Web Based Session Monitoring: A Unity3D Framework Proposal

A Capstone
Presented to
the Graduate School of
The University of North Carolina Wilmington

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science in Computer Science and Information Systems
Computer Science

---

by
Seth Angell
May 2022

Proposed to:
Dr. Toni Pence, Committee Chair
Dr. Ron Vetter
Dr. Douglas Kline

Abstract

Virtual Reality is often touted for its' immersive qualities, but the one-user-at-a-time aspect of headsets often leaves the rest of the room in the dark. Giving the audience the ability to view what is happening inside the headset makes it a more collaborative and enjoyable experience for all. As the technology becomes more commonplace, this lack of awareness for those outside of the headset will become a bigger issue. Especially as the tech becomes utilized in remote or classroom settings. Many platforms have tried to provide viewing solutions to this issue, but those solutions have not worked for the projects being developed in UNCW's Mixed Reality Lab. A truly effective solution to this problem would utilize standard protocols such as WebRTC and Websockets so that developers have greater control over the interaction between their headset and web clients. This capstone project describes a new application viewing solution for use within the Unity3D game engine, which allows for streaming of an in-game camera and synchronous text based communication between headset users and 3rd party web clients, all running on a modern, web service based architecture.

Executive Summary

I've worked in the Mixed Reality lab at UNCW for 3 years, and over the course of the 3 years there has been one single feature that has been requested by developers and users alike that we still have not delivered - a real time stream of what the headset user is seeing. It makes sense, everyone wants to know what's happening inside the headset, especially when a user is reacting positively or negatively. Likewise, it can be incredibly difficult to communicate changes to remote users or show off demonstrations of new features to individuals without access to the headset. For that reason, I set out with this project to create a solution.

The solution needed to be guided by a few things. It needed to be private, functional on a closed network, and packaged in docker to be usable with one of our biggest clients: Oak Ridge National Labs. It also needed to be able to be retrofitted into an existing project, so we could use it in the education based Virtual Access To Stem Careers project which has been in development for multiple years. As for my own requirements, I wanted the project to be modular. In that sense, if I created a chat service or a video streaming service, I wanted to be able to use those services in other contexts. That led to my decision to chose a microservices architecture for the project.

With all of these aspects in mind, I have delivered a framework for use within the Mixed Reality Lab. It is built primarily in Node.JS and Python for web use and C# for unity scripting. The system utilized native packages where applicable, such as using Unity's native video streaming library, and I constructed new services to help tie all the pieces together. This system allows developers to add real-time streaming to any existing Unity project as well as synchronous messaging utilizing the Alert service. This project was built with the Mixed Reality labs development style in mind, and as a result will have immediate potential for use in our active projects. All in, it finally answers the single question users and clients ask of the lab during demos: **So what are they seeing in there?**

Table of Contents

Title Page . . . . . **i**

Abstract . . . . . **ii**

List of Figures . . . . . **vi**

1 Introduction **1**
- 1.1 UNCW's Mixed Reality Lab 2
- 1.2 Oak Ridge National Labs Data Visualization Project 3
- 1.3 Virtual Access To STEM Careers 5
- 1.4 Objective 5
- 1.5 Structure Of The Paper 6

2 Background **7**
- 2.1 Virtual Reality 8
- 2.2 Game Development In 2022 10
- 2.3 Virtual Reality Development 13
- 2.4 Microservices 16
- 2.5 Existing Solutions 17
  - 2.5.1 Native Oculus Casting 17
  - 2.5.2 Tethering and Desktop Streaming 18
  - 2.5.3 3rd Party Solutions 19

3 Proposal **20**
- 3.1 Preface 20
- 3.2 The Problem 20
- 3.3 Pitch 21
- 3.4 Technology Stack Justification 22
  - 3.4.1 Ease Of Development: An argument for Python, Django, and Flask 22
  - 3.4.2 Deconstructing The Monolith: Why Microservices? 23
  - 3.4.3 Why Docker? 24
- 3.5 Unity Client Package 25
- 3.6 Web Service 26
  - 3.6.1 User Management API 26
  - 3.6.2 WebRTC Media Server 26
  - 3.6.3 Realtime Database 27
  - 3.6.4 Web Client 27
- 3.7 Deliverables 27

4 Predictions **29**
- 4.1 High Level Assumptions 29

4.2 Component Level Assumptions . . . 31
4.2.1 Alert functionality implementation with a Real-time Database . . . 31
4.2.2 User Auth and Key API . . . 31
4.2.3 Streaming Service . . . 33
4.2.4 Unity Package . . . 33
4.2.5 Demonstration Site . . . 33
4.3 Tabular Predictions . . . 34

5 Development . . . 35
5.1 Infrastructure . . . 35
5.1.1 Docker and Docker Compose . . . 35
5.1.2 Proxy . . . 37
5.1.3 Store and Store Interface . . . 37
5.1.4 Hosting Environments . . . 37
5.2 Core Functionality . . . 38
5.2.1 Authorization . . . 38
5.2.2 Alert . . . 43
5.2.3 Stream . . . 45
5.2.4 Client . . . 48

6 Conclusion . . . 50
6.1 Revisiting Our Predictions . . . 50
6.1.1 Development predictions revisited . . . 51
6.1.2 Component predictions revisited . . . 53
6.2 For future developers . . . 56
6.2.1 What I'd do differently . . . 56
6.2.2 What can still be done . . . 57
6.3 Closing thoughts . . . 58

7 Appendix . . . 60

Bibliography . . . 63

List of Figures

2.1 Release numbers for VR Motion Tracked Controller support games to the steam store between 2015 and 2021. See figure 7.2 9
2.2 Graphical Search Popularity Of 'Mixed Reality'. See figure 7.1 9
2.3 The Unity editor displaying some of the components of my finished project 11
2.4 Instructional material from Vive.com on how to set up your towers [viv, 2020] 14
2.5 Vive Pro in action from Vive's Business Brochure 14
2.6 The newest iteration of the Oculus product line, the Oculus Quest 2 15

3.1 Current ERD as of Nov 19. 2021 26
3.2 Example of potential interface for Web Client 28

5.1 A diagram displaying the network layout of the services within Docker 36
5.2 Simplified State Diagram for serverside health check logic 40
5.3 Simplified State Diagram for client side health check logic 40
5.4 UML Class Diagram displaying the data relationship behind the Authorization service 43
5.5 The initial view when a web client opens up the streaming service 46
5.6 The streaming view presented to a web client. 47
5.7 The editor view of the unity scene used for demonstration purposes 48

7.1 What users see when first opening the chat client 61
7.2 What users see once they've joined a room 62

Chapter 1

Introduction

Since the winter of 2019, I've worked within UNCW's Mixed Reality lab in a number of capacities, from one man team, to individual contributor, to project lead. Over the 3 year span, I've worked on a number of projects and began to notice some similar problems arising. Initially, the biggest issue the lab would run up against was how to create dynamic applications. That is to say, how do we construct interesting data visualizations or learning experiences, and then modify them in a way such that we can provide brand new data at runtime and be confident that the project will work. In working our way through this problem, I became aware of a number of the growing pains within the field of VR development. The most pressing being the constant instability caused by the release of new headsets and toolkits. With each release, exciting new features and abilities are added to the space, but oftentimes forced a near rewrite of any existing project to be able to run on the new platforms. For that reason, the lab has always showed interest in more stable, platform agnostic solutions to problems. Over time, common issues such as interaction toolkits (the libraries which map physical device input to in-game actions and vice-versa) were met with flexible solutions such as Unity's XR Interaction Toolkit. However, there was one common problem that has yet to be solved in an equally adaptable way: **How to allow outsiders to easily view what is occurring inside of a VR headset and communicate with the individual currently using the headset?** Therein lies the goal of my project, to develop a platform agnostic solution, which assembles a number of readily available open source projects into a simplified microservice architecture for use within the Mixed Reality lab's future projects. This goal and the requirement's outlined for it have been guided by the two biggest projects that I've worked on during my time in

the lab.

1.1 UNCW's Mixed Reality Lab

Labs all across the world are working on novel experiences taking advantage of the advancements in the field of mixed reality, including UNCW's very own Mixed Reality lab. Simulations of how to treat heat exhaustion and teach school children about sea turtles are just two of the educational experiences being developed in VR. Extending from that, the lab develops therapeutic virtual environments to help former smokers cope with common triggers, and data visualization tools to aid projects at Oak Ridge National Labs. The possibilities are limited only by development time, imagination, and the tools at hand.

While progress sometimes happen with a boom, more often than not it's a slow burn. Both of which hit very close to home in the lab. The introduction of many of the tools to be discussed later have made the lab not only more productive, but expanded our capability set, enabling different types of projects. Two of which have inspired this project in specific. I began work with the Mixed Reality lab back in January of 2019, exploring the feasibility of data visualization within Virtual Reality for Oak Ridge National Labs. This project began in an open-ended fashion, with very little content guidance and three guiding requirements: to explore new data visualization techniques, allow for dynamic data ingestion at run-time, and the ability to communicate with external services being developed by the face aging group. In its' lifetime, the project has utilized four different versions of Unity, three different headsets, and five different interaction toolkits. A series of rewrites and migrations which at times felt never ending and consumed development time which would have been much more valuable for new features. Over time, as the lab grew, systems stabilized, and the scope of work expanded, a desire to build more reusable frameworks for multi-project use arose. Most recently, an established design language for API creation and a low-level data collection system which ties into Unity's interaction system have allowed for code reuse and a decrease in lead time between idea conception to minimum viable product.

Outside of concrete frameworks, institutional knowledge has become a subject of interest within the lab. Oftentimes, projects differ in their stylistic or functional requirements but share implementation and development similarities. For example, oftentimes it is identified that a project should be capable of run-time ingestion of data and both synchronous and asynchronous communi-

cation with one or many web services. Over time, experience and its' subsequent documentation has been built up to outline best practices for accomplishing tasks of this nature. This came in handy when the lab switched from data visualization for ORNL to development on the Virtual Access To Stem Careers (VASC) project.

VASC is an NSF funded collaboration between the Mixed Reality Lab and the Watson College of Education aimed at expanding the access of STEM curriculum to elementary school classrooms without the necessitation of field trips or guest speakers. In its' pilot state, the current VASC program is centered around the experience of a coastal park ranger as they research and care for a newly laid sea turtle nest. In order to satisfy a number of requirements, it was decided that the system needed a data collection and web service component to track the progress of students. The lab was able to utilize previous experience to implement these features into an existing project. To really drill into how these two projects have inspired this solution, we'll go more in depth on the logistics and functional requirements for each one.

1.2 Oak Ridge National Labs Data Visualization Project

Circa 2019, the mixed reality lab began a collaborative effort with Oak Ridge National Labs (ORNL) to produce a number of demonstrations as to how virtual reality could help data analysts better interpret information they receive. The pitched scenario varied depending on the season, sometimes the idea could be that they only had a few minutes with a hard drive, or maybe they were revisiting data they had previously analyzed. Another constant brainstorming topic was how to tackle the situation of a team working as a collective to sift through a huge collection of info as opposed to doling it out for individual review. Throughout all of these demos, there was a constant wishlist item that has yet to be completed. That is the request that there be some sort of real time collaborative aspect to the simulations, so that individuals outside of the analysts office could see what is happening and get involved themselves. This request, while interesting, has not yet been satisfied due to a couple of blockers.

Firstly, however it is implemented it needs to be incredibly efficient. Through multiple iterations, the project has always included some aspect of photo analysis. This chews up lots of computing power in order to process the images at runtime, and has led to a number of crafty data structures to help handle the influx of work.

Secondly, the system needs to be secure. The sponsors of the project envision it being used with potentially sensitive information. For that reason, any system that is routed through 3rd party servers is instantly declined. Likewise, Oak Ridge has specifically requested on numerous occasions that new systems be delivered in a *dockerized* form, meaning that a Dockerfile and and other supporting data be included alongside the working project. This is another side effect of their secure system, even outside of the direct sponsors of this project, ORNL runs in an incredibly secure fashion with very tightly locked down systems and very little interaction with external contributors. Docker, and more specifically dockerized versions of newly proposed projects, accomplish two things for system administrators running their servers. Firstly, the containers are a single unit which can be more intensely vetted and monitored for security risks. The administrators can take the project and it's resulting container and run it in a secured, sandboxed environment for security testing. Secondly, it simplifies the deployment process in such a secured system. One could imagine in a different scenario, the development team who built the project may assist the administrators who are deploying it, however that is not possible in this scenario. Rather, the Dockerfile itself provides all the instructions necessary to the docker engine as to how to build and deploy the application within the simulated machine. This cuts down on the amount of configuration an operations team needs to perform in order to deploy a new application.

But the creation of this system would provide benefits even outside of fulfilling the requests mentions above. Oak Ridge National Labs is located in Oak Ridge, Tennessee and the location of the project's sponsors is constantly changing. For all intents and purposes, this project is being actively developed by a fully remote team. This has always presented issues when it comes to demonstrating new capabilities or developments. Oftentimes it necessitated the creation of demo videos, but these videos were created before the meeting itself would take place. At the time of display, the videos were already potentially outdated. However, the bigger issue would be the inevitable "What happens if you do this?" or "What does that look like" questions that would arise during the video call when we would demonstrate new progress. We could do our best to answer the question descriptively, but the ability to instead initiate a live demonstration in which a headset user is streaming to a website viewable by all participants of the call would have been a far better experience.

Building a system meant for remote, decentralized use, it's not a surprise that a remote streaming solution was often one of the most requested features from developers and users alike.

1.3 Virtual Access To STEM Careers

Another project of interest is the Virtual Access To STEM Careers (VASC) initiative, being completed in collaboration with UNCW's Watson College of Education. The larger simulation involves a narrative style VR experience in which users learn about how to identify different types of turtle tracks and the activities that wildlife conservation groups perform to help keep sea turtles safe. In its' completed state, the project is meant to be used in a classroom environment as part of the science curriculum.

When imagining the deployment process of this system into the real world, the topic of classroom management quickly popped up. Primarily, when you imagine working with younger children, being able to see what they are seeing can help keep student's on track as well as guide them through the activity and any problems they may encounter. While this could be accomplished by plucking a headset of a student's head to see what they see, the temperamental calibration and adjustment process that takes place when putting on a headset would make this action a bit of a time sink. For that reason, a high priority item of interest is some sort of teacher portal which would allow instructors to see what their students are seeing, in realtime.

This concern came to fruition in the Spring of 2022, during experiment trials aimed at testing the usability and effectiveness of the simulation on a population of 2nd grade students. Seeing as how second graders were below the target audience for the final system, the participants struggled with the simulation necessitating additional guidance from the research proctors. The proctors attempted to use The Oculus Quest's native streaming solution known as *Oculus Cast*. The system was not intended for our use case, resulting in an opaque experience juggling multiple facebook accounts, devices, and streaming receivers to try and figure out the right combination to make the streaming work. The end result was about a 50% success rate for streaming and a number of frustrated proctors and students. An experience which only showcased the necessity of this feature.

1.4 Objective

These two projects really guided my desire to build a system that could solve this regular problem of external awareness of inter-headset happenings. Likewise, the development constraints presented by the ORNL project helped form some of the more concrete requirements, such as a docker based deployment strategy. With that in mind, below I will layout the landscape of this

project.

The Problem: We have managed to deliver a number of features over time, but one request which keeps coming up that we've yet to deliver on is a way to stream what a user is seeing in real time. While options exists, nothing has come up that has satisfied our requirements of cross-platform availability without vendor lock-in or the necessity of third party processing servers.

The Proposal: A simple two part system which enables the easy creation of a WebRTC stream and WebSocket (Socket.IO flavored) connection from a unity client, and a backend system for ingesting said stream as well as broadcasting realtime messages from an external user to all subscribed unity clients. This project would be built in a modern, containerized, micro-service architecture to enable rapid deployment as well as the utilization of any of the core features as a standalone component. The creation of this system would allow for multiple team's to integrate realtime streaming of either the player camera or another in game device to external sources, as well as communication with users inside of the headsets. A commonly requested, yet rarely delivered feature request.

The Final Product: A micro-service based system enabling the rapid development of realtime streaming applications, paired with a synchronous, event-driven data channel. The system utilizes native libraries within the Unity3D engine to enable the streaming of a camera, and standard protocols such as WebRTC and Websockets to transfer the data between peers.

1.5 Structure Of The Paper

The general flow of this paper will be as follows. First, we covered some introductory material and a bit of the motivation behind the project. Next, we'll cover some background information on the virtual reality development space, the technology behind it, and some of the solutions which already exist to this problem. From there, we move onto my original proposal for the project submitted in December of 2021 as well as my predictions as to how development would progress. Then, a development section which covers my development experience as well as the documentation and diagrams for my finished product. Finally, a conclusion which revisits my predictions from earlier in the project as well as my notes for future developers who may use this project.

Chapter 2

Background

The crux of this project lies on the goal to expand institutional knowledge within the Mixed Reality lab on how best to implement synchronous video transmission between a VR headset and a 3rd party client. To fully solve this problem, a channel for textual data transmission alongside the video stream is necessary to allow for fully featured remote communication between the headset user and a web client. Many of the problems we have tackled seemed deceptively easy, as a cursory google search yielded tons of tutorials on the topic. What we quickly found is that many of those tutorials and techniques fell down in one way or another. Be it because they wouldn't work for materials included at runtime or they focused too heavily on optimization of elements that we were unable to tweak. We encountered a similar fate when looking into a solution for this problem in the past. To better understand these aspects of the process, I think it's important to lay out some background on what the VR development space looks like, what we're doing in the lab, and why and how I want to solve this problem.

To do so, we will first examine the state of Virtual Reality in 2022, and what advancements led us to this point. From there, a look at Game Development in 2022 - What are the tools and what are their quirks. Next we'll look at virtual reality development in specific, exploring the concept of toolkits and outlining why Unity's XR Interaction toolkit was such a huge advancement. Next, a quick brief on micro services, what they are, and why it is the architecture I've chosen. Finally, we'll look at streaming solutions which exist already, working through why they haven't been the solution we're looking for.

2.1 Virtual Reality

Mixed Reality creeps more into conversation with each passing day, with systems such as Meta's (previously Facebook) Metaverse becoming a commonplace topic. Probably one of the most notable instances of it within popular culture would be the explosion of the Augmented Reality application *Pokemon Go* in 2016. However, the uses of mixed reality extend far outside the social sphere. With over half a billion downloads[Sensortower.com, 2020] the application has no doubt been a success, so much so that it has piqued the interest of educators. A French study found that by integrating experiences similar to Pokemon Go, students were able to connect more deeply with the subject they were currently studying[Remmer et al., 2017].

The biggest catalysts for these advancements have been the release of powerful game engine's such as *Unity3D* and *The Unreal Engine* for free use, and the development of headsets such as the *Oculus Quest* and *HTC Vive* for sale at consumer price points. Whereas previously these experiences and the development tools needed to create them were reserved only for research labs and AAA game studios, now the average person with a laptop and a small initial investment (~$300) can have access to the same tools as big studios. The creation and proliferation of these tools has been instrumental in the advancement of the Mixed Reality space from a niche academic pursuit into mainstream culture. Looking at search frequency for the term *Mixed Reality* on google since 2010 shows a clear uptick, see figure 2.2. More notably from a development mindset, the sheer number of VR Motion Controller supported releases on Steam (a popular video game marketplace) saw a 4200% increase between 2015 and 2016, see figure 2.1. This explosion of development perfectly coincides with the release of the Oculus Rift, the first consumer targeted virtual reality headset to gain mass market appeal. Since then, development of newly released applications has steadily continued. Much like the introduction of the personal computer, citizens now have access to not just the technology required to experience these simulations, but also the tools required to develop them.

Slowly overtime, developers tend to figure out better ways to use the tools at hand. When that doesn't work, new tools are developed. The lab has been pretty lucky in it's discovery and successful utilization of many different free-to-use or open source solutions to problems we've encountered. In a way, the Unity engine itself is an example of that. However, within the virtual reality development space, many of these tools tend to be focused on the type of work required by game development. For example, one area the team has explored is how to build a *"multiplayer"*

Figure 2.1: Release numbers for VR Motion Tracked Controller support games to the steam store between 2015 and 2021. See figure 7.2

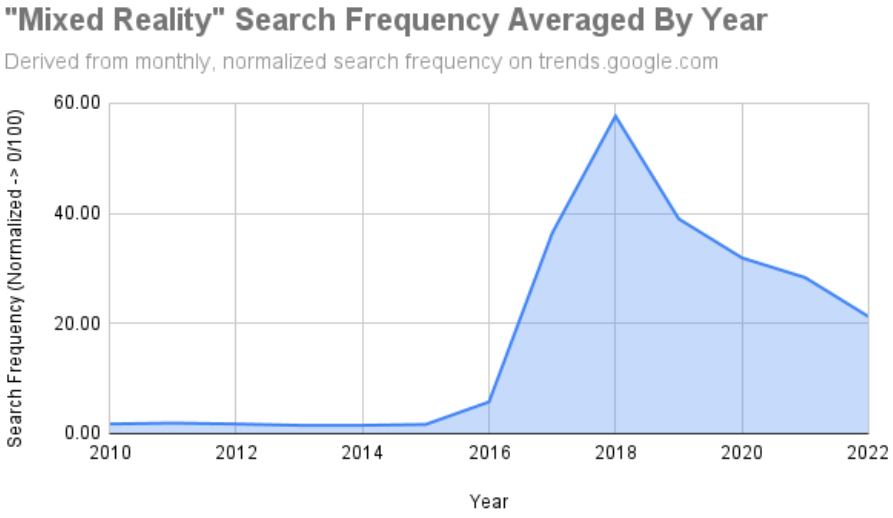

Figure 2.2: Graphical Search Popularity Of 'Mixed Reality'. See figure 7.1

experience, in which multiple users are interacting in the same space. Many off-the-shelf solutions exist to solve this problem, such as the popular Photon Engine[1]. However, these solutions expect to be deployed in a traditional game environment which makes use of common game development techniques. In the case of photon, one of the development issues we encountered was the assignment of tracking status to newly generated objects. Popularly, video games will optimize their performance by pre-generating and hiding a collection of in-game assets to be used at a later date. A technique known as *object pooling*. These could be 3d models of monsters in a dungeon exploration game or low-poly pedestrians in a city simulator. When it becomes necessary to display these models to the player character rather than create a new model at that point in time, the game will instead transport one of these hidden models from elsewhere in the scene to the player's current position. Since this practice is commonplace, the support and documentation for the synchronization of newly generated objects is spotty and hard to work with. Because of disconnects such as the one listed above, when possible we will opt to build either an in-house solution or wrapper around an existing solution to better suit our needs. One great example of this issue has been synchronous transfer of visual and textual data between the headset and an external service. Through the use of standard HTTP requests and some data structures work, we've developed a system for efficiently loading and displaying photo assets at runtime without crashing the program.

2.2 Game Development In 2022

While major studios are still building their own engines, a large market share of the gaming industry is built atop of one of two engines: Unity3D or Unreal Engine. Now these are effectively feature complete systems, coming out of the box with physics engines, lighting libraries, sound design tools, and a ton more. With all of this functionality built in, the editor's can be a bit complex, as seen in figure 2.3. In the case of Unity3D, which we'll focus on heavily for the rest of this work, it also includes a fully functional C# scripting backend for the implementation of more complex game logic.

Like any other programming pursuit, game development comes with it's own collection of best practices and weird quirks. Despite the labs work not necessarily being the development of video games, many of the rules of game development still apply. This is mostly because the underlying

[1] https://www.photonengine.com

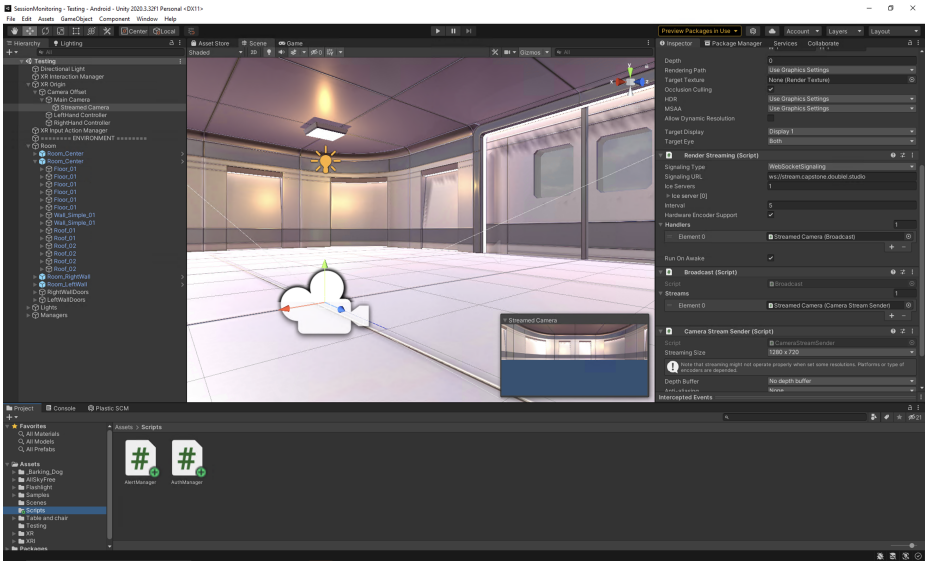

Figure 2.3: The Unity editor displaying some of the components of my finished project

computational tasks are very similar and the tools utilized are primarily developed for purpose of creating video games.

The tasks of a game engine, especially in the case of Unity and Unreal where they have been abstracted out for use by the general public, are resource intensive. The expectation is that one should be able to build a graphically complex game with thousands of calculations happening a second, without any lag. But the onus can't always be on the engine, so many guides and courses will outline a variety of techniques to help cut down on computation costs. One technique is the use of Low Polygon models to help reduce the amount of work the rendering engine must do. While we're able to utilize this trick for environment creation, through the use of stylized environments as opposed to hyper realistic ones, oftentimes we're loading full res images in which eat into these performance gains. Another popular technique mentioned above is Object Pooling - the generation of 3d models at start up for reuse as opposed to on-demand generation. Both of these techniques focus on pre-optimization to gain run-time benefits. Like many other methods, these are centered around knowledge of what your runtime environment will look like. An expectation which is not necessarily a deal breaker for video games but harder to guarantee when you're building something that should have access to dynamic data. [Koulaxidis and Xinogalos, 2022]

Game Development is home to a suite of these types of trade-offs. Sometimes these trade-offs come in the form of content decisions like mentioned above, other times they manifest themselves

as programming techniques or language features - a huge one being asynchronous programming.

In Unity, one such practice is the model of Coroutines. Coroutines are a unity specific C# language feature built atop the native C# IEnumerator class[2]. Coroutines allow for a form of asynchrounous programming by requiring "at least one 'yield' statement in the body of the coroutine, These yield statements...allow the code to be run asynchronously." [Wolf, 2021] The yield statements acts as a sort of contract between the developer and the engine stating that as long as subsets of a logical flow are followed in order, the engine is free to pause execution at the specified yield points to complete more mission critical tasks such as rendering the next frame.

This coroutine pattern is emblematic of the engine's larger role in the system. An almost kernel like one in which developers register listeners and make requests to change data and the engine determines whether that is allowed and when it should happen. Meanwhile, the engine itself is running the mission critical tasks such as ensuring that a frame is drawn on schedule. When working with the Unity3D engine, a developer is bound to encounter the concept of the Unity Event System. Unity's user interface interaction system is built with an event driven approach, in which user interface interactions are registered as events which are published to a global message queue. Developers can then register listeners which subscribe to the queue and fire off actions when subjects of interest (such as the pressing of an onscreen button or scrolling of a page) arise.

Unity has begun to lean fully into this relationship with their upcoming architecture shift to an *entity-component system*. The Entity Component System is part of a larger push at Unity to become more Data Oriented. Within this system, the programming paradigm becomes substantially more declarative. Rather than writing procedural code to iterate through a list of Game Objects to turn off gravity, you'd instead declare something like "Set the gravity attribute of all entities with a Rigid Body component tagged 'enemy' to False". This thought process lends to this larger system of asking the game engine to make changes on your behalf, and then awaiting their response [Meijer, 2019].

The space of Game Development is full of these opinionated design decisions out of necessity. Simply determining how to navigate them to create your ideal product is a key skill in this space, but learning why they are in place and the advantages they bring can pay efficiency dividends in your compiled project.

[2]IEnumerator is a generic interface for the creation of iterable objects. It allows a user to create a system collection which can be utilized by a for-each loop for example

2.3 Virtual Reality Development

The virtual reality development space has been progressing steadily overtime, but truly exploded in growth around 2015. To fully grasp how the rapid progress has effected development, we'll quickly review the progress in both the hardware and software sides of the development space.

We'll start with the hardware, that being the Head Mounted Displays (HMDs) themselves. Research into VR and HMD's reaches as far back as the 80's. It wasn't until relatively recently that the hardware became cheap enough for widespread adoption. A 1994 paper surveying the current landscape of HMD's had the cheapest device priced at $6,000, with the most expensive being priced at $95,000 [Veron et al., 1994]. For the sake of brevity we're jumping right to 2016 when consumer headsets became openly available. We begin with the Outside-In, 'lighthouse' generation of headsets, most popularly you have the Oculus Rift and HTC Vive. These HMDs did exactly what they said on the tin. They were effectively two computer monitors appropriately spaced to simulate binocular vision. They were 'tethered'[3] to a powerful desktop machine which did all of the computation and rendering of what was displayed on the headset. The reasoning for their designation as 'outside in' tracked is that they were aided by sensor's commonly called 'lighthouses' which used Lidar to determine where a user was in the room. Lidar, "a remote sensing method that uses light in the form of a pulsed laser to measure ranges", quickly became the defacto method for spatial mapping within the VR space [US Department of Commerce and Administration, 2012]. These Lighthouses would be placed in opposite corners of the room, as explained in Figure 2.4, and help accurately map the player's real life movements to their virtual environment. Anecdotally, these setups were large, immobile, and necessitated constant calibration.

One small advancement was the introduction of wireless adaptors for headsets. Through the use of PCIe cards attached to an IR camera, headsets could be wirelessly driven from a desktop. This allowed users to do away with the thick bundle of cables (see figure 2.5) running from the back of the headset, and feel more free to move around. While this was useful, the next major advancement came with the shift from outside-in to inside-out tracking.

Inside-out tracking moved the room mapping work of the towers into the headset itself. Through the utilization of cameras now embedded in the front of the headset which filled the spatial mapping role of the former external lidar towers, users were able to cut down on the number of

[3]Connected by a varying collections of cables back to a traditional computer in order to receive power and display data

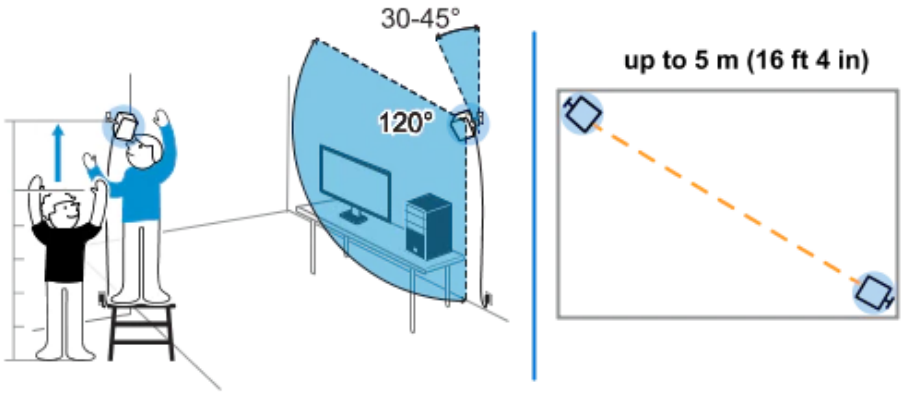

Figure 2.4: Instructional material from Vive.com on how to set up your towers [viv, 2020]

Figure 2.5: Vive Pro in action from Vive's Business Brochure

Figure 2.6: The newest iteration of the Oculus product line, the Oculus Quest 2

discrete devices involved in setting up a functioning VR experience. All of these improvements led to the modern day hardware landscape, standalone VR HMD's.

With over 10 million version 2 units sold as of November 2021, the Oculus Quest has quickly become a staple of the Virtual Reality marketplace [Sag, 2021]. The Quest ushers in a new generation of headsets in which all computation takes place on device. No device tether, no lighthouses, a true all in one device small enough to be tossed in a bag. The Quest utilizes the same Qualcomm Snapdragon processor we commonly see in Android smartphones, running a version of Android OS. While this aids in its' mobility and ease of development, you do lose some of the compute power that was afforded by a hardwired desktop with a powerful graphical processing unit (GPU). That being said, the hardware is only half the story. The developer tools for VR programming have seen similar waves of change.

One of the more frustrating aspects of VR Development before Unity's XR Toolkit was vendor lock-in. Since the consumer headset space was young, headset vendors such as Oculus or HTC were trying all different types of hardware configurations and development tooling. Likewise, the input options for each headset varied. For this reason, headset vendors would release toolkits to developers. Effectively software development kits (SDKs)[4], these toolkits would provide hooks

[4]Software development kits (SDKs) are a concept within the software engineering space. These are libraries of code written in different programming languages which allow developers to interface with software or hardware offerings

into the headset they were associated with, that could be programmed against within Unity. They were a method for communicating to the game engine that a user had tilted their head forty-five degrees, or held down the trigger on their controller. This allowed the game engine to map their movements to the in-game camera, and allowed developers to perform actions based off user input. The problem with this setup is that it was difficult to switch between headsets as it would necessitate reprogramming your application against a brand new toolkit. This happened on more than one occasion within the Mixed Reality lab, for example when we migrated from HTC Vive headsets using the SteamVR toolkit to Oculus Quests using the Oculus VR toolkit.

Microsoft then entered the scene with their novel concept, a middleware type system which existed between Unity and the headset toolkits. With the first commit landing around 2018, Microsoft began work on a system that would allow developers to create cross platform applications with minimal overhead [Microsoft, 2022]. Selfishly, this aided Microsoft as they were producing a number of headsets at the time. Developing against the mixed reality toolkit meant more content for Microsoft's line of HMDs. To accomplish this task, developers write to Microsoft's SDK, and then the mixed reality toolkit maps to a number of different headsets. While this was nice, it still involved a 3rd party solution becoming deeply intertwined with your core codebase. That is until the release of Unity's XR Toolkit.

The Unity XR Interaction toolkit effectively implemented Windows Mixed Reality Toolkit's solution as a first party offering. They created a collection of standardized input mappings to be used by developers, allowing for a "write once, deploy anywhere" experience similar to that touted of Java. Unity then works in collaboration with platform vendors such as Oculus to write mappings from their XR toolkit to the native platform's SDK. This allows developers to develop a single application with confidence that it will be usable on a variety of headset types and vendors. This change took the development cycle from a series of hard decisions and bespoke codebases to a more modern, cross platform system.

2.4 Microservices

Within *Leveraging microservices architecture by using docker technology*, the authors lay out 3 core concepts which define the microservice architecture: small and focused, loosely coupled,

from other companies in a simplified fashion. This is in opposition to a developer needing to grasp the entirety of the codebase behind a new tool in order to use it's features

and language neutral. Small and focused in that no single service intends to take on the lion's share of features. Rather, services within a microservice architecture focus on doing one thing, and doing that thing in an incredibly resilient and generic fashion. This allows for service reuse across multiple problem domains. Secondly, they should be loosely coupled in that the internal implementation of any single service should not be overly reliant on the internal implementation of another service. That is to say that a chat client could rely on a chat service to relay a message, but neither the chat service nor the client should be utilizing any knowledge of the other party outside of the services publicly exposed endpoints. For example, developers of a chat client should only need to be aware of the format it must authenticate and transmit messages to the chat service with, but should not be asking information of the chat service developers about how they serialize messages or other internal details. This loose coupling allows for the internal implementations of services to change overtime without the need to communicate these changes to those who rely on the services. Finally, microservices should be language neutral. Communication should be handled over a standard protocol such as a universal message bus or http web requests. This allows services to be implemented in the best language to fulfill their requirements without needing to worry about how other services will interact with them. If all members of the architecture utilize the same standard protocol, there need be no concern about language mismatch. As the authors outlined, these attributes pair very nicely with the tooling and conceptual model laid out by containerization technology such as docker, which focuses on the separation and coordination of multiple discrete services running within the same system [Jaramillo et al., 2016].

2.5 Existing Solutions

Before deciding to try and piece together my own system, I wanted to see if there was anything on the market already which could potentially satisfy the necessary requirements to get this project running. While many solutions answer a subset of the requirements, nothing quite covered all the necessary components. Below is a sample of some of the solutions considered.

2.5.1 Native Oculus Casting

The most immediately available solution, and the one we've used in a pinch before, is native screen-casting within Oculus OS. This is a first party streaming solution which allows a user to

stream their current screen to the Oculus app or a supported web browser. It accomplishes this by implementing the streaming role of the google cast protocol, commonly referred to as chromecast. In some ways, the convenience of this solution is a big incentive in its' use, but there are some pain points as well which must be factored in.

Since the system utilizes an existing, popular protocol, the steps to set up a device to receive the stream are very simple. In fact, with the use of a $35 chromecast HDMI receiver, a user could stream the headset directly to any HDMI enabled display. From an ease-of-use standpoint, this solution excels. However, the Oculus cast streaming option works on a device level as opposed to an in-game feature. That means that any system built on this technology will need to utilize an Oculus quest or other google cast compliant device. This solution also presents a roadblock for future improvements such as a multi-stream environment in which multiple users are streaming to a single display. It also necessitates that users be on the same local area network (LAN), so the remote utilization mentioned previously is also unachievable.

2.5.2 Tethering and Desktop Streaming

The next option utilizes a backwards compatibility feature still available in many headsets: desktop tethering. While most headsets are capable of standalone operation, many still offer the ability to tether back to a computer and act more as a monitor. In this method of use, what the headset is seeing will be displayed on screen. A solution could be used to capture the screen from the host computer, and stream it to a web client using tools such as OBS[5] or FFMPEG[6]. While these would be easier solutions than a roll-your-own approach, they come at the cost of returning to a state where all headsets much be tethered to a computer.

The extra computation power provided by a standalone desktop machine would help alleviate some of the performance concerns that arise when streaming from a headset on top of rendering a game. However, it does necessitate the return to a state in which headsets must be connected to a desktop machine. This adds complexity both in a physical sense (more machines to move from location to location) and digital sense (how do we automate the capture and streaming of the tethered headset view on play). This solution provides answers to our performance problems, but necessitates the addition of a second piece of hardware to our final setup.

[5]https://obsproject.com Open Source Software for video streaming with a GUI
[6]https://ffmpeg.org Another open source solution offering more flexibility at the cost of being CLI only

2.5.3 3rd Party Solutions

The last option I examined was a survey of Unity Assets on the Unity asset that claim to solve this problem. The *Unity Asset Store* is a marketplace built around selling packages for Unity development. These often provide easy implementations of complex features, visual assets to improve the look of your game, or libraries of codes to implement features such as hit detection or inventory management. Among these packages are a few different streaming solutions I looked into, the first being Agora Video SDK[7]. Agora is a software-as-a-service company which offers plug and play communication services in the form of SDKs. They abstract away the issue of server management by routing all communication through their own systems. While this simplifies development on our end by removing the need for serverside development, the use of 3rd party servers would be met by an immediate veto from the team at ORNL, meaning any solution utilizing this system only solves the problem for half of our clients.

There are also packages such as FMETP Stream[8], a system that boasts tons of features and allows for more interoperability with a variety of clients. But this system was entirely built around the concept of keeping everything inside Unity, including the server. While this was appealing from its' purported "5 minute setup", it felt like the value it provided wasn't enough to justify the time investment to grok what exactly this 3rd party service was doing and how we'd utilize it. I also found that documentation for running a Unity game in a headless fashion[9] was sparse, meaning this solution would introduce additional complexity which needs to be handled. While it would reduce the amount of work required by developers to understand how to implement these features, it introduces a dependence and requirement for knowledge on a closed system as opposed to something more open.

[7]https://assetstore.unity.com/packages/tools/video/agora-video-sdk-for-unity-134502
[8]https://assetstore.unity.com/packages/templates/packs/fmetp-stream-v2-202537
[9]Running software on a machine with no display attached

Chapter 3

Proposal

3.1 Preface

The following two sections (Proposal and Predictions) were written as part of my first deliverable in Winter 2021. The final project was delivered in Spring of 2022. I've included these sections in the final report as a way to show how the project transformed over time, with that being said there is mention of software and design decisions in the following sections which were not implemented in the final design. Those changes and the justification for them are contained within the conclusion.

3.2 The Problem

In the context of the work being done in the Mixed Reality Lab, the projects always have 2 distinct categories of tasks. There is the primary task, anything from a demonstration of how to visualize data in VR to an instructional simulation on how park rangers measure the size of sea turtles. We'll call that the *novel task*. Moving within concentric circles, we then encounter what I'll call *operationalizing tasks*. These are requirements which exist in parallel to the novel tasks at hand, which focus on how to realistically implement whatever is created.

Historically, these have been centered around how to take a system that works with known images, and allow it to dynamically load new images at runtime. These tasks usually involve some outside coordination with a web service and can be implemented with varying levels of difficulty

depending on the performance envelope of the targeted end device.

Over time, the lab has built up institutional knowledge as to how to implement a variety of these tasks. Most have been smaller in scope, in which a single developer can explore and learn how to best implement these solutions alongside other tasks. One, however, has always been difficult. That is, the streaming of video out of the headset and into another generic platform. While platform dependent solutions exists, oftentimes these are not an option due to network or privacy reasons. Likewise, while a working understanding of RESTful service design and HTTP server creation can be picked up alongside other work, video streaming has always been put on hold due to it's inherent complexity.

This project aims to outline a variety of desired features and then function as an exploratory venture to implement as many as possible. The hope being that by the end, some of the institutional knowledge as well as a functioning prototype can exist for further development. With that being said, let's get into the specifics of what we'd like to create.

3.3 Pitch

To address the lack of platform agnostic solutions to the remote viewing problem mentioned previously, I'm proposing a two part framework aiming to provide plug-and-play access to a WebRTC stream from a Unity client, as well as a communication channel for the synchronous transfer of information back and forth. Piece 1 will consist of a Unity package with simple configuration parameters to allow for a connection to be made back to the web service. This Unity package will contain scripts to attach to an in-scene camera as well as a blank GameObject [Technologies, 2022] to act as hooks for the WebRTC and Messaging system. Piece 2 will be a web service with a microservice based architecture, capable of providing 4 specific services: User management, realtime messaging, WebRTC relay, and a simple web client to interface with the entire system. For more specific details, a feature list is provided below. This chapter will begin by laying out a specification for each individual component. As the project timeline allows, the goal is for each piece of the framework to be written as a respectful citizen for each given platform (Unity, Django, etc). However, this stretch goal is liable to be dropped if it threatens the potential for deliverables to be completed by their prescribed deadline. As deliverables are provided and components are finalized, the prototype chapter will contain in-depth documentation.

Features:

- A turnkey unity package that attaches to a camera in scene and instantiates a game object which will provide a listener hook to be alerted if any new messages come through.
- Simple Unity Inspector panel based configuration forms for easy setup
- A Restful, Django based web api which ties into the Django User System
- An API Key based permissioning system that allows unity game clients to connect to the web service without any additional login information
- A WebRTC media server to act as a peer-to-peer relay for all transmitted video.
- A Realtime Database (Open Source Supabase)[1] for message broadcasting
- A simple web application for testing that allows for the display of a unity client and a text box for sending messages to said client

3.4 Technology Stack Justification

3.4.1 Ease Of Development: An argument for Python, Django, and Flask

The entirety of the web based component will be built using Python and two of its' most popular web frameworks: Django and Flask. This may seem odd, seeing as how Python is widely considered to be a slow language (largely due to its' implementation details [VanderPlas, 2014]). Considering so many of the mentioned tasks are speed and resource sensitive, it bears some explanation as to why this project would chose Python for the implementation. There are a number of key reasons which primarily boil down to the developer experience.

Firstly, this is a system built for work done in the Mixed Reality lab. In the past, the primary point of hire is shortly after completing CSC231. As of Spring 2021, CSC131 and CSC231, the first two courses in the development track within UNCW's computer science degree, are taught in Python. Oftentimes, Students are usually completing CSC231 within the first 2 years of their program, meaning there is a substantial amount of time left for them to be productive members of the lab before they complete their degree.

[1] https://supabase.com

Since it is useful to get early insight on whether a new candidate has strong long term potential, having a system built in a language they're comfortable with will reduce the onboarding time it takes to assign them their first task. This could be the difference between finding out a student is a strong hire within the course of a semester long directed independent study, or after a year of learning and tutorials in a brand new language and system. By creating a system in a commonly understood language, new hires can hit the ground running and provide meaningful contributions to the project while they take the time to learn the other languages required for our projects.

Secondly, Python is "batteries included" [Dubois, 2007] and has a huge developer community [Stackoverflow, 2022]. From an efficiency standpoint, writing this in golang, .Net, C, or really any other compiled language would most likely show strong improvements. However for a minimum viable product, python would be the fastest language to iterate in and try new things out.

Finally, Django is an established framework powering sites as large as instagram.com [Engineering, 2016], it is also what we've been using in the lab for the past 2 years. It does come with a learning curve, but similar to python's *batteries included* approach, it includes a lot of niceties that makes writing web applications easier. Many of which we will utilize such as their User model complete with authentication and session management. Add in the fantastic Django-Rest-Framework library and the decision is easy.

3.4.2 Deconstructing The Monolith: Why Microservices?

Microservice architectures introduce a non-trivial amount of complexity. Rather than making use of functionality with internal function calls, things which used to be separated by a directory are now separated by a router. This can have drawbacks, such as a degradation of service during network outages, duplication of similar efforts across services, and the introduction of architectural complexity.

That being said, especially in a project like this, there are some arguments as to why one would utilize this model. Right off the bat, it enforces well documented inputs and outputs as well as behaviors. Since microservices are supposed to communicate using external messaging formats as opposed to directly calling code, you're forced to be more fine-grained with your endpoints. Unlike within a monolith where one can jump over to the file and read the entire function definition, microservices are intended to be accessed using their publicly available API. This can serve as

an additional step within the separation of concerns, even more enforceable than something like Interfaces. While this project will begin as a single developer effort, if used in the future it could be worked on by a number of developers. Clearly outlining the inputs and outputs now can help establish a system with low coupling and high cohesion [Montiel, 2018].

Another huge benefit of microservices with clearly defined inputs and outputs is the ability to mix and match languages. Unlike a monolith in which languages would need to call executables directly or implement a wrapper of some sorts, microservices can instead simply define a common messaging format. As long as all members adhere to said format (HTTP and RestAPI's in our case) they need not be aware of each other's implementation details. This allows one to select the best tool for the job.

This trait allows for easy upgrading. If we match the previous API spec exactly, we could drop in the new implementation of one of our services in as little time as it takes to spin down and spin up a new docker container. So, hearkening back to the statement earlier that compiled languages may be a better option, if we find in the future that the Django Auth API works fine, but the WebRTC server needs more horsepower we can do just that. Re-implement the spec in Go, change the docker-compose to spin up the new container, rebuild, and we're back in business.

3.4.3 Why Docker?

Docker, and containers as a concept more broadly, have taken the industry by storm. Containers have numerous use cases, ranging from environment management to ease of on boarding [2], both of which are beneficial to us here. The project is intended for use in a variety of environments, so being able to run the project as long as you have docker installed on the host device is advantageous. Following that, with this being a microservice based architecture we will need to regularly initialize a variety of different services - each with their own requirements. To aid in this, docker-compose can act as a lightweight orchestration software to help quickly start up all of the services with all of their specific configuration values. Finally, docker leaves a lot of space for future expansion. If we're interested in load balancing or other scaling solution such as Kubernetes, building our architecture with containers in mind can make these solutions much easier to implement.

[2] https://www.docker.com/use-cases/

3.5 Unity Client Package

All things considered, the Unity components should be relatively lightweight and easy to produce. For performance reasons, only essential processing will take place on device with the rest being offloaded to the web services. For that reason, the architecture of the unity component should relatively simple. Most likely it will container two heavy lifter scripts, a single script to handle system configuration in the Unity Inspector, and potentially one final script to handle coordination between tasks.

- CameraManagement.cs: This will be the script attaching to the desired in-game camera we'd like to stream. It will heavily utilize the experimental Unity-WebRTC package [3]. This package is in an early beta state, with Android support (what we build to for VR executables) being recently released. I anticipate some potential debugging work during this portion of the project, since support was so recently rolled out.

- AlertManagement.cs: This will be the script attached to a prefab which provides a listener hook for in-game alerts from the message broadcasting portion of our web service. At runtime, utilizing the (tentative) api-key based connection scheme, the script will send a request to our realtime-db that it would like to subscribe. Once the subscription is granted, it will listen for any new data and store it in a buffer for the developer to pull from.

- InspectorConfigurationPanel.cs: This will be a script which will render a new Inspector pane on import [sample photo here]. This panel will allow users to easily input necessary information such as api keys, web service urls, and other important configuration data.

- Coordinator.cs: If necessary, this will be a script which manages the runtime syncing of all three components mentioned above. If the scripts themselves start to feel bloated, this offers an easy offloading point to jump to. It would simply maintain references to each script individually, provide those references when necessary, and log all error messages.

[3] see https://docs.unity3d.com/Packages/com.unity.webrtc@2.4/manual/index.html

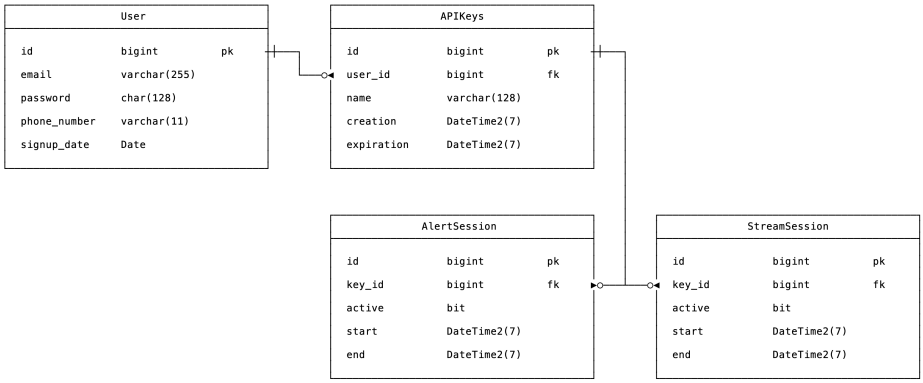

Figure 3.1: Current ERD as of Nov 19. 2021

3.6 Web Service

3.6.1 User Management API

The user management portion of the project will handle the creation of a new user, authentication of API keys, and validation of active sessions. It will contain endpoints for authenticating a user, creating an api key, and listing all active sessions. The API will only serve data over JSON, meaning the web client will be responsible for displaying this information.

Additionally, when the Media or Alert server receive a new API key, they will first check to see if said API key is legitimate. If so, they will then check their internal system to see if a service is already active associated with this key, if not they will start one and then register this new session with the User management API. The media and alert servers should also notify the UserManagementAPI when the last user disconnects from the session, so that the session can be marked as inactive in the database.

3.6.2 WebRTC Media Server

The WebRTC media server is being used as opposed to a traditional p2p architecture for future features. While they are not within the scope of the project, tiling multiple streams into a single view as well as recording active streams, are both future goals that would require the use of a media relay server. In this implementation, the media server will contain an endpoint that listens for a WebRTC initialization request, user type (streamer or viewer), and an APIKey. It will then authenticate the key with the user management API, and then either capture or relay the

stream depending upon the user type. To begin, this is planned to be implemented in Python using the aiortc library[4], if problems are encountered with latency or efficiency, the fall back plan is an implementation in either .NET or Go.

3.6.3 Realtime Database

The Alert system has been the most dynamic part of this design. Moving from websockets to firebase, back to websockets, and now landing on an open source implementation of firebase. Supabase is an open source realtime database engine built atop PostgreSQL. For this project, we'll utilize it to provide a 2-way synchronous channel for messaging between the web and unity client. For demonstration purposes, this will be artificially locked to 1 way (web client -> unity clients) but could be turned into a 2way implementation with very trivial code changes. The primary reason it will not be, is that defining a system for textual user input in VR could be a capstone entirely on it's own.

3.6.4 Web Client

For demo purposes, I aim to have a single page similar to below that will allow a user to see what the VR headset is displaying and send messages which are delivered to the user in VR. This is simply to act as visual proof that the system is working, as without it it would be hard to demonstrate.

3.7 Deliverables

The product I am proposing to deliver at the end of my capstone can be defined by the following deliverables:

- **Unity:** A collection of scripts which allow for the configuration of video streaming to an external service and communication with some sort of chat entity.
- **Chat:** A service, packaged in docker for ease of deployment, which allows for synchronous communication between services over a clearly defined protocol with outlined messaging formats.

[4]https://github.com/aiortc/aiortc

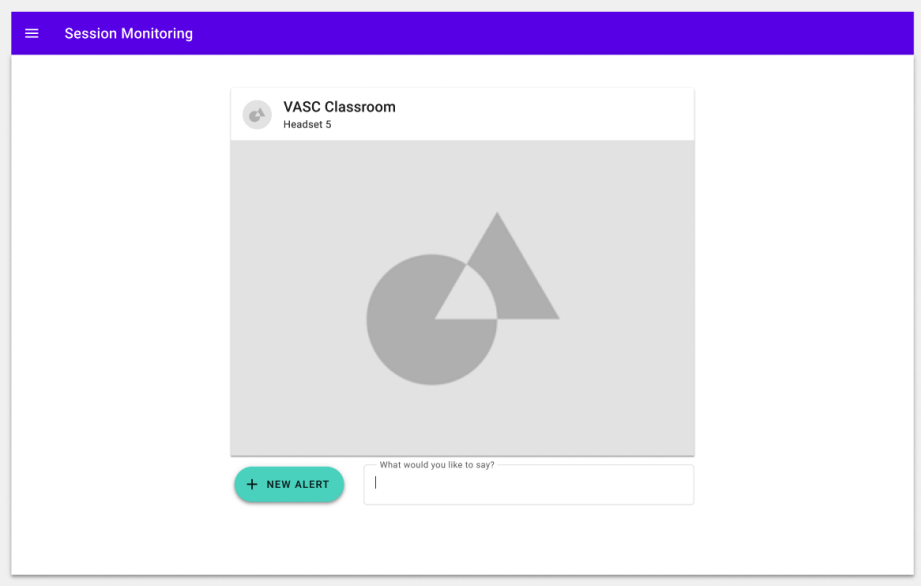

Figure 3.2: Example of potential interface for Web Client

- **Streaming:** A service which enables the receiving and relaying of streamed video to clients.
- **Authorization:** A service which authenticates API keys and acts as an orchestration device - generating unique session ID's for new instances of streamed content.
- **Demo Site:** A bare bones HTML/CSS/JS webpage which allows a user to authenticate, view a stream from unity, and send a message to the unity client.

The following section outlines my predictions for how the project would progress as of December 2021. In section 6.1 I revisit these predictions after the project was completed and examine where they were accurate and in what ways they fell apart.

Chapter 4

Predictions

As with any new project, there are a few assumptions that must be made up front. Given the academic nature of a capstone, it only seemed prudent to adopt the hypothesis/results framework in the form of predictions and review. As such, this section will contain a collection of pre-development assumptions and decisions that have been made. With the completion of each portion of the project, the respective predictions section will be revisited and evaluated for accuracy. These musings on the expectations versus reality will be found in the *conclusion* section.

4.1 High Level Assumptions

One of the big decisions with this project early on was the micro-service architecture which is heavily coupled with the container first build strategy. On this side of the development path, there seem to be 3 major server-side domains: User Authentication, WebRTC relay, and real-time database subscriptions. While these could have been built in a monolithic fashion, one of the large goals of the project is modularity. Ideally, this should be a system which can be integrated selectively into other efforts within the lab. Regardless of the opinionated design found within each component, they should present a clear list of input and output parameters allowing for isolated use in other projects. For example, the User/Key Authentication API component should contain all the necessary inputs and outputs to create users, create keys, authenticate users and keys, and provide permission evaluation for actions defined within the admin panel. While all of these features are being built specifically for this project, in isolation they could easily be dropped into a different

project in the lab with just a little bit of legwork. Likewise, by splitting these large concepts into what are effectively a collection of smaller projects, it makes the required work far less daunting.

Going container first was a decision made to alleviate concerns around deployment. A potentially large amount of software packages and libraries will be needed in order to effectively develop this project. As the list grew, the potential for pain during deployment only seemed to increase. For that reason, containers were chosen as a development tool to help enable the simplification of the development environment. Dockerfile's [1] will be used to automate the environment setup of each component and pre-seed the state necessary for each portion to function. Docker Compose [2], Docker's lightweight container orchestration system, will streamline the creation and destruction of all necessary services. Compose was was chosen over Kubernetes in the interest of staying true to the K.I.S.S. [3] ideology in some aspect of this already complicated structure. Allowing for the easy attachment of system volumes and mapping of ports for web exposure, Compose provides just enough functionality to get the job done without adding the unnecessary overhead of dynamic load balancing and resiliency.

That being said, this architectural decision doesn't come without drawbacks, to list a few:

- Everything is a web app, meaning error handling needs to be replicated across each component
- Transit lag is being introduced. An operation may now take 2 hops (1 to authenticate, the next to real-time db for example) before it can return. As opposed to 1 request to a single endpoint which then handles all the work with no external calls needed
- Specs need to be rock solid before development begins. Since each component will need to communicate with each other other web requests, the input/output documentation will need to be up to date and comprehensive in order to achieve proper communication between components.

The duplication of work will definitely be a potential issue, the question still stands as to how much of an issue it will become. That can only be answered after development has begun.

[1] Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image https://docs.docker.com/engine/reference/builder/

[2] Compose is a tool for defining and running multi-container Docker applications https://docs.docker.com/compose/

[3] *Keep It Simple Stupid* https://en.wikipedia.org/wiki/KISS_principle#In_software_development

4.2 Component Level Assumptions

For each component, there are a few different assumptions being made which have guided their design. These can be in the form of language selection, off-the-shelf versus roll-your-own decisions, and feature set definitions. To better see what the project looks like at both ends, these assumptions are being listed here.

4.2.1 Alert functionality implementation with a Real-time Database

For the alert functionality, I've opted to utilize a real-time database to implement this feature to cut down on development time. Originally, this was going to be accomplished with a web socket based design, which would open and subscribe all active users to a message queue on the server. This would allow the management interface (this is the demo site in this project) to publish messages to this queue which would be propagated to all clients. This feature started originally with the plan to use Google's Firebase, however it was quickly decided that the system should be able to function properly in a closed LAN environment with no access to the larger web. Upon the discovery of Supabase, an open source Firebase alternative with self-hosting potential, the feature's implementation was migrated back to a Realtime database from its' brief foray with web sockets.

At time of writing, seeing as how Supabase implements an HTTP api with realtime subscription, the implementation seems rather straight forward. A very simple Flask API will sit alongside the Supabase container. All the API will do is act as a `start_session()` endpoint, which receives and session key and ensures that a table exists in the Supabase service for said session. If it does not, it creates one. Upon creation or validation of existence, it will return a 200 code and then client will then connect to the `ALERTS` table with the provided API and session key, and listen for new messages to be published.

4.2.2 User Auth and Key API

The authentication API is primarily focused on solving the issue of differentiating between streaming sessions in the event that multiple users are active at the same time. To do so, it will implement a simple API Key system which allows developers to register an account and then request API keys. These keys will be stored alongside the unity client during the compilation process, and used to access the Alert and Stream services. Alongside the API Key issuing functionality, it will

also contain endpoints for authenticating keys and creating new `Session` objects. These `Session` objects will act as references to every instance of a client engaging in a streaming session. This allows for a timestamped log showing every time an individual Unity client began streaming back to the server.

This allows for 2 features, one for potential future development and another that we get for free right away. The free one, is that it allows for the querying of all past messages that were sent to a unity client.

Imagining a scenario in which this system is integrated into a training application, in which a user works their way through a task in a a VR headset and an instructor follows along from a webpage. The instructor may offer advice to the user in the form of session alerts, by sending messages from their webpage. As a way to track progress, one could go back and see the types of messages that were sent from the instructor to the student, by requesting a collection of corresponding session keys and querying the alert database for all messages with a reference to said key.

For future development, a recording feature could be implemented into the *Stream* service which would allow for the capture of a WebRTC stream from a client for future viewing. This captured output could be referenced to the `Session` object which would allow for future querying.

For the implementation of this API, Django and Django Rest Framework were selected as the platform for development. Django is a web framework implemented in python, which follows the python ideology of being "batteries included", meaning it ships with substantial functionality that other frameworks may ignore. The particular functionality we're interested in is the fully featured User model. By using Django as a base, we won't need to implement any of the common requirements of a user model such as session tracking, password management, permissioning, and many other useful features. Not only will we be able to use this for the actual User model, but it is anticipated that these features can also be inherited into an API Key class to save development time.

That being said, Python is commonly discounted as a "slow" language due to its interpreted nature and lack of true multithreading. However, since Django is the framework powering applications as large as Instagram [Engineering, 2016] and many multithreading WSGI servers capable of serving Django applications such as Gunicorn exist, I believe that losses in speed are more than made up for by the returns driven by faster development cycles.

4.2.3 Streaming Service

As for the WebRTC Streaming service, the biggest up front assumption being made is the network architecture for the stream relay. While I could have opted to use WebRTC as it was originally intended, in a peer to peer fashion, the potential for VR headset resource limitations became a concern. For example, both projects in which this will be integrated run on the Oculus Quest, a standalone VR headset powered by an android phone processor. While streaming to 1 to 5 peers may not be too exhaustive, as the number of active viewers rises the potential for visual degradation due to resource limitation rises. For that reason, the server will implement an SFU [4] architecture, which will accept a stream from a headset, and then duplicate it for any viewer. This offloads the stream duplication workload from the headset allowing for lower resource utilization, and potentially higher quality streams as a result of only needing to transmit it once.

4.2.4 Unity Package

The Unity Package has the potential to require a very small amount of development time. This is due to Unity's inclusion of WebRTC capabilities within the engines standard library. For that reason, this portion of the project will effectively be a wrapper task, in which I will create a collection of scripts which use Unity's WebRTC libraries to stream to our desired endpoints. For that reason, a lot of time is planned on creating a simple developer experience for integrating this package into their project. I plan to create a series of UI panels which provide clear information on what information goes where, where to store the API Key, and how to ensure that the library is functioning. That being said, none of those features are guaranteed, and will be very quickly dropped as requirements if the usage of Unity's WebRTC library does not go as planned.

4.2.5 Demonstration Site

This will be a very simple, bare bones, HTML web-page which allows a user to enter an API Key and see any streams which are active underneath said key. There will also be an input box which allows messages to be sent to connected clients. Seeing as how this web-page should be able to be written in HTML, CSS, and vanilla JS, without the use of any external framework like Angular or React, the anticipated outcome is that development is relatively short. The site will be hosted

[4] Selective Forwarding Unit https://webrtcglossary.com/sfu/

Project Component	Estimated Difficulty	Confidence in Estimate	Confidence In Ability To Build To Spec
Realtime Database	Easy - 1 week	90%	100%
User/Key API	Medium - 3 Weeks	75%	90%
WebRTC Server	Hard - 5 weeks	90%	80% (SFU v. P2P)
Unity Client	Hard - 3 weeks	95%	90%
Web Client	Easy - 1.5 Weeks	100%	100%

Table 4.1: Estimation Of Component Development Timelines and Difficulty

Date	Milestone
01/14/22	Git repository created and directory structure established
01/21/22	Supabase + Postgres containers created and added to git
01/28/22	Base Django project created in repo
02/04/22	User Auth + html pages for browser access to API created
02/11/22	API Key + all endpoints added to API
02/18/22	Base WebRTC service in repo + Simple Unity Test Client
02/25/22	Basic WebRTC endpoints functioning with Unity Test Client
03/04/22	SFU Architecture implemented in WebRTC Client
03/11/22	API Key validation logic implemented in WebRTC Client
03/18/22	WebRTC Component complete (Extra week left for flex time if needed)
03/25/22	Base Unity client created and in repo
04/01/22	Demo environment created and WebRTC attachment script created
04/08/22	Alert script created and UI panels for framework. UnityPackage in repo
04/15/22	Basic web client frame created and hosting set up
04/22/22	Basic Web client complete. Begin E2E Testing
04/29/22	E2E Testing Complete
05/03/22	Capstone Defense

Table 4.2: Ideal Timeline With Milestones

behind the larger Nginx proxy running the application, so deployment will be a couple of lines in the Nginx configuration and a docker mount command to place the files at `/usr/nginx/share/html`.

4.3 Tabular Predictions

Presented here are two tables, one being an estimation of the difficulty of each element and the other being a rough week by week timeline of planned milestones.

Chapter 5

Development

Development of this project took a number of twists and turns, many of which will be discussed in section 6.1 where I revisit my predictions from earlier in the project. For this section, I want to simply outline what code has been written for this project, the thought process that went into it, and the infrastructure necessary to run it.

5.1 Infrastructure

To start, we'll outline the infrastructure which allows the isolated services I've built to interoperate successfully. Most are simple, off-the-shelf services which are popular in the field. This section aims to discuss how they are utilized, and what is running behind them. With a solid understanding of what they project is built atop of, we'll then dive into the specifics of each piece.

5.1.1 Docker and Docker Compose

Docker is the preferred tool for software deployment at ORNL, so this project would eventually need to be dockerized[1] in order for it's delivery. With that in mind, I approached the project from the beginning with the goal if it being Docker native. It could run on bare metal hardware as a native installation, but from the start it was developed, tested, and eventually deployed within docker containers.

As stated previously, every single element of this project runs within a docker container.

[1] Colloquial term for modifying a project so that it runs within a docker container

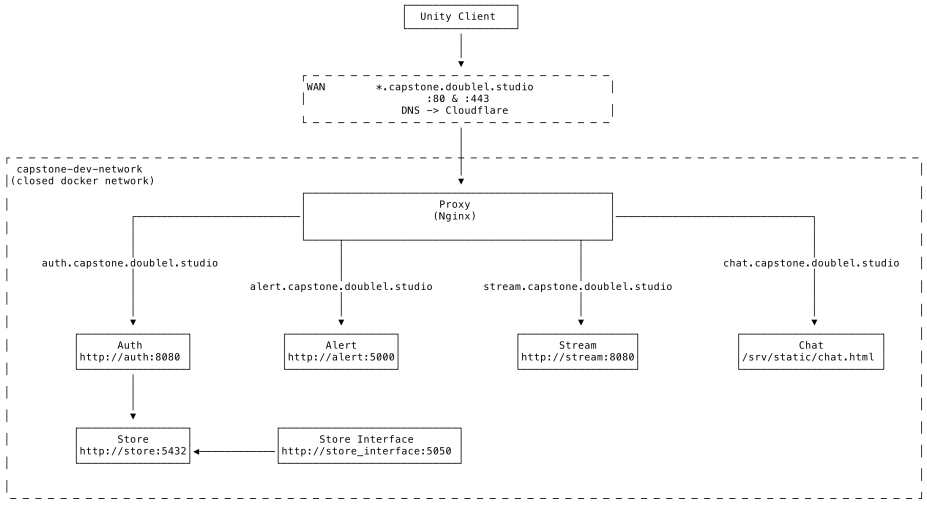

Figure 5.1: A diagram displaying the network layout of the services within Docker

Each service has it's own container with two corresponding Dockerfiles: `Dockerfile` and `Dockerfile.prod`. These files tell the Docker engine what steps must be taken to properly construct the container from scratch, including the copying of the code from the host machine into the container, installation of dependencies, build commands, and environment variable configuration. The Dockerfile.prod file specifies additional environment attributes to allow the service to run on a publicly hosted machine.

Each service also has it's own docker-compose.yaml file. This is a basic orchestration service offered by docker, which allows the user to not only specify which service to build, but also what services it relies on, and what environment variables must be set.

The project contains two top-level docker-compose.yaml files, one for development and one for production. These files are capable of starting up every single service in the system in the correct order, ensuring they share the same virtual network, and mapping any required volumes to their correct mount points. Through the use of these files, a development or production environment can be instantiated from scratch with very little input from a developer outside of the original commands incarnation.

Finally, one big advantage that docker brings is that it helps restrict who is able to access each service. Meaning a generic WebRTC client wouldn't be able to connect to the streaming service unless is was authorised client side. This is achieved by following two steps. First, a *docker network* is created within the docker-compose command which all containers connect to. This network is

closed off to exterior access, except where specifically permitted by the opening of ports. We then utilize the proxy container to allow access into the network from exterior requests. Internally, the containers are able to communicate with each other freely. Externally, only port 80 and 443 are open to the Nginx container containing our reverse proxy. Each service is then configured to only allow requests which originate from approved sources such as the headsets or other container within the network. This adds an additional layer of control to the entire system.

5.1.2 Proxy

One of the supporting services utilized is a containerised version of the Nginx web server. This service (called `proxy` within the architecture) acts as the entry point for every service in the system. Depending on the deployment location, it maps each service to either service-name.capstone.local or service-name.capstone.doublel.studio. So for example, in local development mode the authorization service can be found at auth.capstone.local, but in production requests are routed to auth.capstone.doublel.studio.

Additionally, the proxy service hosts static files for some of the web applications found in the system. It also hosts the previously mentioned chat service which can be used to test the generalized alert system. See figures 7.1 and 7.2 in the appendix for images of this service.

5.1.3 Store and Store Interface

The last big software pieces include a dockerized instance of PostgreSQL for the Authorization service to tie into as well as an instance of PGAdmin4 for easy configuration. PostgreSQL was selected for its' popularity in the open source space, and since it is the preferred database within the Mixed Reality lab.

5.1.4 Hosting Environments

Finally, thanks to the portability Docker provides, this system has been deployed and tested on MacOS, Windows 10, and Ubuntu 18.04/20.04. In it's deployed state, it runs with more than enough headroom on the smallest virtual machine Linode[2] offers, with 1 core, 1 gigabyte of ram, running Ubuntu 20.04. The project can be found at stream.capstone.doublel.studio.

[2]linode.com

5.2 Core Functionality

Now we'll go through the core functionality of the system. These subsections will each contain a details about a single service that fulfills one of the requirements of the system. We'll first look at the Authentication API and how it works, talk about it's tech stack, and outline it's core inputs and outputs. From there, we'll examine the Alert system and talk about the events it expects clients to handle. From there, the streaming interface and how one interacts with it, and finally, the Unity Client for integration into games.

For reference, the services architecture can be seen in figure 5.1 to help visualize which services are publicly exposed.

5.2.1 Authorization

The *authorization* service has two core objectives: To restrict viewership access to streamed content and data channels to an allowed set of applications as well as session management to provide unique ids to each instance of a user streaming to a viewer. This identifier will aid future work in multi-stream environments and data collection. The authentication scheme is client based as opposed to user based, meaning each valid client would have access to an API key which it uses to authenticate with the server, as opposed to a user providing a username and password. This is done for future expansion reasons, a future developer could piggyback off this system to then add a second level of user-identification which allows a user to provide credentials to gain access to specific streams or content. The service also runs a background job every 2 minutes to check on the timing of the last *health check* from a client, to determine which services are still operational. More on this functionality later.

5.2.1.1 Terminology and internal systems

There are three concepts that exist as the backbone of this system, those are the ideas of Active Projects, Health Checks, and Session Rooms. Each represented a core component to the system I wanted to build, in which multiple client devices were tied to a single *project*, and as such would want to be connected to the same room. However, since so much of this project was taking place over a network connection how could we be sure that messages such as a `disconnect` command were properly communicated? As such, I created a small system to manage this state.

At a high level, there are two key components which link clients together: Their API Keys and the Projects they belong to. A Project would represent a single instance in which a developer would be building a streaming client (A Unity3D game) and a monitoring client (some sort of web application) which should be paired together in some way. When clients authenticate with the server, their API keys would be linked to a project and as such the system would know which Session Room to connect them to. In the example of VASC, a developer would create an API key for the compiled binary destined for the headsets and assign it to a Project for that classroom. They would then instantiate an instance of the Teacher Portal which allows teachers to monitor what their student's are doing within the VR space, and assign it an API key linked to the same project. But how would we differentiate between an instance of the students streaming on Monday and Tuesday, or even just between one class period and another? That is where the *Session Rooms* come in.

From the beginning, this project was an exploratory endeavor to help determine the best way to implement these features in future projects. With that in mind, when possible I wanted to leave in the framing for future expansion. Session Rooms are an example of that. There needed to be some way to differentiate between discrete instances of users connecting to the streaming service. That is where *Session Rooms* come into play. A Session Room represents a period of time beginning with the first registered client's connection to a project and ends when the last registered client's health check is over 2 minutes old. These Session Rooms currently provide separate channels within the chat service so that user messages from different project don't collide. A similar feature could be implemented within the WebRTC streaming service to allow for differentiated streams as well. The Session Room is a string identifier optionally prefixed with a *friendly room prefix* (see figure 5.4 followed by a 32 character UUID. If another client within the same project is already connected, we return to the user the Session Room generated for that client. However, if a client is the first member of their project to connect, we must generate a new Session Room. To do this, I needed a way to determine which projects had users actively streaming.

To accomplish this task, I dynamically define a subset of all Projects known as *Active Projects*. Active Projects live in their own separate table, inserted during the `/get-session-room/` call and removed by the `check_for_timed_out_room()` job, this logic can be seen in figure 5.2. For this implementation, I considered a service such as Redis[3] which felt like a logical fit. Store a single key known as Active Rooms which contained an array of ids corresponding to rooms that were

[3]https://redis.io

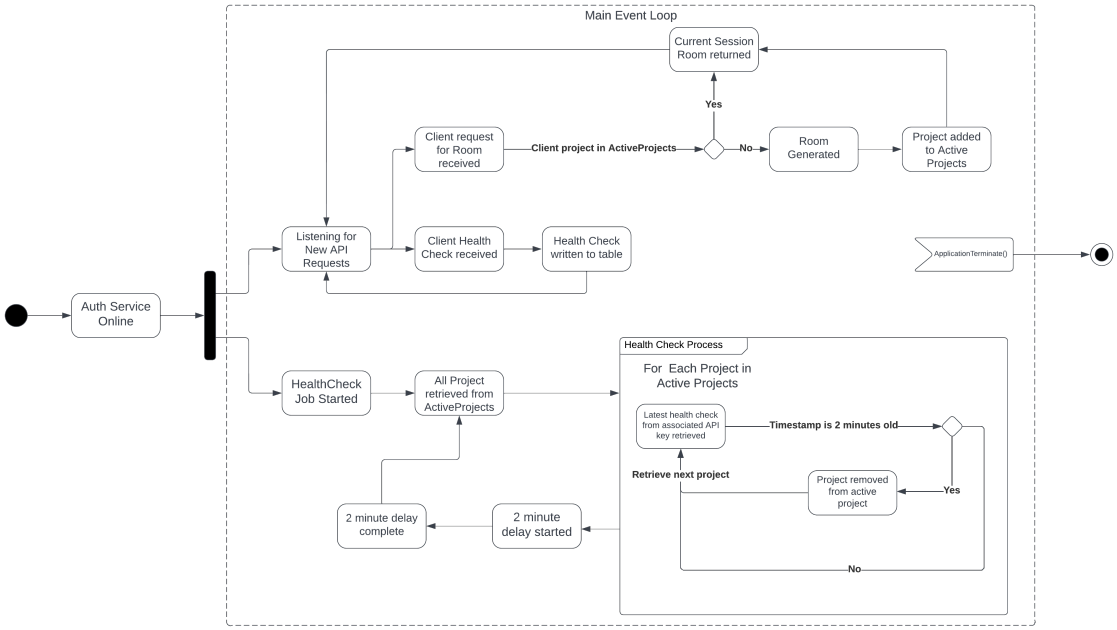

Figure 5.2: Simplified State Diagram for serverside health check logic

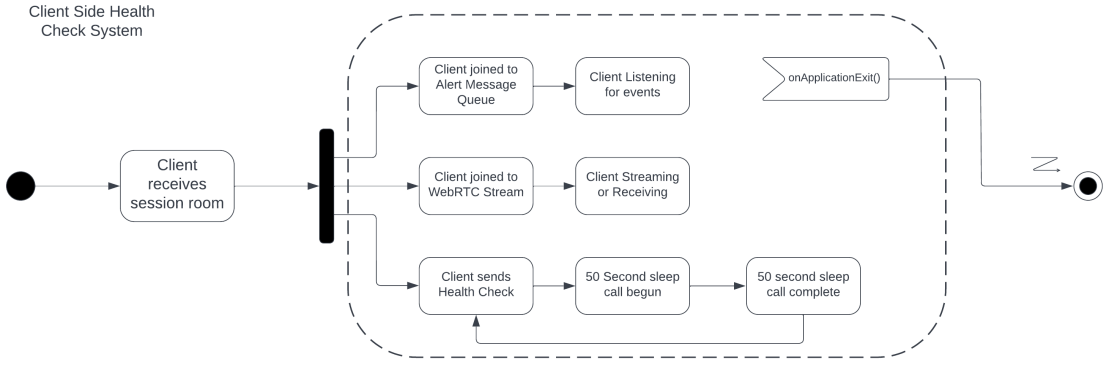

Figure 5.3: Simplified State Diagram for client side health check logic

currently being streamed to. However, it seemed foolish to pull in an entirely new service to store a single Key/Value pair. Instead, I used a table in the database which already needed to exist. The active projects table contains all currently active project IDs and their corresponding Session Rooms. This means that anytime a new request hits the `/get-session-room/` endpoint, we can quickly scan to Active Projects table for the project matching the request's API key. If we have a hit we return the room, otherwise we create a new Session Room and store that alongside the project in the Active Projects table. This handles adding new projects and meant we were close to completion, but how do we determine when a currently Active Project no longer has any active streamers[4] or monitors[5]?

Therein lies the concept of Health Checks. Since the entire system is microservice based, the authorization system was completely disconnected from both the stream and alert services. One option to determine status would be to have the authorization system 'listen in' on the happenings within the alert and stream services to determine which streams were active. However, that felt like an injection of unnecessary complexity into the system. The next option would be to have the clients transmit a `disconnect` request before the application exited. My concern here was that with the opaque nature of both web browsers and headsets, how would we handle a crash or simply a lost packet. While those fears may have been unsubstantiated, I'm satisfied with the route I chose in the usage of *health checks*. After first connection with the streaming service, all clients open a thread which has one single job: every 50 seconds, send a POST request to the `/api/v1/auth/ping/` endpoint containing their API key and Session Room. These requests are stored alongside a timestamp of their arrival within the Health Check table. This logic is show in figure 5.3

Within the auth service, a job runs every 2 minutes which follows three steps:

1. Retrieve the ids of every project in the Active Project table.
2. For each id, retrieve the most recently timestamped Health Check with an API key which belongs to the selected project.
3. If the Health Check is more than 2 minutes old, remove the project from the Active Projects list.

[4]the headset client which transmits a video stream back to the server
[5]the web client which allows a user to view the stream and send messages to the headset client

This allows the system to prune the Active Projects list regularly, and attempt to ensure that Session Rooms are doled out when a project completely vacates and is then revived. This also allows the system to store a conclusion timestamp for session rooms, for later analysis of how long users are spending in Sessions. This system does have the failure state in which a project could be completely vacated and then rejoined within a 2 minute time frame, and a new Session Room would not be generated. In that event, the new grouping of clients would received the same Session Room already assigned to their project. Further testing would allow the time frame of this job to be shrunken, as well as a hybrid approach which listens for `disconnect` messages as well as checking for stale timestamps. All in, these are the concepts which make up the backbone of the Authorization service.

5.2.1.2 Implementation

This system was implemented with the **Django** web services framework, a python based framework for rapid, server-side web development. Alongside Django, a number of key packages were used to make this system possible:

- **Django Rest Framework**[6] - A toolkit for creating restful web services and endpoints, piggybacking off of the existing Django ORM and logical infrastructure.
- **Django Rest Framework API Key**[7] - A library for Django Rest Framework which reduces the amount of boiler plate necessary for custom API key authorization.
- **Schedule**[8] - a lightweight python library for the creation and scheduling and background tasks. Helped cut down on infrastructure complexity by providing an internal method for evaluation client health checks as opposed to external ties to cron jobs or a celery queue.
- **Green Unicorn**[9] - A python WSGI HTTP server used for production hosting to help make the system more robust by managing multiple instances of the web service.

The system only uses a user account to authenticate into the server side administrative panel, which allows a developer to create API keys for new client devices. The data model can be seen in figure 5.4. The service defines two endpoints: `api/v1/auth/get-session-room/` and

[6] https://www.django-rest-framework.org
[7] https://florimondmanca.github.io/djangorestframework-api-key/
[8] https://schedule.readthedocs.io/en/stable/
[9] https://gunicorn.org

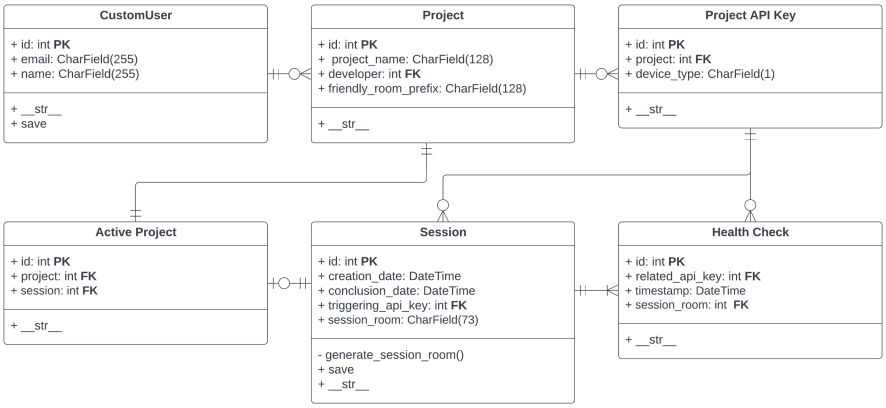

Figure 5.4: UML Class Diagram displaying the data relationship behind the Authorization service

`api/v1/auth/ping/`, used for session room retrieval and the submission of health checks respectively. `/get-session-room/` accepts GET requests with an API key, and returns a Session Room name for use in the alert system. `/ping/` accepts POST request with a body containing the client's current Session Room. This helps the authorization service keep track of active clients. All communication is encoded as JSON with a special authorization header with the following key/value mapping: `"Authorization": "Api-Key <CLIENT_API_KEY>"`.

5.2.2 Alert

The alert service is effectively a two-way chatting system which is artificially locked into a unidirectional path: Monitors can send messages to Streamers. However, the only thing stopping the two-way communication system is for the proper events to be implemented client side (as seen by the existence of the `chat` service implemented elsewhere in the stack). The service is written entirely in Socket.IO, a WebSocket framework with convenient HTTP-Polling fallback in the event that WebSockets are unavailable.

5.2.2.1 Implementation

The service runs on top of the following crucial libraries:

- **Flask**[10] - A lightweight web framework built in python. Trades Django's "batteries included" approach for greater modularity.

[10] https://flask.palletsprojects.com/en/2.1.x/

- **Flask-SocketIO**[11] - A flask module which streamlines the use of the Socket.IO protocol within a Flask project.
- **Green Unicorn**[12] - Green Unicorn is again utilized, this time primarily for it's Pre-Fork model in which we can accepts requests and then spin up new processes for each response.

Flask was selected for this service simply because we need very few of the niceties that Django comes pre-loaded with. Instead, this service needed a lightweight web framework to house out Socket.IO service for all other services to communicate with. The Alert service does not implement any sort of authentication or authorization internally. Instead it requires a `room` attribute to be passed upon connection, to help direct messages to the correct recipients. As long as a client adheres to the communication specification set forth by the service, it can use the service to relay messages to other users which connect to the same room. The service however does not advertise what rooms are available, so clients must establish their room id before connecting. This means that the only way to "listen in" on an active streaming session would be to brute force the prefix and UUID assigned by the auth server. In it's implementation, the Alert service is more or less a very simple chat service capable of transmitting user messages and state changes (user joined, user left, etc). The following section outlines the messaging specification.

5.2.2.2 Messaging Specification

The Socket.IO framework is event driven, meaning clients all subscribe to a centralized messaging queue and should implement listeners to react accordingly to different types of events. The Alert service expects three types of events to be emit (sent) by a client:

- **joined** - Emitted when a client joins a room. The event should contain a room name to join tied to the key `room` and a user's display name which will correspond with all of their messages, tied to the key `name`.
- **text** - Emitted alongside a message the client would like to transmit. Expects the same `name` key as above, as well as a `msg` key containing the text they'd like to broadcast to the room.
- **left** - Emitted right before a client closes their connection, contains simply the display name of the client.

[11] https://flask-socketio.readthedocs.io/en/latest/
[12] https://gunicorn.org

By implementing these 3 event emitters, A client is now able to broadcast messages to the server. Serverside, the Flask service is listening for these 3 events, performing some actions, and then broadcasting it's own messages to the queue. On `joined` and `left`, the service adds or removes a user from the specified room and then transmits a `status` event to the queue. When a `text` event is emitted, the service receives the event and then broadcasts a `message` event to any rooms that the emitter is a member of. For this reason, for a client to fully implement bi-directional communication, it must implement listeners for the following two events:

- **status** - Clients should be prepared to accept a status event containing a `name` key and a `state` key. The state key represents a change in state (joined, left) and the name key represents the user who's state has been changed. Clients are free to display this information as they see fit.
- **message** - Clients should be prepared to accept a message event containing a `msg` and `name` key. `Msg` will contain a string value representing a new message sent by the user found in the name key. Clients are free to display this information as they see fit.

This system was intentionally written in an open fashion as to allow for its' use in other circumstances. For example, a functioning chat client was written using basic web technology (HTML, CSS, JS) as a test case for this service. It can be found at `chat.capstone.doublel.studio`. The event driven system also allows for additional events to be added serverside without forcing clients to change their system. If future developers wanted to add an event for object position changes in Unity, they could define a `location` event to transmit this data on, and the chat application would still work as it did before.

5.2.3 Stream

The streaming client utilizes an open source project put out by Unity which ties in with their experimental Render Streaming package which we'll talk about in the following section. Unlike the rest of the Unity engine, since this is an experimental package it's offered without many of the niceties that come with the rest of the engine. As a result, I did some tweaking to make it fit this project.

The web application is written in Node.JS[13] and utilized the Express.JS[14] library to build

[13]https://nodejs.org/en/
[14]https://expressjs.com

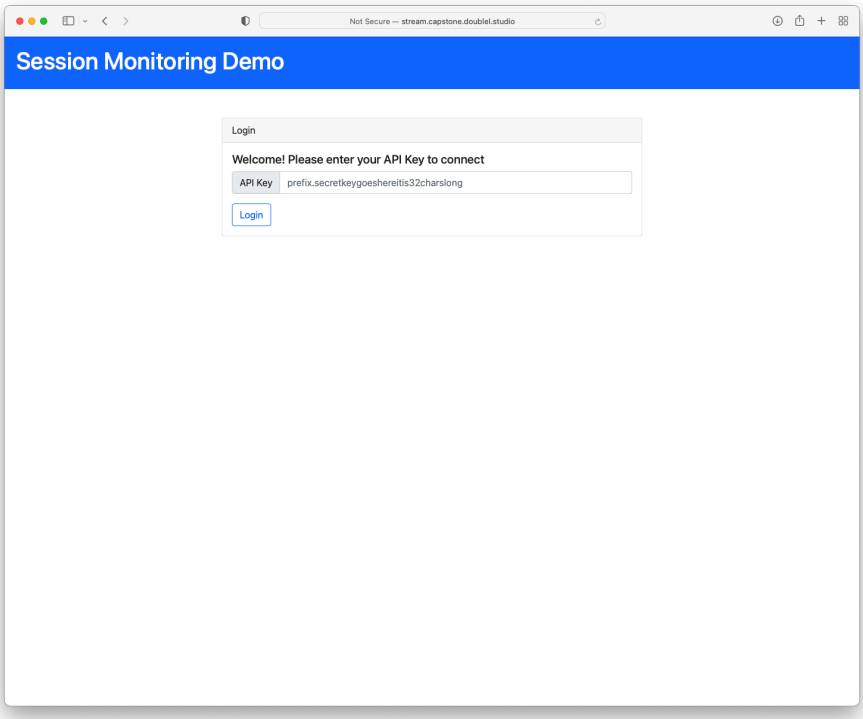

Figure 5.5: The initial view when a web client opens up the streaming service

the backend application. In addition, I pulled in the Socket.IO[15] javascript library to tie into the alert system, and Bootstrap[16] for styling.

The client implements a basic peer-to-peer WebRTC connection between the web client and the first headset which connects. Many web clients can view a single headset stream, but as of currently a web client cannot view multiple headset streams simultaneously.

For demonstration purposes, the web client communicates with the authorization server by requesting an API key from the user and then grabbing the Session Room corresponding to that API key, see figure 5.5. Upon entering an acceptable API key, the web client is then presented with a video player for viewing and an input box to transmit messages to the streaming headset, see figure 5.6.

[15] https://socket.io
[16] https://getbootstrap.com

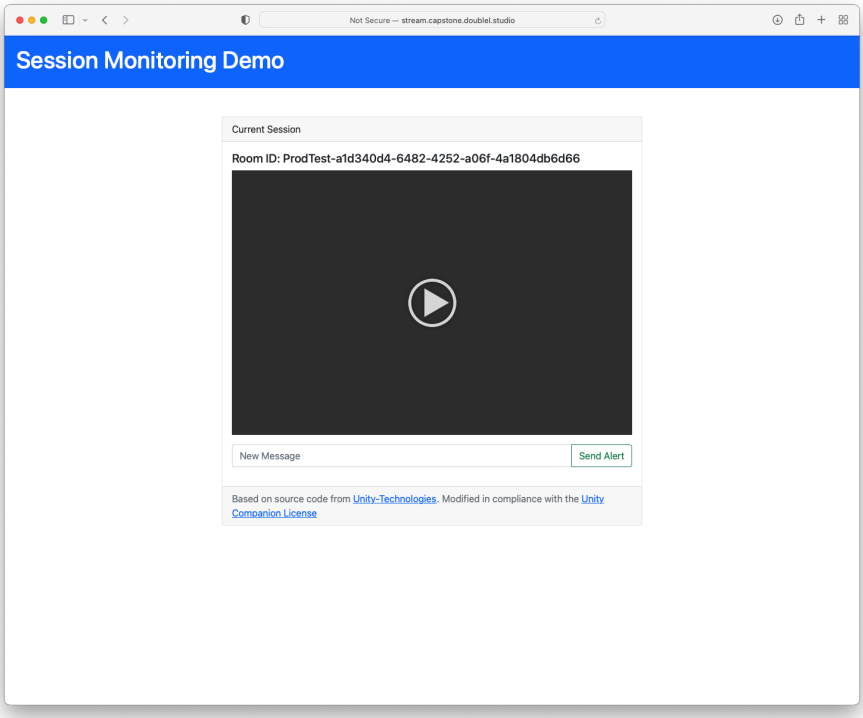

Figure 5.6: The streaming view presented to a web client.

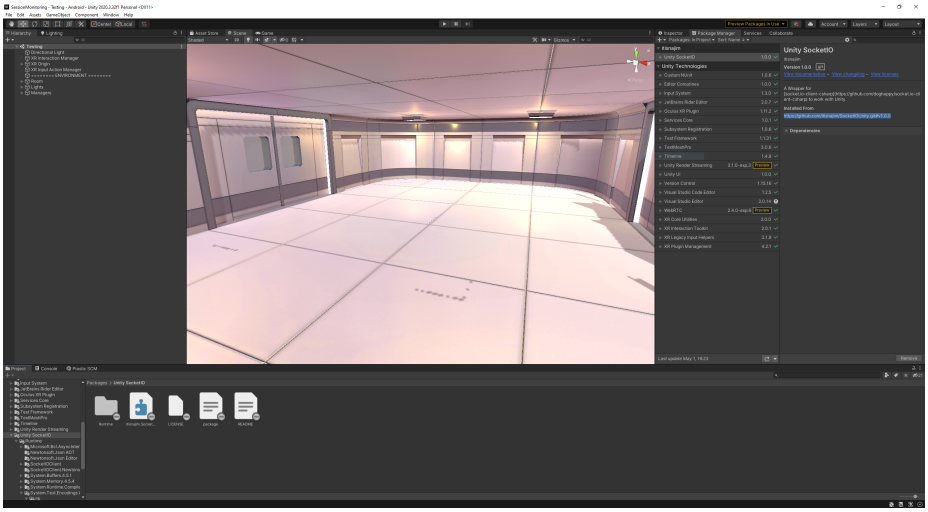

Figure 5.7: The editor view of the unity scene used for demonstration purposes

5.2.4 Client

Finally, we zero in on the Unity Client package. Unity has two experimental packages in their open source pipeline currently: WebRTC and Render Streaming. In very typical Unity fashion, WebRTC is intended to be a low level, thread safe, Unity Implementation of the protocol for projects with specific requirements. Render Streaming on the other hand is a wrapper around the WebRTC package meant to cover most use cases. It is effectively a WebRTC client package that could interoperate with any WebRTC server, just with a simplified configuration.

Both of these packages are labeled as *experimental*, the label Unity gives to packages they aren't even prepared to ship in the beta channel yet. For that reason, they're a little rough around the edges. Documentation is sparse, but what they do provide are useful sample projects for examples of how to use the package. I was able to reverse engineer a working system from these pieces.

The Unity client for this project is 90% demonstration content and 10% actual code relating to the streaming and alert monitoring. I created a basic scene using free assets from the Asset store as seen in figure 5.7 and utilized the Unity XR Interaction Toolkit to make the project VR compatible.

To connect to the Alert system, I utilized an open source library SocketIOUnity[17] for their C# Socket.IO implementation. I created a Prefab game object which connects to the message queue and provides an Unity Event System hook to listen for new messages. Unity Prefabs are akin to classes in the OOP paradigm. They are models for which you can instantiate replicated

[17]https://github.com/itisnajim/SocketIOUnity.gitv1.0.0

GameObjects. This allows the object to be moved from project to project without a loss of data or configuration parameters.

The Render Streaming packages provides two scripts to connect a video source with the WebRTC server. However, when utilizing the primary in-game camera that is synced with the headset, I noticed lag and framerate issues. To fix, I created a second in-game camera which was locked to the position and rotation of the headset camera in-game. This allowed the viewers to see exactly what the headset user was seeing, but help avoid some of the stuttering that was encountered when two intensive receivers were trying to read from the in-game camera simultaneously.

These two core components: The Alert and Stream prefabs were able to extracted into a Unity package for external use. Unity packages are akin to packages or modules in other libraries, importable code which contains all the necessary dependencies.

Chapter 6

Conclusion

At many points throughout development, this project swung between a development task, into an integration task, back to a development task, and concluded somewhere in the middle. I entered with ambitious goals and quickly realized the scope of work to accomplish everything would far out stretch the allotted time to complete the project. With that being said, the biggest goal in the pursuit of this project was to figure out how best to move forward with implementing this feature in a number of projects within the lab. At this end of the project, I have delivered a functional, microservice based architecture which allows for unity clients to stream their in-game cameras to a remote web server, and gives external users the ability to view these streams and send messages to the users in the headsets.

The system I constructed blends off the shelf solutions with a few newly created services into a solid base for future development. Over the course of the past two semesters, I tried out a number of different systems, software, and techniques to accomplish the task at hand. I wanted to look back at some of the paths I explored and lay out why they weren't pursued further. Along with this, I'll be providing some notes on where future work could lead, as well as information for any developers who would like to use this project.

6.1 Revisiting Our Predictions

At the onset of the project, I wrote out some predictions as to how I thought the project would go. How I would implement features, how long each feature would take, and how hard each

feature would be to create. These seemed like useful markers to return to on the back half of the project for two reasons.

Firstly, the act of creating predictions forced me to think critically about how exactly each feature would be implemented. Obviously you don't want every prediction to be incorrect, so there is a vested interest in putting some thought into each one. This in-depth analysis provided useful content for the background section as well as helping catch potential pitfalls early on. Despite some of them being incorrect, I still think this was a good decision as it helped lay the groundwork for deciding which features to prioritize and the sequence in which they should be built.

Alongside this, the other big benefit is that it helps illustrate the development path. My hope is that another developer could look at my predictions, my conclusion, and the finished project and be able to reconstruct the thought process better. There's a few reasons it's so important to me that this take place. Primarily, it can help other developers avoid the potential pitfalls that I fell into and possibly help justify future timelines. For example, the Selective-Forwarding-Unit concept I was interested in for the WebRTC component didn't pan out. This is mostly due to development requirements of that feature being out of scope if the project was going to be finished on time. A future developer looking to continue this work could better understand the magnitude of the work needed for this feature and lay out a better timeline for themselves.

Likewise, I made some decisions regarding the architecture and development patterns I'd use for this project that I'd like to revisit. In the predictions section, I made some strong claims as to why I thought these decisions were the right ones. I think most of those claims came true, but they still need to be revisited to dial in on exactly how they effected the development process.

6.1.1 Development predictions revisited

6.1.1.1 Were micrsoservices the right decision?

As predicted previously, the microservice architecture most definitely introduced some complexity to the code. This is best exemplified by the health check system I needed to implement within the Authorization service. Since the streaming and alert services were separated by the network from the authorization service, there wasn't a simple solution for monitoring that didn't involve bespoke endpoints and an injection of logic and complexity outside of the core task that the service was meant to provide.

With that being said I think the decision was the right one, but for different reasons than I originally thought. The biggest was how much it forced me to generalize some of the services for generic use. The best example in this regard is the Alert service. While I directly link some of this project's implementation to it, it is effectively just a generic chat app backend. Rather than submit Session Room to the room value, you could submit any room name and then send the link to a friend. Since I had the idea of individually functioning microservices as the goal, I wanted to build a service that could be used not only for this project, but for the implementation of other features in different projects later on. I believe this to be the correct decision, because in building a generic messaging queue for use as the Alert service in this project, the lab now has access to a general use, synchronous messaging queue for use in projects elsewhere.

6.1.1.2 Was Python the correct language?

One of the biggest things that happened throughout the course of this project was shifts in design decisions. Whether I planned to develop my own solution when I had previously planned to use an off-the-shelf option, or needed to build up infrastructure I hadn't expected, I usually ended up incurring some lost development time. Likewise, there were some problems I need to prototype out different solutions to. For this reason, I was glad I chose python as it's the language I was most comfortable in. I never felt like I was losing time to tutorials trying to figure out how the language worked. Rather, I could use my knowledge of the language to experiment with different ideas.

The other big benefit was the vastness of 3rd party libraries. Once I had finally landed on a solution, such as the alert apps pivot to a Websocket based implementation, I was able to find the necessary libraries needed to make the system work. For the larger Authorization service I could lean on Django, yet for the smaller alert service Flask served the purpose perfectly. The biggest benefit was that the majority of my work was in Python, so I was able to reuse code snippets across the project to help cut down on development time.

6.1.1.3 Was docker to correct decision for environment management?

Docker ended up being the choice I was most happy with, as it meant I never really needed to deal with environment changes across hosting environments. As long as I could install the docker engine and open some ports the system could run happily. I also really enjoyed the speed at which I was able to try out new techniques. When I switched to Unity's Node.JS webrtc server, I was

able to quickly package the project up in a docker container and test it out without ever having to install Node.JS on my personal development machine. Likewise, when I provisioned my production server for the application, I didn't need to spend time installing packages and ensuring that binaries were available on my path. Instead, I pulled my repository from github, installed docker, moved my environment files[1] over, and spun the project up. Out of my big three meta-development decisions, if I could only have picked one it would without a doubt be the decision to lean on Docker so heavily.

6.1.2 Component predictions revisited

6.1.2.1 The Alert Service

The alert system was the most turbulent of all the services, with multiple changes in its' planned design. Originally, I had been very excited to use a realtime database system in some form. First it was Firebase, and then Supabase (an open source implementation of Firebase). However, soon after setting up Supabase I switched back to a websocket implementation for one big reason: Supabase was providing way more functionality than I needed.

After following the developer documentation to set up a dockerized instance of the system, I was confronted by admin panels, user signups, configurable API panels and more. I played around with the service for a couple hours and then came to the conclusion that as cool as it all was, it was far too opionated for what I needed. In that I mean the system had strong requirements for how it was hosted (what subdomains were required, the reverse proxy) and how you should interact with it (users who correspond to API endpoints). I briefly considered moving forward with Supabase, as a lot of the pieces I was planning to build could have been kitbashed[2] together from different services within Supabase. I decided against that though as it went against my original intentions. Had I moved forward, I would have been building my project around Supabase, not using Supabase to achieve goals I had for the system. With our reliance on Unity3D in the lab already, I wasn't keen to introduce another required piece of software.

Instead, I reverted back to my original plan: A websocket based messaging queue. In my research, I stumbled upon Socket.IO - A websocket based framework for creating real time applications. It had a few niceties such as an inbuilt concept of "messaging rooms" to separate users and had been implemented in all the languages I planned on using. It provided the one piece

[1] https://docs.docker.com/compose/environment-variables/the-env-file
[2] https://en.wikipedia.org/wiki/Kitbashing

of functionality that I needed, an ability to create and maintain a messaging queue, while being modular enough to wedge it into all the places in my software stack I needed it to exist. What I did lose is the out-of-the-box chat history that Supabase would have offered, but the implementation of this feature would be as simple as adding a logging function within the Alert service.

Overall, the Alert service made a huge diversion from it's original plan. I think this diversion was the right decision as it helped maintain the flexibility that the project needed and avoided a rewrite of other components to make them work within the Supabase system.

6.1.2.2 The Streaming Service

The streaming service was also an instance of major changes. Originally I intended to implement a WebRTC media server which all the clients would stream through. This would allow the headset to offload some of the required processing onto the server. The design I wanted to implement in specific was the Selective Forwarding Unit. This design would ingest the streamers video, and duplicate it serverside for every interested viewer.

I tried a number of solutions for this problem, some bespoke and some off-the-shelf. First, I tried to implement the SFU with the python WebRTC library AIOrtc[3]. I had found an implementation of the SFU on github that I attempted to reverse engineer, but found the documentation for the library lackluster. I then tried two different options out: Pion-SFU, an SFU server written in Go[4] and Janus, a WebRTC media server written in c[5]. I had more success with both of these options, but then discovered the issue which pushed me into my fall back option of implementing a peer-to-peer (P2P) system.

While I had an understanding that P2P communication was easier to implement in WebRTC than a media server option, I wasn't fully aware until I began development why exactly. What I found was that standing up the media server was only half the battle. There was additional infrastructure work that needed to take place to make it possible for the media server to run and the clients to connect. While I wasn't opposed to working through this process, I quickly realized that the scope of work needed was far larger than the time I had allotted to complete this portion of my project. Seeing that my top level goal was to create a session monitoring system, it seemed that coming out the other end with only a functioning media server would be a failure of the task at hand.

[3]https://aiortc.readthedocs.io/en/latest/
[4]https://github.com/pion/ion-sfu
[5]https://janus.conf.meetecho.com

Likewise, while I had wanted to implement the SFU for performance reasons, I realized this was a pre-optimization pitfall. In all of the scenarios where we want to use this technology, there would be a maximum of 3 viewers per stream. In that scenario, the headset would be more than capable of managing all of the P2P connections.

For all of these reasons, I ended up using an open source, P2P WebRTC implementation offered by Unity as part of their Render Streaming package. With this source code in hand, I was able to mold together a functioning WebRTC web client and signaling server for communication with the Unity client. More functionality was added on top of this, but that is touched on in the Demonstration Site section. In conclusion, this service faced a major change that I wasn't expecting and definitely leaves behind pathways for future work. But it does help accomplish the task we set out to complete.

6.1.2.3 The Authorization Service

Development of the Authorization service went more or less as predicted. In my predictions I laid out the concept of a `Session` object for making periods of time when user's streamed more permanent. The biggest difference afterwards is that the Session object necessitated what became known as Session Rooms, Active Projects, and Health Checks to accomplish the goals I was interested in. That being said there were no major shifts in development that come to mind. One could argue that the inclusion of the job scheduler may count, but seeing as I had covered the idea of sessions in the prediction that feels more like an implementation detail. The only other major change is that I had anticipated sub-classing Django's internal API key library for use, but instead used a 3rd party library which had already done exactly that. All in, I'd say the Authorization service had the most accurate set of predictions laid out.

6.1.2.4 The Demonstration Site

The predictions for the demonstration site were also mostly on track. The biggest changes being how the site itself was hosted. Originally I had planned on running the site as a collection of static files behind the main instance of nginx. Instead, it ended up living within the Stream service and being hosted out of the Node.JS app which runs that feature. It really seemed like the bulk of the functionality involved in the demonstration site rested on the WebRTC stream, so it felt natural that it be bundled in with the rest of the Streaming service. As for the Alert chat box, I was able

to pull almost all of that code from the chat app I built to test out the Alert service itself. In total, the site consists of an HTML and javascript file. I used bootstrap for all of the styling, so I was able to do away with the need for any seperate CSS work.

Interestingly enough, the most unexpected part of the project came in the amount of Javascript I needed to learn. I had always used Javascript for basic web development, but this project necessitated some basic DOM manipulation, data interpretation, and web requests. None of these things I had much experience with in Javascript. So while not necessarily contained entirely within any of the predictions, I would say that the amount of time I spent within the browser console and learning about how Javascript works as a language most surprised me.

6.1.2.5 The Unity Client

The unity client ended up taking about as long as expected, just with different underpinnings. As anticipated, it was more or less a wrapper creation task, with the Render Streaming library handling streaming and and the SocketIOUnity library helping connect to the Alert instance. Both of these libraries contained either pregenerated scripts or modules which allowed me to create easily configurable objects to help with connecting back to the streaming service. The final product outside of the demo scene is a Unity Package which can be installed in any unity project to enable this functionality, with all dependencies included.

6.2 For future developers

To finish out this retrospective, I wanted to briefly touch on some of the areas I wish I could have tweaked as well as where future work could head next. A large portion of the configuration information can be found within the repository in the form of READMEs, but there are a few things that I do wish I could have changed.

6.2.1 What I'd do differently

If I was starting this project from scratch, the two biggest things I would change would be including a command line interface (CLI) in pretty much all of the web applications and creating a project wide event queue.

First, one of the things I really liked about Unity's open source WebRTC application is that it included a command line interface for starting up the project which allowed you to specify a few different parameters such as what port to run on or what protocol to use. I usually accomplish this through the use of environment variables, but a certain level of work is needed to get these into the containers at runtime. This can lead to a chain of complex Dockerfiles and docker-compose commands, strung together by a bash script. Instead, If I could spin up the service from within the Docker-Compose file with some command line arguments, that would have greatly cut down on build times for simple configuration changes. This is mostly a quality of life change, but I do think it would have been a beneficial one.

The other design decision I learned about too late in the process to implement would have been an event queue. As of currently, I utilize HTTP requests for pretty much all of the inter-service communication. An event queue, as described in *Architectural Patterns with Python*, would have been a universally shared message queue connected to only by the project's services [Percival and Gregory, 2020]. This event queue would have allowed applications to broadcast messages that may be pertinent to other services within the stack, or even request actions to take place without needing to make individual calls to each service. If we think back to the Health Check concept, that was borne out of not having a way to know what is occurring within the Alert service from the Authorization service. With this sort of system in place, the Alert service could broadcast an event every time a user left, which the Authorization service could keep track of. In fact, this system probably could have been implemented with the same Socket.IO stack we used for the Alert service itself. While this feature could definitely be added later on, it would necessitate a substantial rewrite of multiple services. Not necessarily impossible, but definitely would have been easier to do during the initial creation of the project.

Those are the two big changes I would make if I could talk to myself a few months ago. Luckily, I don't have many regrets in that form, on the other hand I do have a ton of features I would have loved to implement. Those would make excellent candidates for future work.

6.2.2 What can still be done

If I was planning to spend a few more months with this project, there are a number of next steps I would pursue. Outlined below are the features which still require implementation:

- WebRTC Refactor to utilize a media server: As of now the signaling room only allows for a single streaming session at a time, the use of a media server would allow for the implementation of multiple, simaltanous streaming sessions

- Group Stream Feature: Either through the use of javascript to lay out multiple video players or the creation of a single stream with multiple feeds laid out in a grid, implement some sort of feature which allows a web client to view multiple stream simultaneously.

- Add database logging to the Alert service: Intercept all new events within the Alert service and log their contents to a database. This would allow for a historical view as a user of the system as well as analysis of how users use the system as a developer.

- Migrate inter-service communication to message bus: To help get around some of the complexity caused by the loose coupling of service, have every container connect to a message bus and announce events occuring internally. This could help simplify complexities such as the Health Check system.

6.3 Closing thoughts

At the onset of this project, I felt rather clueless as to how one would actually implement an end-to-end, cross platform streaming solution for use within the Unity3D engine. I had some basic requirements and a lot of anecdotal experience with how I would want something like this to work, but no route forward that I was 100% confident about. On this end of the project, I feel confident I've found a solution that could work for most of the applications we need. Furthermore, the system is flexible. While it does enforce certain requirements in how a Unity client connects with the services, it really doesn't require any large scale changes be made to existing projects to implement it. While there are some further requirements I could make to make the system more stable, that would come at the cost of this flexibility. In its' entirety, I've delivered a system capable of the realtime streaming of an ingame unity camera, synchronous, bi-directional communication over websockets, and the recording of discrete sessions based on API key authentication. This system has been delivered in a microservices architecture, packaged with all the necessary supporting infrastructure. With this system, the lab will be able to offer realtime video streaming alongside any of our new or existing projects, as well as chat and alert features. Likewise, the alert feature will be able to

be utilized in any environment in which we need to transmit data synchronously in order to share messages, track in-game movements, or any other number of use cases for which this type of data transmission is necessary.

As discussed extensively before, the best contributions that lab has made to its' internal tools are the ones which can see use across multiple projects. While we've built some cool one off demos, more often than not they're not reproducible without substantial amounts of work. What I've delivered with this capstone is a simple to install library of code which allows any project with a local internet connection and an in-game camera to implement a first party streaming solution. Along with this, viewers of the stream are able to communicate with the user inside of the headset through synchronous messaging. This final product will benefit current projects in the Mixed Reality Lab greatly and with continued development could become one of the core features that the lab is able to deliver to new projects.

Chapter 7

Appendix

Table 7.1: Search Popularity for the Term "Mixed Reality" in the US since 2010. Source URL here: https://trends.google.com/trends/explore?date=all&geo=US&q=Mixed%20Reality

Year	**Normalized Popularity (0-100)**
2010	1.75
2011	1.92
2012	1.75
2013	1.50
2014	1.50
2015	1.67
2016	5.75
2017	36.42
2018	57.67
2019	39.00
2020	31.92
2021	28.33
2022	21.25

Table 7.2: Steam Store data for yearly releases filtered by VR Motion Controller Support: https://steamdb.info/instantsearch/?refinementList%5BvrCategories%5D%5B0%5D=VR%3A%20Tracked%20Motion%20Controllers

Year	Count Of New Releases
2021	679
2020	794
2019	766
2018	996
2017	1143
2016	645
2015	15
2014	4

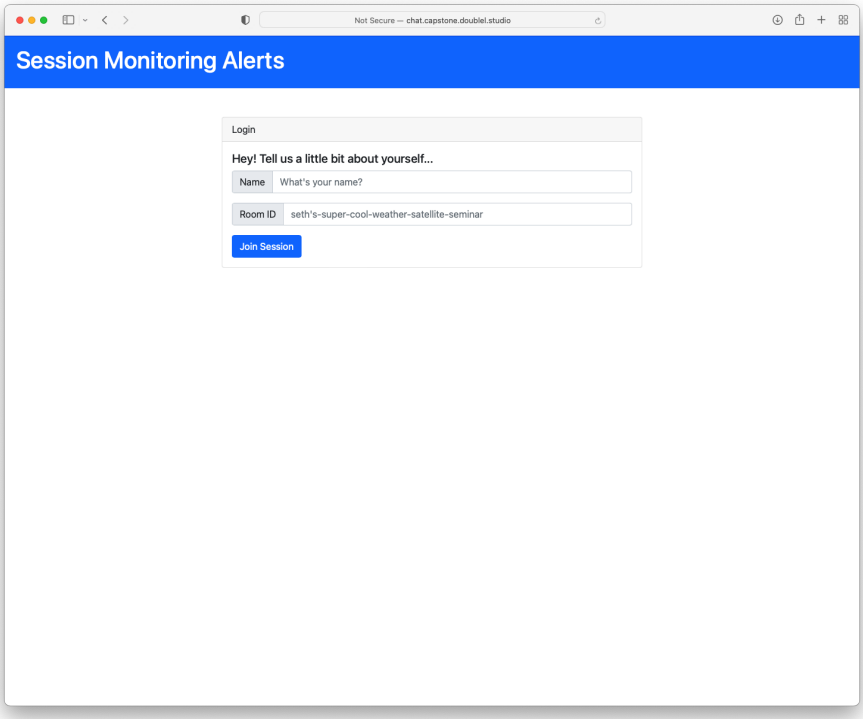

Figure 7.1: What users see when first opening the chat client

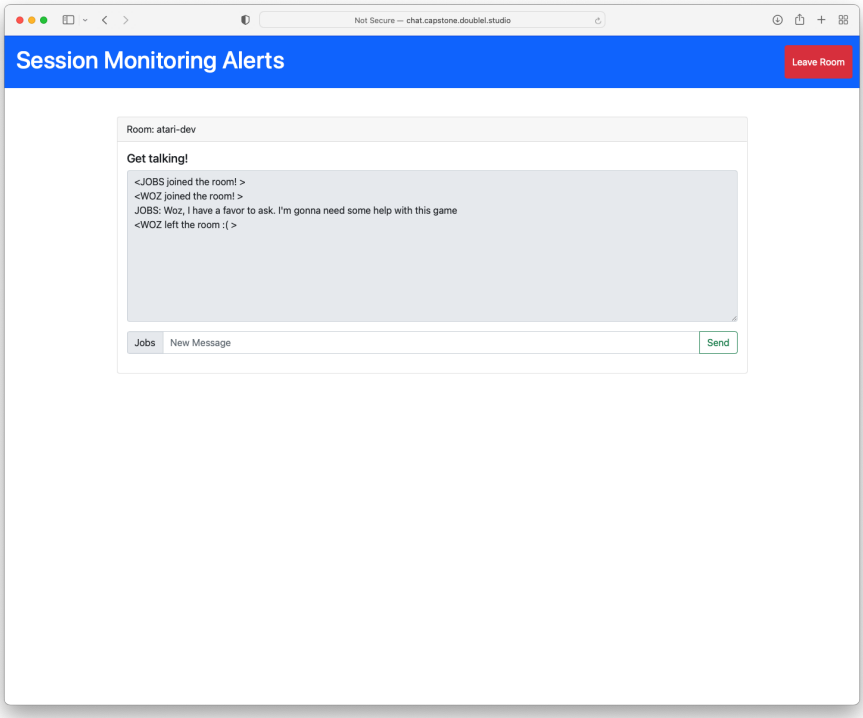

Figure 7.2: What users see once they've joined a room

Bibliography

[viv, 2020] (2020). Tips for setting up the base stations.

[Dubois, 2007] Dubois, P. F. (2007). Guest editor's introduction: Python: Batteries included. *Computing in Science amp; Engineering*, 9(3):7–9.

[Engineering, 2016] Engineering, I. (2016). Web service efficiency at instagram with python.

[Jaramillo et al., 2016] Jaramillo, D., Nguyen, D. V., and Smart, R. (2016). Leveraging microservices architecture by using docker technology. *SoutheastCon 2016*.

[Kleppmann, 2017] Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media, Inc.

[Koulaxidis and Xinogalos, 2022] Koulaxidis, G. and Xinogalos, S. (2022). Improving mobile game performance with basic optimization techniques in unity. *Modelling*, 3(2):201–223.

[Loreto and Romano, 2014] Loreto, S. and Romano, S. P. (2014). *Real-Time Communication with WebRTC*. O'Reilly Media, Inc.

[Meijer, 2019] Meijer, L. (2019). On dots: Entity component system.

[Microsoft, 2022] Microsoft (2022). Contributors to microsoft/mixedrealitytoolkit-unity.

[Montiel, 2018] Montiel, I. (2018). Low coupling, high cohesion.

[Percival and Gregory, 2020] Percival, H. and Gregory, B. (2020). *Architecture patterns with python: Enabling test-driven development, domain-driven design, and event-driven microservices*. O'Reilly Media Inc.

[Remmer et al., 2017] Remmer, M., Denami, M., and Marquet, P. (2017). Why Pokémon GO is the future of school education: Effects of AR on intrinsic motivation of children at elementary school. In *Proceedings of the Virtual Reality International Conference - Laval Virtual 2017*, pages 1–5, Laval France. ACM.

[Sag, 2021] Sag, A. (2021). Ooooh @qualcomm's @cristianoamon is talking about xr leadership and @meta partnership as well as others like @lenovo and says their technologies are the ticket into the metaverse says quest 2 was 10 million units and that was @snapdragon xr2 qid2021 pic.twitter.com/jocq6iwcjw.

[Sensortower.com, 2020] Sensortower.com (2020). Pokémon go catches $5 billion in lifetime revenue in five years.

[Stackoverflow, 2022] Stackoverflow (2022). Stack overflow developer survey 2021.

[Technologies, 2022] Technologies, U. (2022). Important classes - gameobject.

[US Department of Commerce and Administration, 2012] US Department of Commerce, N. O. and Administration, A. (2012). What is lidar.

[VanderPlas, 2014] VanderPlas, J. (2014). Why python is slow: Looking under the hood.

[Veron et al., 1994] Veron, H., Hezel, P. J., and Southard, D. A. (1994). Head-mounted displays for virtual reality. In Lewandowski, R. J., Stephens, W., and Haworth, L. A., editors, *Helmet- and Head-Mounted Displays and Symbology Design Requirements*, volume 2218, pages 41 – 50. International Society for Optics and Photonics, SPIE.

[Wolf, 2021] Wolf, J. (2021). Coroutines - unity amp; c#.