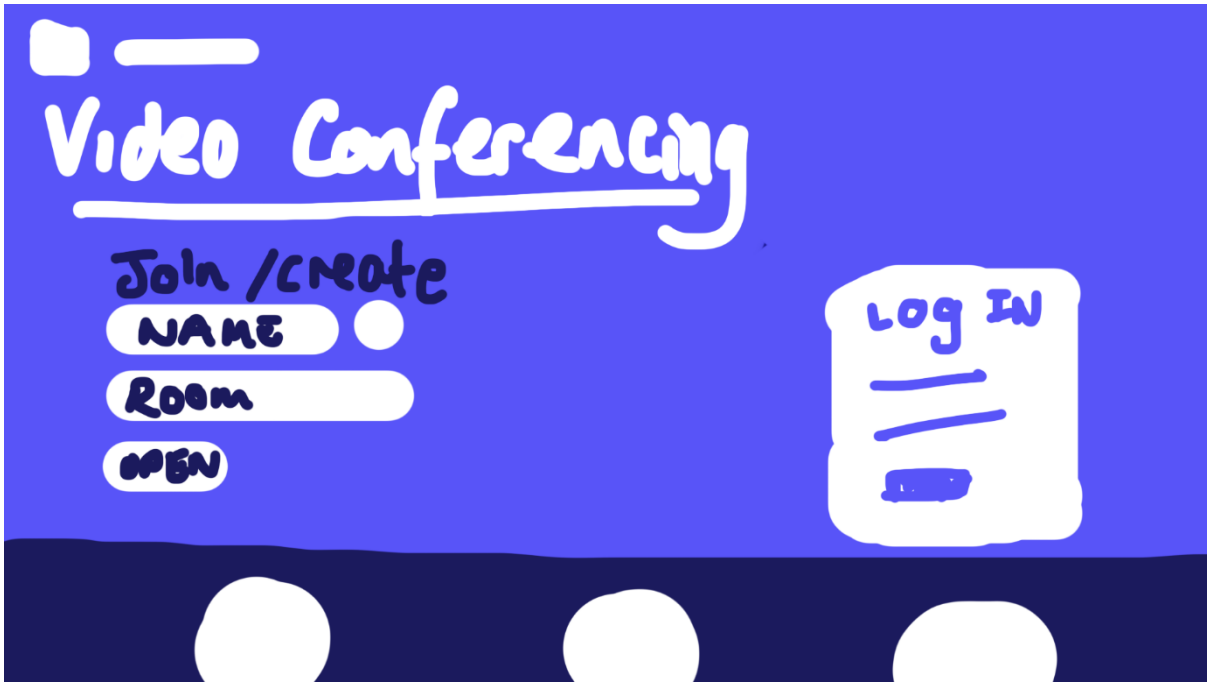



*Figure 20: Digital sketch for the landing page layout*

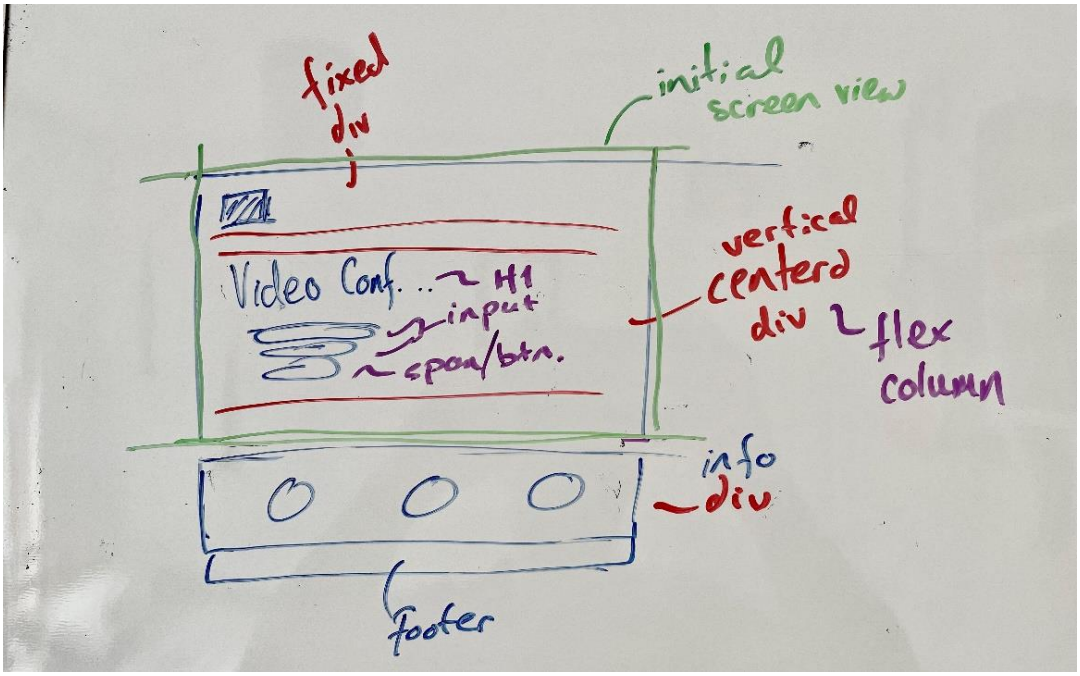



*Figure 21: Whiteboard notes breaking down the landing page layout into HTML components*

## *Appendix D. Room Page Layout*

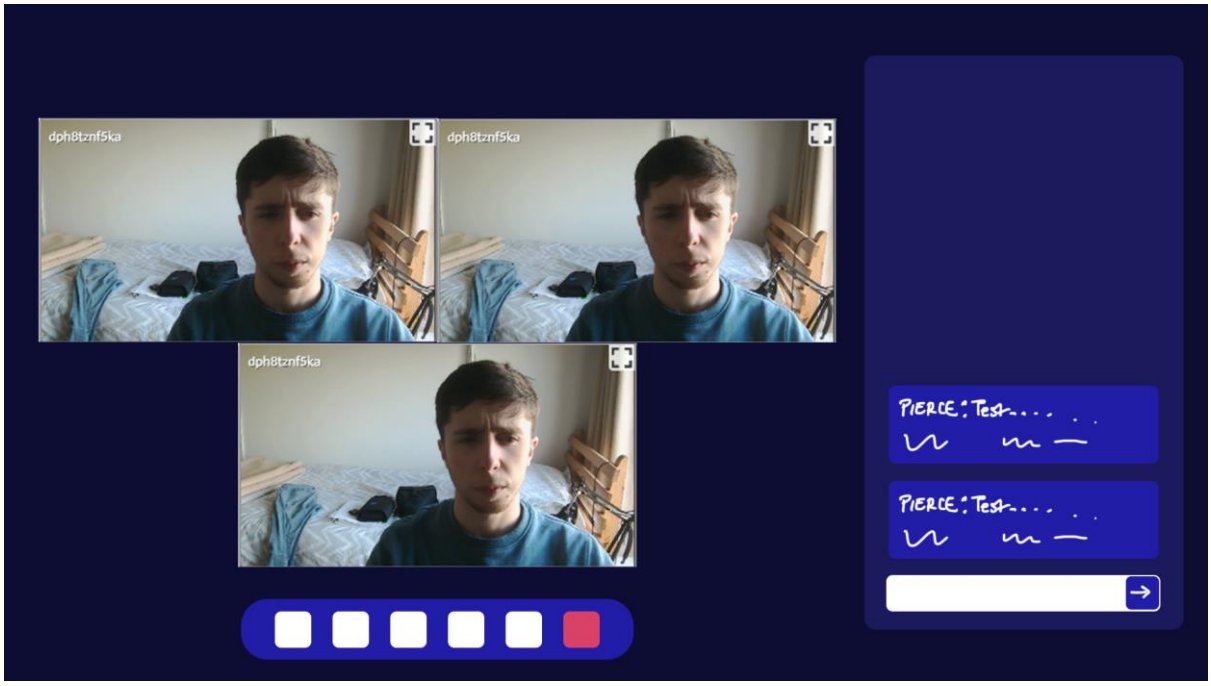



*Figure 22: Digital sketch for room page layout*

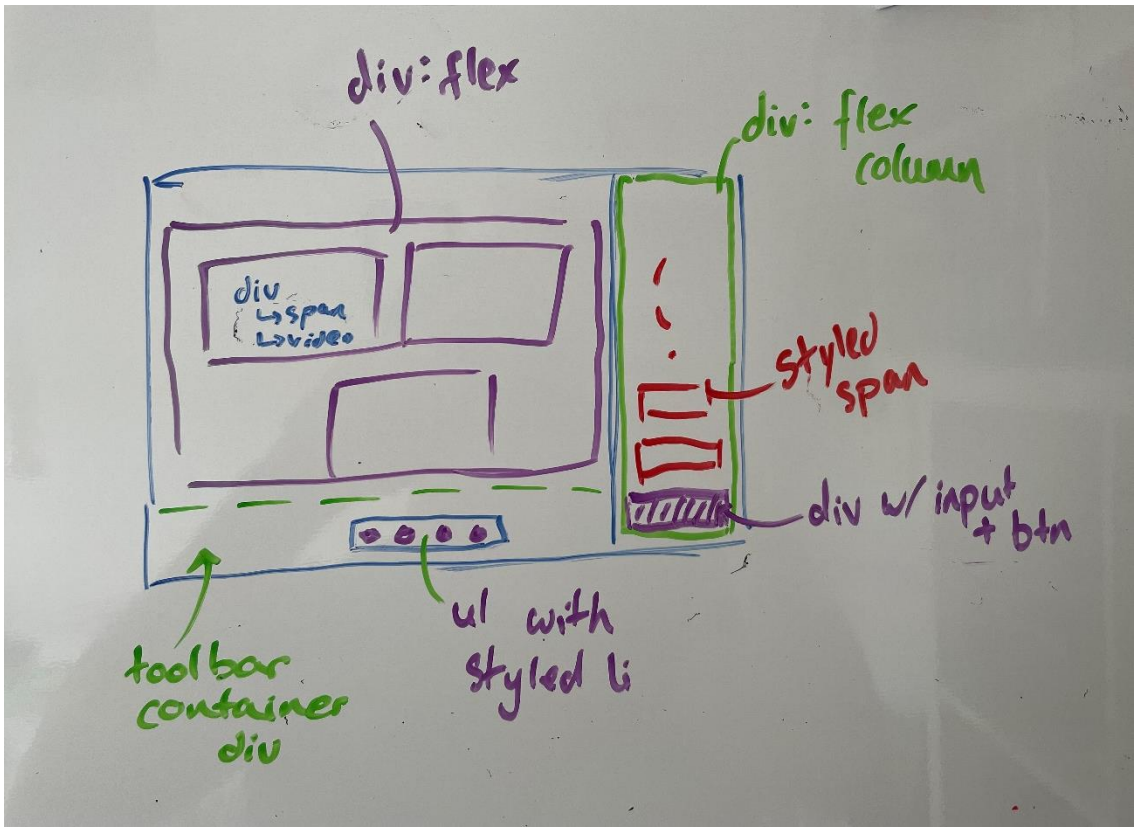



*Figure 23: Whiteboard notes breaking down the room page layout into HTML components*

## *Appendix E. Mixin and Global Styles*

```
import { createGlobalStyle } from "styled-components";
import theme from "./theme";

const GlobalStyle = createGlobalStyle`
    /* remove link high lighting */
    a {
        text-decoration: none;
        :visited,
        :-webkit-any-link {
            color: ${theme.colours.darkBlue};
        }
    }
    /* remove defualt list styles */
    ul
    {
        margin: 0;
        padding: 0;
    }

     li {
        list-style: none;
    }

    /* hide media controls for streams*/
    video::-webkit-media-controls {
        display: none;
    }
`
export default GlobalStyle;      You, 3 months ago • adds
```

*Figure 24: Global styles applied the entire application*

```
const theme = {
    colours: {
        lightBlue: `#5854f8`,
        darkBlue: `#1b1a5d`,
        teal: `#06d6a0`,
        white: `#f0eff4`,
        pink: `#da4167`,
        black: `#0d0d33`,
    },

    fonts: {
        primary: 'Circular Std, system, -apple-system, BlinkMacSystemFont, sans-serif',
    },

    fontSizes: {
        base: `16px`,
        xs: `12px`,
        sm: `14px`,
        md: `20px`,
        lg: `24px`,
        xl: `28px`,
        xxl: `32px`,
        title: `64px`
    },

    transition: `all 0.2s ease-in-out`,
}

export default theme;      You, 3 months ago • added styles …
```

*Figure 25: Style variables for consistent styling across the application*

```
import { css } from "styled-components";
import theme from "./theme";

const mixins = {
    fill: css`
        width: -webkit-fill-available;
        width: -moz-available;
    `,

    flexCentre: css`
    display: flex;
    align-items: center;
    justify-content: center;
`,

    interactive: css`
    border-radius: 61px;
    font-weight: bold;
    font-size: ${({ fSize }) => `${fSize || theme.fontSizes.lg}`};
    color: ${({ color }) => `${color || theme.colours.darkBlue}`};
    background-color: ${({ bgColor }) => `${bgColor || theme.colours.white}`};
    width: -webkit-fill-available;
    margin: 16px 32px 0;
    max-width: ${({ width }) => `${width || "fit-content"}`};
    transition: ${theme.transition};
    :hover{
        cursor: pointer;
        transform: scale(1.1);
    }
`,
}

export default mixins;
```

*Figure 26: Mixin styles for interactive elements as well as centring items and filling empty space*

## *Appendix F. Application Setup*

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import { BrowserRouter } from 'react-router-dom';

ReactDOM.render(
  <BrowserRouter>
    <React.StrictMode>
      <App />
    </React.StrictMode>
  </BrowserRouter>,
  document.getElementById('root')
);
```

*Figure 27: index.js: wrapping the application in a BrowserRouter tag to allow React Router to manage page routing*

```jsx
import './App.css';
import { Routes, Route } from 'react-router-dom';
import { Room } from './pages/room';
import { HomePage } from './pages/homepage';
import GlobalStyle from "./styles/globalStyle";

function App() {
  return (
    <div className="App">
      <GlobalStyle />
      <Routes>
        <Route exact path="/" element={<HomePage />} />
        <Route path="/room/:roomID" element={<Room />} />
      </Routes>
    </div>
  );
}

export default App;
```

*Figure 28: App.js: setting up React Router used to manage page routing on the client*

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <meta name="theme-color" content="#000000" />
    <meta name="description" content="WebRTC React App" />
    <link rel="apple-touch-icon" href="%PUBLIC_URL%/logo192.png" />
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json" />
    <title>WebRTC React App</title>
    <script src="https://github.com/muaz-khan/RTCMultiConnection/releases/download/3.7.0/RTCMultiConnection.min.js"></script>
    <script src="https://www.webrtc-experiment.com/RecordRTC.js"></script>

    <script
      src="https://cdnjs.cloudflare.com/ajax/libs/socket.io/4.4.1/socket.io.js"
      integrity="sha512-MgkNs0gNdrnOM7k+0L+wgiRc5aLgl74sJQKbIWegVIMvVGPc1+gc1L2oK9Wf/D9pq58eqIJAxOonYPVE5UwUFA=="
      crossorigin="anonymous"
      referrerpolicy="no-referrer"
    ></script>
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root"></div>
  </body>
</html>
```

*Figure 29: index.html: boiler plate HTML page to render the app within. RTCMultiConnection library is imported here as it is a native DOM library*

# *Appendix G. Landing Page Implementation*

```
74  export const HomePage = () => {
75      // room stateful variables
76      const [Name, setName] = useState("");
77      const [roomID, setRoomID] = useState("");
78      const navigate = useNavigate();
79
80      // read room id from input
81      const getRoomID = () => {
82          let id;
83
84          // store read room id, if no room id then generate a random id
85          id = roomID.length ? roomID : Math.random().toString(36).substring(2)
86
87          // if name entered store in session storage
88          if (Name.length) {
89              sessionStorage.setItem(id, JSON.stringify({
90                  "name": Name
91              }));
92          }
93          // return room id
94          return id;
95      }
96
97      const handleJoin = useCallback(() => {
98          // redirect to room page passing name and id as props
99          navigate(`/room/${getRoomID()}`);
100     });
101
102
```

*Figure 30: Landing page logic*

```
103     return (
104         <LandingContainer>
105             <NavBar />
106             <Body>
107                 <Title>Start a Meeting!</Title>
108                 <StyledInput
109                     placeholder="Name"
110                     width="250px" // override default width
111                     onChange={(e) => setName(e.target.value)} />
112                 <StyledInput
113                     placeholder="Room ID"
114                     width="350px" // override default width
115                     onChange={(e) => setRoomID(e.target.value)} />
116                 <StyledButton
117                     bgColor={theme.colours.teal} // override background color
118                     onClick={handleJoin}>Join Meeting</StyledButton>
119             </Body>
120             <Footer> Pierce Lowe - DT021A - Final Year Project - 2022</Footer>
121         </LandingContainer >
122     );
123
124 }
```

*Figure 31: Landing page layout with Styled Components*

```
import React, { useState, useCallback } from "react";
import { useNavigate } from "react-router-dom";
import styled from "styled-components";
import { NavBar } from "../components/Navbar";
import mixins from "../styles/mixins";
import theme from "../styles/theme";

// Styled Components used in the Landing Page
const LandingContainer = styled.div`
    display: flex;
    flex-direction: column;
    background-color: #5854f8;
    color: #F0EFF4;
    height: 100vh;
`;

const Body = styled.div`
    flex-direction: column;
    justify-content: center;
    align-items: flex-start;
    , a{
        text-decoration: none;
    }
    height: 100vh;
    padding: 20px;
    display: flex;
`;

const Title = styled.span`
    padding: 2rem;
    flex-wrap: wrap;
    font-size: ${theme.fontSizes.title};
    font-weight: bold;
    text-align: left;
    text-decoration: underline;
    @media screen and (max-width: 499px) {
        font-size: 42px;
    }
`;

const StyledButton = styled.button`
    ${mixins.flexCentre}
    ${mixins.interactive}
    padding: 0.5rem 1.5rem;
    transition: ${theme.transition}
    border: none;
    :hover{
        cursor: pointer;
        box-shadow: 0px 0px 6px 1px #1b1a5d;
    }
`;

const StyledInput = styled.input`
    ${mixins.flexCentre}
    ${mixins.interactive}
    : hover{
        transform: none;
        cursor: text;
    }
    justify-content: center;
    align-items: center;
    padding: 0.5rem 2rem;
    border: none;
`;

const Footer = styled.div`
    ${mixins.flexCentre}
    padding: 32px 0;
    font-size: ${theme.fontSizes.sm};
    background-color: ${theme.colours.black};
    color: ${theme.colours.white};
    ${mixins.fill};
`;
```

*Figure 32: Landing page imports and Styled Components*

# *Appendix H. Toolbar Component Implementation*

```jsx
import React, { useEffect, useState } from "react";
import { FaShare, FaCamera, FaMicrophone, FaPhone, FaFacebookMessenger, FaCircle } from "react-icons/fa";
import { MdScreenShare } from "react-icons/md";
import styled from "styled-components";
import theme from "../styles/theme";
import mixins from "../styles/mixins";
import { useNavigate } from "react-router-dom";

const StyledToolbar = styled.div`
    background-color: ${theme.colours.darkBlue};
    border-radius: 32px;
    width: fit-content;
    padding: 0 16px;
    transition: ${theme.transition};
    box-shadow: 0 0 9px 0px #ffffff78;
    ul {
        ${mixins.flexCentre};
        flex-direction: row;
    }
`;

const ToolbarContainer = styled.div`
    width: 100%;
    display: flex;
    justify-content: center;
    position: fixed;
    z-index: 1;
    padding-bottom: 24px;
    bottom: 0;
    ${StyledToolbar} {
        transform: translateY(200%);
    }
    &:hover ${StyledToolbar} {
        transform: translateY(0);
    }
`;

    You, 3 months ago • Initial Room Code …
const ToolbarItem = styled.li`
    ${mixins.flexCentre};
    ${mixins.interactive}
    padding: 8px;
    margin: 8px;

    transition: transform 0.1s ease-in-out;
    :hover {
        box-shadow: 0px 0px 3px 1px ${theme.colours.black};
    }
`;
```

*Figure 33: Toolbar imports and Styled Components*

```
    return (
        <ToolbarContainer>
            <StyledToolbar >
                <ul>
                    <ToolbarItem onClick={() => {
                        navigator.clipboard.writeText(window.location.href);
                    }}>
                        <FaShare />
                    </ToolbarItem>
                    <ToolbarItem onClick={toggleCam} color={states.cam ? theme.colours.darkBlue : theme.colours.pink}>
                        <FaCamera />
                    </ToolbarItem>
                    <ToolbarItem onClick={toggleMic} color={states.mic ? theme.colours.darkBlue : theme.colours.pink}>
                        <FaMicrophone />
                    </ToolbarItem>
                    <ToolbarItem onClick={toggleChat}>
                        <FaFacebookMessenger />
                    </ToolbarItem>
                    {(() => {
                        if (!isMobile) {
                            return (<>
                                <ToolbarItem onClick={toggleScreen} color={states.screen ? theme.colours.pink : theme.colours.darkBlue}>
                                    <MdScreenShare />
                                </ToolbarItem>
                                <ToolbarItem onClick={toggleRecording} color={recording ? theme.colours.pink : theme.colours.darkBlue}>
                                    <FaCircle />
                                </ToolbarItem>
                            </>)
                        }
                    })()}
                    <ToolbarItem color={theme.colours.pink} onClick={() => { navigate("/"); }}>
                        <FaPhone />
                    </ToolbarItem>
                </ul>
            </StyledToolbar>
        </ToolbarContainer >
    )
}
```

*Figure 34: Toolbar layout with Styled Components*

```
export const Toolbar = ({ connection, toggleCam, toggleMic, states, toggleChat, toggleScreen }) => {

    // stateful variables
    const [recording, setRecording] = useState(null);
    const [isMobile, setIsMobile] = useState(false);
    const navigate = useNavigate();

    const toggleRecording = async () => {
        let msg;
        const startRecording = async () => {
            // request user screen media
            window.navigator.mediaDevices.getDisplayMedia({ video: true }).
                then(function (stream) {
                    // for each remote and local stream get the audio track
                    connection.streamEvents.selectAll().forEach((streamEvt) => {
                        // add the audio track to the stream for recording
                        stream.addTrack(streamEvt.stream.getAudioTracks()[0]);
                    });

                    // init recorder object and start recording
                    let recorder = window.RecordRTC(stream, {
                        type: 'video',
                        mimeType: 'video/mp4',
                    });
                    setRecording([recorder, stream]);
                    recorder.startRecording();

                    // if the user stops streaming through a browser interaction
                    stream.getVideoTracks()[0].onended = function (e) {
                        // call end recording function
                        endRecording(recorder, stream);
                    }
                    // notify peers recording has started
                    msg = `System: ${connection.extra.name} has started recording`
                    connection.socket.emit("recording-status", msg);

                }).catch(function (error) {
                    // log error
                    console.log(error);
                });
        };

        const endRecording = (recorder = null, stream = null) => {
            // if recorder and stream are not passed, read from state
            if (!recorder && !stream) {
                recorder = recording[0];
                stream = recording[1];
            }
            // stop recording
            recorder.stopRecording(function () {
                // create blob from recorded data
                let blob = recorder.getBlob();
                // create file from blob and open user file dialog to save
                window.invokeSaveAsDialog(blob, connection.sessionid + '.mp4');
            });

            // only stop video tracks
            // stopping audio tracks stops them in the call as well
            stream.getVideoTracks()[0].stop();
            // reset state
            setRecording(null);
            // notify peers recording has ended
            msg = `System: ${connection.extra.name} has stopped recording`
            connection.socket.emit("recording-status", msg);
        }

        // depending on state, start or end recording
        !recording ? startRecording() : endRecording();
    };

    // https://abdessalam.dev/blog/detect-device-type-javascript/
    // check if device is mobile on mount
    useEffect(() => {
        // get user agent
        const ua = navigator.userAgent;

        // parse with regex for device type
        if (/(tablet|ipad|playbook|silk)|(android(?!.*mobi))/i.test(ua)) {
            setIsMobile(true);
        }
        else if (
            /Mobile|iP(hone|od)|Android|BlackBerry|IEMobile|Kindle|Silk-Accelerated|(hpw|web)OS|Opera M(o
                ua
            )
        ) {
            setIsMobile(true);
        }
    }, []);
```

*Figure 35: Toolbar initialisation and meeting recording logic*

# *Appendix I. Room Page Implementation*

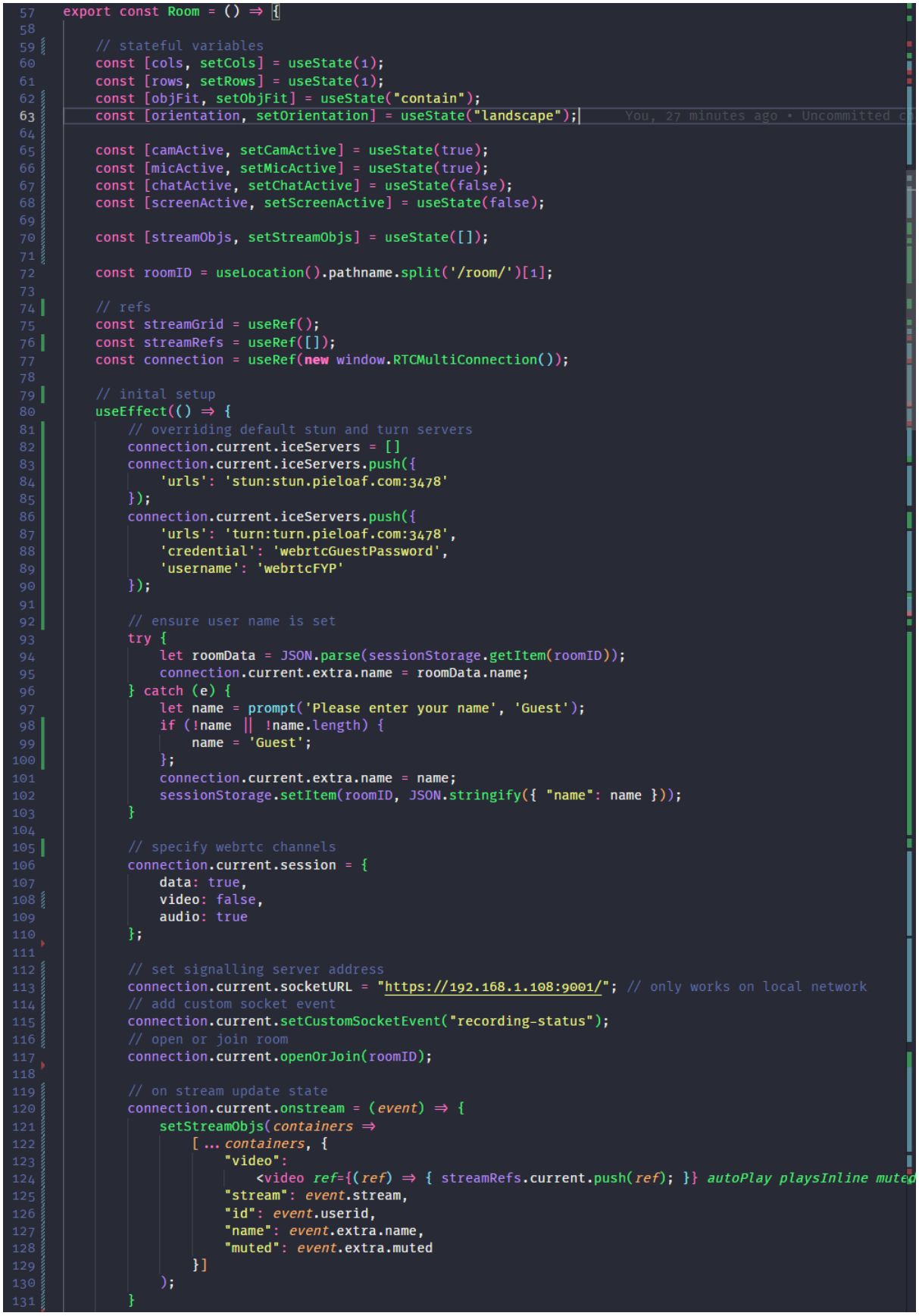
```
import React, { useEffect, useState, useRef } from "react";
import styled from "styled-components";
import { Toolbar } from "../components/toolbar";
import { ChatArea } from "../components/messages";
import { useLocation } from "react-router-dom";
import mixins from "../styles/mixins";
import theme from "../styles/theme";
import noCam from "../data/images/noCam.jpg";

const MainContainer = styled.div`
    display: flex;
    width: 100vw;
    height: 100vh;
    background-color: ${theme.colours.black};
`;

const VideoGrid = styled.div`
    display: grid;
    grid-gap: 10px;
    padding: 10px;
    ${mixins.fill};
    height: -webkit-fill-available;
    grid-template-columns: ${({ cols }) => '1fr '.repeat(cols)};
    grid-template-rows: ${({ rows }) => '1fr '.repeat(rows)};
    place-items: center;
`;

const VideoContainer = styled.div`
    width: 100%;
    height: 100%;
    overflow: hidden;
    position: relative;
    border-radius: 7px;
    > span{
        position: absolute;
        top: 0;
        left: 0;
        color: ${theme.colours.white};
        font-size: ${theme.fontSizes.xl};
        font-weight: bold;
        padding: 5px;
        background-color: #000000ea;
        border-radius: 7px;
        opacity: 0;
        transition: ${theme.transition}
    };
    :hover > span{
        opacity: 1;
    }
    > video {
        object-fit: ${props => props.fit};
        width: 100%;
        height: 100%;
    }
`;
```

*Figure 36: Room page imports and Styled Components*



```
export const Room = () => {

    // stateful variables
    const [cols, setCols] = useState(1);
    const [rows, setRows] = useState(1);
    const [objFit, setObjFit] = useState("contain");
    const [orientation, setOrientation] = useState("landscape");      You, 27 minutes ago • Uncommitted c

    const [camActive, setCamActive] = useState(true);
    const [micActive, setMicActive] = useState(true);
    const [chatActive, setChatActive] = useState(false);
    const [screenActive, setScreenActive] = useState(false);

    const [streamObjs, setStreamObjs] = useState([]);

    const roomID = useLocation().pathname.split('/room/')[1];

    // refs
    const streamGrid = useRef();
    const streamRefs = useRef([]);
    const connection = useRef(new window.RTCMultiConnection());

    // inital setup
    useEffect(() => {
        // overriding default stun and turn servers
        connection.current.iceServers = []
        connection.current.iceServers.push({
            'urls': 'stun:stun.pieloaf.com:3478'
        });
        connection.current.iceServers.push({
            'urls': 'turn:turn.pieloaf.com:3478',
            'credential': 'webrtcGuestPassword',
            'username': 'webrtcFYP'
        });

        // ensure user name is set
        try {
            let roomData = JSON.parse(sessionStorage.getItem(roomID));
            connection.current.extra.name = roomData.name;
        } catch (e) {
            let name = prompt('Please enter your name', 'Guest');
            if (!name || !name.length) {
                name = 'Guest';
            };
            connection.current.extra.name = name;
            sessionStorage.setItem(roomID, JSON.stringify({ "name": name }));
        }

        // specify webrtc channels
        connection.current.session = {
            data: true,
            video: false,
            audio: true
        };

        // set signalling server address
        connection.current.socketURL = "https://192.168.1.108:9001/"; // only works on local network
        // add custom socket event
        connection.current.setCustomSocketEvent("recording-status");
        // open or join room
        connection.current.openOrJoin(roomID);

        // on stream update state
        connection.current.onstream = (event) => {
            setStreamObjs(containers =>
                [ ...containers, {
                    "video":
                        <video ref={(ref) => { streamRefs.current.push(ref); }} autoPlay playsInline muted
                    "stream": event.stream,
                    "id": event.userid,
                    "name": event.extra.name,
                    "muted": event.extra.muted
                }]
            );
        }
```

*Figure 37: Stateful variable and reference initialisation, WebRTC connection setup and onstream event handler*

```
132
133         // onuser leave update state
134         connection.current.onleave = (event) => {
135             // find stream and remove from state array
136             setStreamObjs(streamObjs => streamObjs.filter(obj => obj.id !== event.userid));
137         };
138
139         // on mute remove stream
140         connection.current.onmute = (event) => {
141             if (event.muteType === 'video') {
142                 // find stream and remove src
143                 streamRefs.current.forEach(ref => {
144                     if (ref.id === event.userid) {
145                         ref.srcObject = null;
146                     }
147                 });
148             }
149         }
150
151         // onunmute replace stream
152         connection.current.onunmute = (event) => {
153             if (event.muteType === 'video') {
154                 // find stream and add src
155                 streamRefs.current.forEach(ref => {
156                     if (ref.id === event.userid) {
157                         ref.srcObject = event.stream;
158                     }
159                 });
160             }
161         }
162
163     }, [roomID]);
164
165     useEffect(() => {
166         // on new stream,
167         streamObjs.forEach((stream, index) => {
168             // add user id attribute
169             if (!streamRefs.current[index].id)
170                 streamRefs.current[index].id = stream.id;
171
172             //add src to video element if not muted on join
173             if (stream.muted === false) {
174                 streamRefs.current[index].srcObject = stream.stream;
175             }
176         })
177     }, [streamObjs]);
178
```

*Figure 38: onleave, onmute and onunmute event handlers, with a useEffect hook for streamObjs array updates to add new stream sources and user ids to elements*

```
    useEffect(() => {
        // resizing grid on new stream added
        // store cols and rows in temp variables
        let tempCols = cols;
        let tempRows = rows;

        // check if cols should decrememnt
        if (streamObjs.length <= (tempCols - 1) ** 2) {
            if (tempCols - 1 > 0) tempCols--;
        }
        // check if cols should increment
        else if (streamObjs.length > tempCols ** 2) tempCols++;

        // calculate rows
        for (let i = 1; i <= tempCols; i++) {
            if (streamObjs.length <= i * tempCols) {
                tempRows = i;
                break;
            }
        }

        // if temp and state not equal update state to rerender
        if (cols !== tempCols || rows !== tempRows) {
            setCols(tempCols);
            setRows(tempRows);

            // calculate if orientation should change
            let rect = streamGrid.current.getBoundingClientRect();
            let width = rect.width / 4;
            let height = rect.height / 3;

            let portrait = width < height;
            setOrientation(portrait ? "portrait" : "landscape");

            // check if should crop streams
            if ((!portrait && (rect.width / tempCols) < (rect.height / tempRows))
                || (portrait && (rect.width / tempRows) < (rect.height / tempCols))) {
                setObjFit("cover");
            } else {
                setObjFit("contain");
            }
        }
    }, [streamObjs, cols, rows]);

    useEffect(() => {
        // resizing grid on window resize
        const resize = () => {
            // check if orientation should change
            let rect = streamGrid.current.getBoundingClientRect();
            let width = rect.width / 4;
            let height = rect.height / 3;
            let portrait = width < height
            setOrientation(portrait ? "portrait" : "landscape");

            // check if should crop streams
            if ((!portrait && (rect.width / cols) < (rect.height / rows))
                || (portrait && (rect.width / rows) < (rect.height / cols))) {
                setObjFit("cover");
            } else {
                setObjFit("contain");
            }
        }

        // add event listener for resize
        window.addEventListener('resize', resize);

        // remove listener on unmount
        return () => window.removeEventListener('resize', resize);
    }, [cols, rows]);

```

*Figure 39: Responsive grid updating on window resize and on new streams added*

```
248
249     useEffect(() => {
250         // set camera muted to camActive state
251         connection.current.extra.muted = !camActive
252     }, [camActive]);
253
254     useEffect(() => {
255         return () => {
256             // on leaving room:      You, 32 minutes ago • Uncommitted changes
257             // disconnect with all users
258             connection.current.getAllParticipants().forEach(function (pid) {
259                 connection.current.disconnectWith(pid);
260             });
261
262             // stop all local cameras
263             connection.current.attachStreams.forEach(function (localStream) {
264                 localStream.stop();
265             });
266
267             // close socket.io connection
268             connection.current.closeSocket();
269         };
270     }, []);
271
272
273
274     const screenShare = () => {
275         const startScreenShare = () => {
276             // request screen media from user
277             window.navigator.mediaDevices.getDisplayMedia({ video: true }).then(stream => {
278                 // get video track
279                 let track = stream.getVideoTracks()[0];
280                 // set callback for track ended
281                 track.onended = stopScreenShare;
282                 // for each peer connected to client
283                 connection.current.peers.forEach(function (peer) {
284                     peer.peer.getSenders().forEach(function (sender) {
285                         // replace video track with screen share
286                         if (sender.track.kind === 'video') {
287                             sender.replaceTrack(track);
288                         }
289                     });
290                 });
291                 // update local view
292                 let localStream = streamObjs.find(obj => obj.id === connection.current.userid);
293                 let idx = streamObjs.indexOf(localStream);
294                 streamRefs.current[idx].srcObject = stream;
295                 setScreenActive(true); // update state
296             }).catch(err => {
297                 console.log(err); // log error
298                 setScreenActive(false); // reset state
299             });
300         }
301
302         const stopScreenShare = () => {
303             // get local camera stream
304             let localStream = streamObjs.find(obj => obj.id === connection.current.userid);
305             // for each peer connected to client
306             connection.current.peers.forEach(function (peer) {
307                 peer.peer.getSenders().forEach(function (sender) {
308                     // stop screen track and replace with camera track
309                     if (sender.track.kind === 'video') {
310                         sender.replaceTrack(localStream.stream.getVideoTracks()[0]);
311                         sender.track.stop();
312                     }
313                 });
314             });
315             // update local view
316             let idx = streamObjs.indexOf(localStream);
317             streamRefs.current[idx].srcObject = localStream.stream;
318             setScreenActive(false); // update state
319         }
320
321         // check if should start or stop screen share
322         if (screenActive) {
323             stopScreenShare();
324         } else {
325             startScreenShare();
326         }
327     }
328
329
```

*Figure 40: Handling clean shut down when leaving a meeting, start screen recording and stop screen recording*

```
// get track of given "kind"
const getLocalTrack = (kind) => {
    if (kind === "video") return connection.current.attachStreams[0].getVideoTracks();
    else if (kind === "audio") return connection.current.attachStreams[0].getAudioTracks();
}

return (
    <MainContainer>

        <VideoGrid ref={streamGrid}
            cols={orientation === "portrait" ? rows : cols}
            rows={orientation === "portrait" ? cols : rows}>
            {/* toolbar function props */}
            <Toolbar
                toggleCam={() => {
                    setCamActive(!camActive);
                    let stream = connection.current.attachStreams[0]
                    getLocalTrack("video")[0].enabled ? stream.mute("video") : stream.unmute("video");
                }}
                toggleMic={() => {
                    setMicActive(!micActive);
                    let stream = connection.current.attachStreams[0]
                    getLocalTrack("audio")[0].enabled ? stream.mute("audio") : stream.unmute("audio");
                }}
                states={{
                    cam: camActive,
                    mic: micActive,
                    screen: screenActive
                }}
                toggleScreen={screenShare}
                toggleChat={() => { setChatActive(!chatActive) }}
                connection={connection.current} />

            {/* mapping streams into video containers */}
            {streamObjs.map(streamObj =>
                <VideoContainer key={streamObj.id} fit={objFit}>
                    <span>{streamObj.name}</span>
                    {streamObj.video}
                </VideoContainer>
            )}
        </VideoGrid>

        <ChatArea connection={connection.current} state={chatActive} />

    </MainContainer >
)
```

*Figure 41: Helper function to get local media tracks and Room layout with Styled Components*

# *Appendix J. Chat Component Implementation*

```
import React, { useEffect, useState, useRef } from "react";
import styled from "styled-components";
import mixins from "../styles/mixins";
import theme from "../styles/theme";

// ewww this is very ugly nested styled component madness 🤢
const ChatContainer = styled.div`
    margin-left: auto;
    display: ${({ show }) => show ? "flex" : "none"};
    flex-direction: column;
    width: 450px;
    min-width: 200px;
    height: 100%;
    align-items: center;
    justify-content: flex-end;
    background-color: ${theme.colours.darkBlue};
    z-index: 2;
    position: relative;
    @media screen and (max-width: 960px) {
        position: absolute;
        left: 0;
        width: 100vw;
    }
    > div {
        padding-bottom: 10px;
        width: -webkit-fill-available;
        display: flex;
        > * {
            border-radius: 7px;
            border: none;
            outline: none;
            : focus {
                outline-style: solid;
                outline-width: 2px;
                outline-color: ${theme.colours.lightBlue};
            }
        }
        > input {
            margin: 10px;
            height: 32px;
            width: 100%;
        }

    }
`;
```

*Figure 42: Chat area component imports and main chat container styled component*

```
const CloseBtn = styled.h3`
        ${mixins.interactive}
        padding: 0 7px 2px;
        border-radius: 5px;
        position: absolute;
        top: 0;
        left: 0;
        margin: 5px;
        : hover {
            background-color: ${theme.colours.pink};
            color: ${theme.colours.white};
        }
`


const MessageBox = styled.span`
    text-align: left;
    width: -webkit-fill-available;
    word-break: break-word;
    padding: 10px;
    margin: 10px 10px 0;
    border-radius: 7px;
    background-color: ${theme.colours.lightBlue};
    color: ${theme.colours.white};
    font-size: ${theme.fontSizes.md};
`;

const SendBtn = styled.button`
    ${mixins.interactive}
    margin: 10px;
    margin-left: auto;
    background-color: ${theme.colours.teal};
`;
```

*Figure 43: Other Styled Components used in the messages component*

```jsx

export const ChatArea = ({ connection, state }) => {
    const [message, setMessage] = useState('');
    const [messages, setMessages] = useState([]);
    const [chatActive, setChatActive] = useState(false);
    const msgInput = useRef();

    // set initial chat state
    useEffect(() => {
        setChatActive(state);
    }, [state]);

    connection.onmessage = (event) => {
        // update message array on new message
        setMessages((msgs, i) => [...msgs, <MessageBox key={i}>
            {event.data.author}: {event.data.content}
        </MessageBox>
        ]);
    }

    useEffect(() => {
        if (!connection.socket) return;
        // send message on start recording event from server
        connection.socket.on("recording-status", function (message) {
            setMessages((msgs, i) => [...msgs, <MessageBox key={i}>
                {message}
            </MessageBox>
            ]);
        });
    }, [connection.socket]);

    const sendMessage = () => {
        if (!message) return // return if empty message
        // send message content and author on RTCDataChannel
        connection.send({ content: message, author: connection.extra.name });
        // update messages array with personal message
        setMessages((msgs, i) => [...msgs,
        <MessageBox key={i}>
            You: {message}
        </MessageBox>
        ]);
        // reset message input
        msgInput.current.value = '';
    }

    return (
        <ChatContainer show={chatActive} >
            <CloseBtn onClick={() => setChatActive(false)}>X</CloseBtn>
            {messages}
            <div>
                <input type="text"
                    placeholder="Message ... "
                    onChange={(e) => { setMessage(e.target.value) }}
                    ref={msgInput} />
                <SendBtn onClick={sendMessage}>Send</SendBtn>
            </div>
        </ChatContainer>
    )

}
```

*Figure 44: Chat area component logic and element layout*

## *Appendix K. Test Results and Final Application*

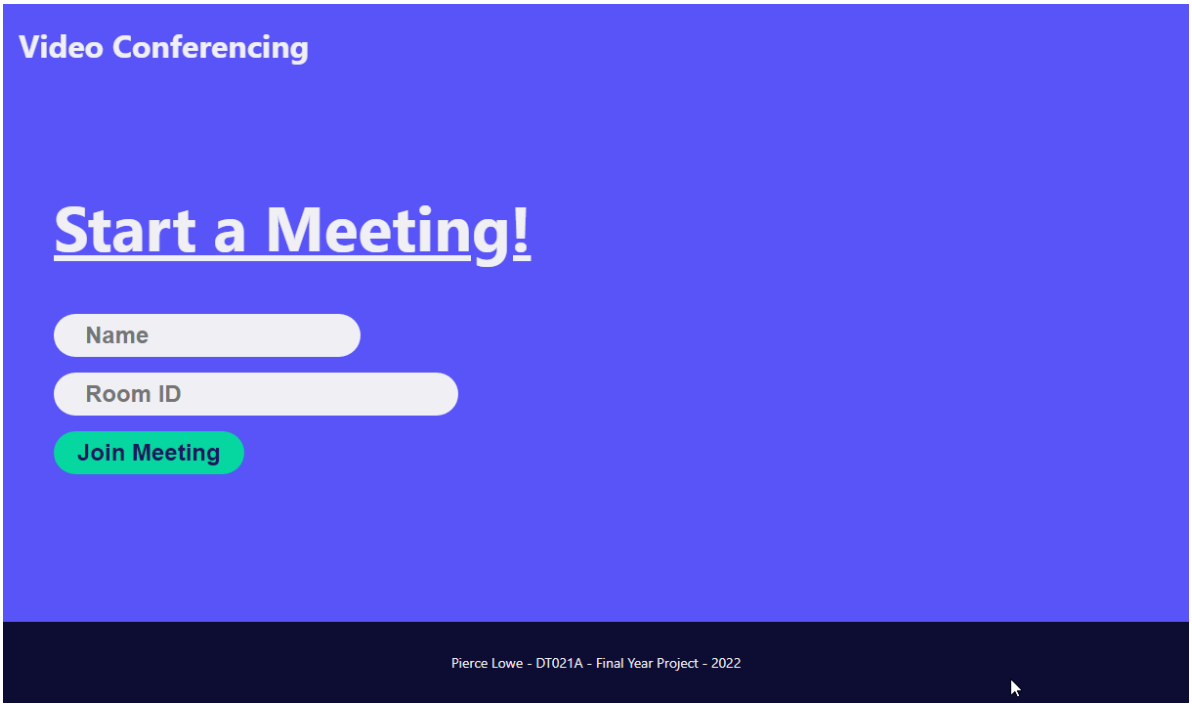



*Figure 45: Landing Page (desktop)*

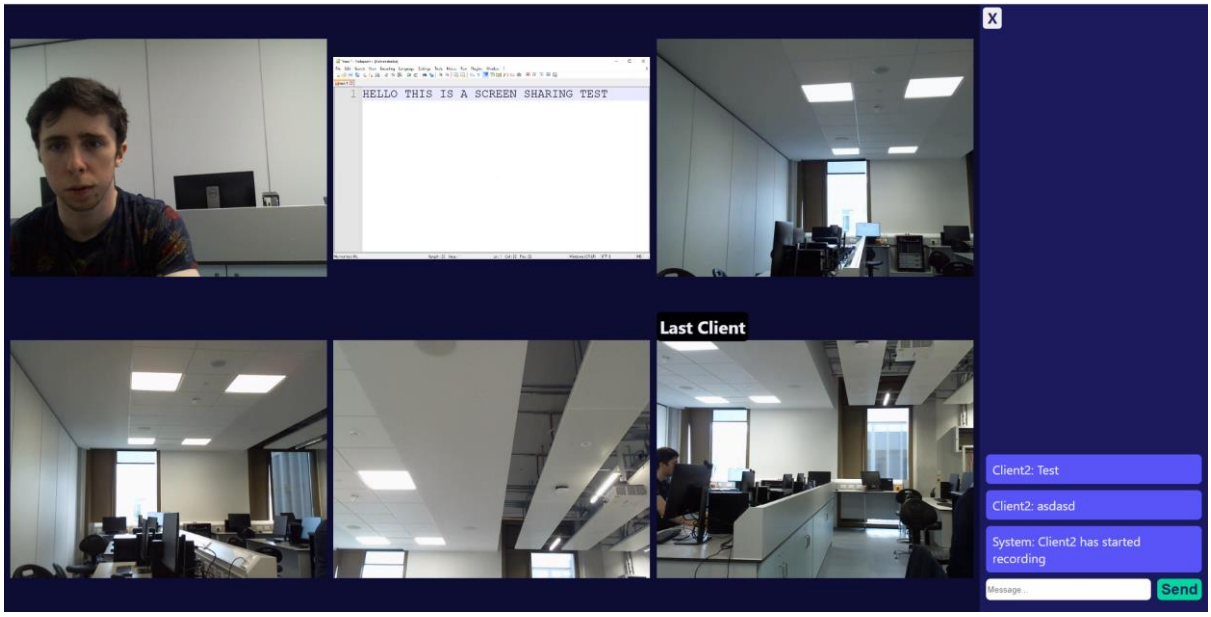



*Figure 46: Room Page with 6 clients testing screen recording, messaging and screen sharing*

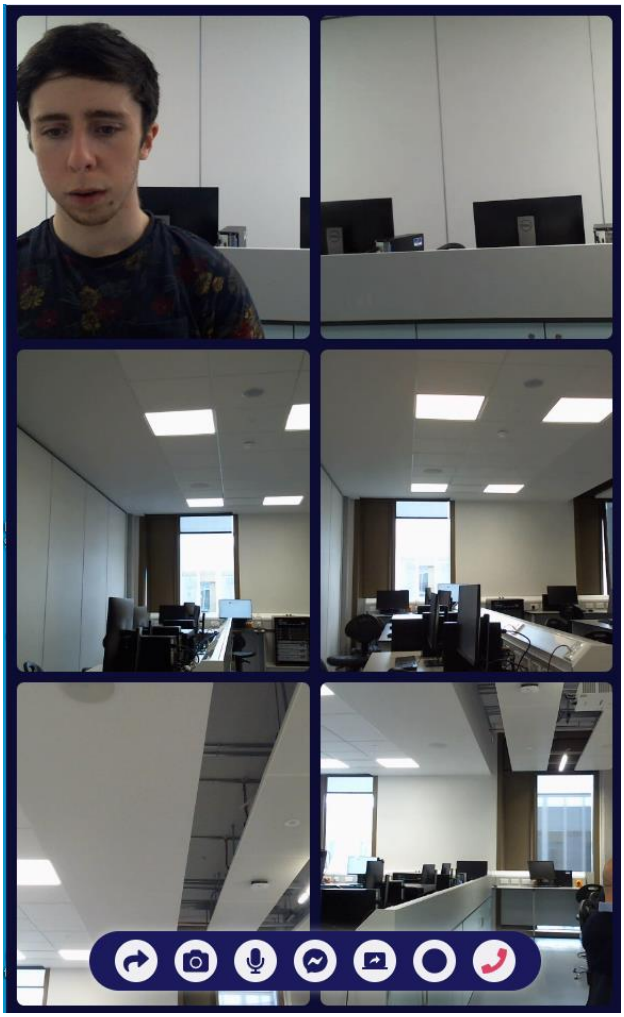


*Figure 47: Testing the responsive grid in a vertical layout*

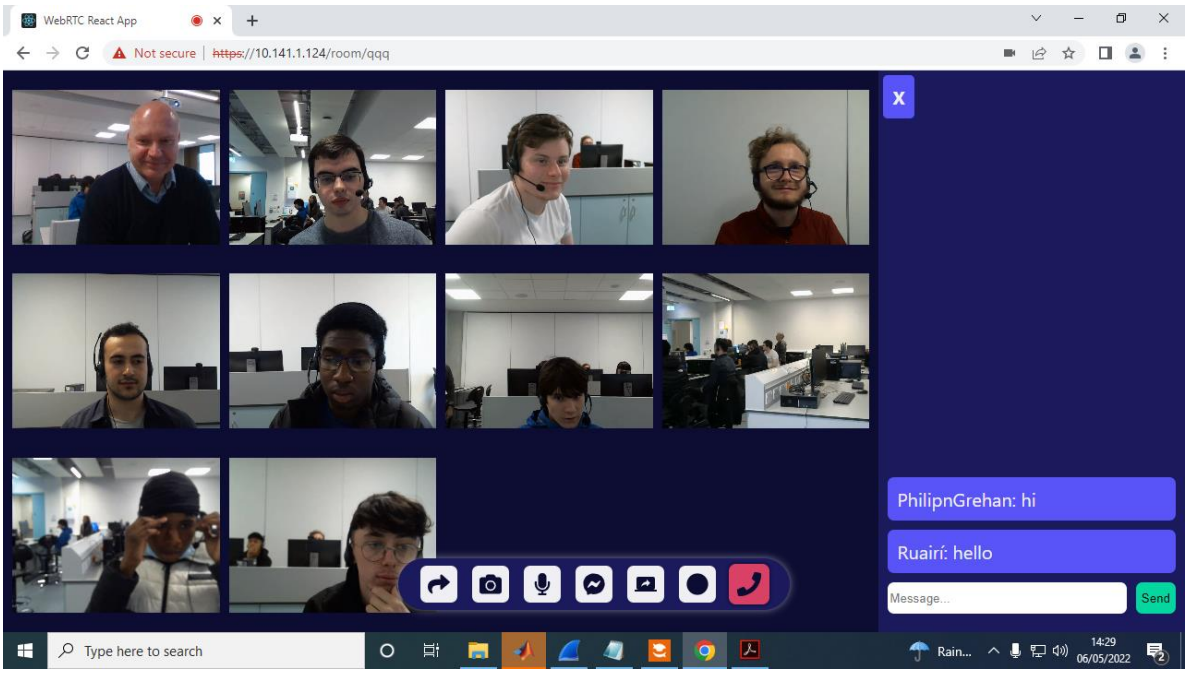



*Figure 48: An earlier version of the application being tested with a large number of clients at an open day for prospective students*

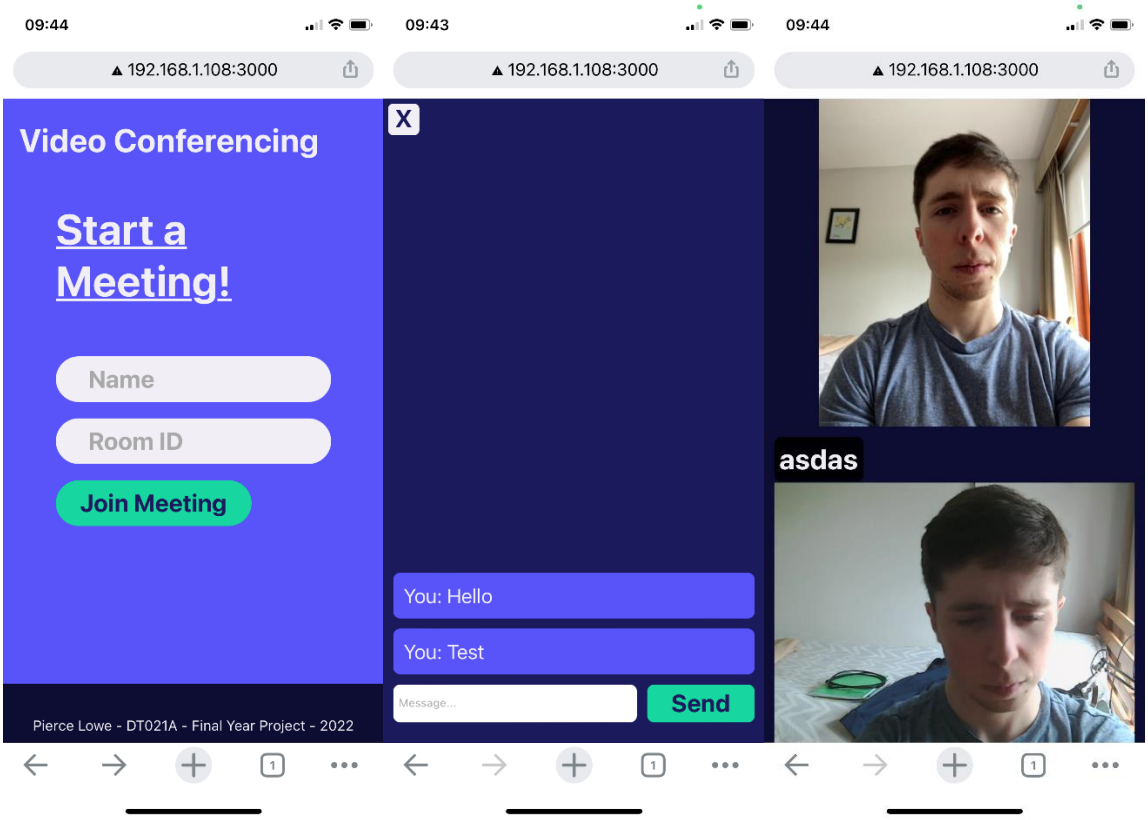



*Figure 49: Testing the application on a mobile phone*