Acapella

Author: Jackson Bogomolny: Electrical and Computer Engineering, Carnegie Mellon University, Ivy Ye: Electrical and Computer Engineering: Carnegie Mellon University, Christy Lee: Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—A system capable of** allowing users to collaboratively create recordings remotely, taking into account latency and delay with real-time monitoring and synchronization with a click track.

***Index Terms*—Audio, Music, Recording, WebSockets**

I. Introduction

IN light of COVID restrictions, ensemble groups now face the quandary of creating music together whilst maintaining safe social-distancing guidelines. As a result, many have turned to remote collaboration; currently available software are varied, but are either limited in function (Soundjack allowing only monitoring) or costly (Audiomovers and Sessionwire requiring monthly subscriptions). Acapella is a response to this need, a free web application that allows users to get together and create a recording any time, in real-time, with any equipment.

To achieve this, Acapella allows up to four users to create a private room, where they can use a DAW interface to record tracks and monitor each other's audio in real-time with the only hardware requirement being a working mic and headphones. The implementation of this monitoring must reduce audio latency to below 100 ms for all users, regardless of their internet connection. To maintain audio quality, there must be a maximum of 5% packet loss over the connection. The combined recording is processed separately. After they are finished playing, users may upload their individual tracks and let the web app sync them to the click track automatically, deviating from the click track by no more than 100 ms after synchronization. These recorded tracks are CD-quality, with a sample rate of 44.1 kHz. Finally, recordings are saved to a webserver and can be downloaded on the users' local harddrive in a lossless format (e.g. .wav).

II. Design Requirements

During recording sessions, users monitor audio from other users in real-time. There is debate over requirements for how fast audio must be streamed to be considered "real-time." A response time of 100 milliseconds appears to be the maximum amount of time perceived as instantaneous [1]. However, musicians, even those less-experienced, are often far more sensitive to small deviations in time from the beat of a song. At 120 beats per minute (fairly typical tempo for a pop song), 100 milliseconds is just under a sixteenth-note (125 milliseconds).

Because of this, we require an absolute maximum latency time of 100 milliseconds for a viable product, but understand that for professional quality recordings, a lower latency is likely needed. To measure latency, a packet is sent between users every two seconds containing the time at which the packet is sent. Upon reception, the sent time is compared to the time the audio is received, and the difference is the latency. Time in this instance is measured using UTC to resolve any differences in time zones between users.

From the same numbers, we can derive a similar requirement for synchronization. That is, once the audio is recorded, it must be "on beat," at least within 100 milliseconds of the beat of the song. The beat of the song is determined by a click track, and the start of a recording determined by onset detection. Frames from the beginning of the file are then added or shaved off depending on whether the note is behind or ahead of the beat. The synchronization error is measured by comparing onset differences between notes of different recordings that should be happening at the same time.

In addition to the timing requirements, the audio quality must also be maintained in monitoring. When streaming audio in real-time, it is possible to drop some packets, but during recording, an excess of dropped packets can cause the monitoring feature to be more harmful than helpful. At a minimum, timing information and pitch information must be retained. To accomplish this, no more than 5% packet loss is acceptable in monitoring, since any more would likely remove necessary timing information. This can be easily measured by the number of packets sent compared to the number of packets lost.

Finally, the system must be easy to use. Anyone familiar with audio editing software should know immediately how to use Acapella without having to read an instruction manual. As such, we have established the somewhat intuitive requirement that no single instruction should take over five seconds for a completely new user to figure out. To test the comparative usefulness of our system to other available options, we survey some ensemble groups who used our web app, asking them to rate our site's features out of 10, compared to other similar applications they have used so far.

METRIC	**VALIDATION**
Latency < 100 ms	For monitoring - Send time (UTC) with packets sent and compare that to the UTC when it is received For synchronization - compare corresponding onset times of each of the uploaded tracks
Audio quality < 5% packet loss	Number of lost packets / number of sent packets.
UI intuitiveness < 5s to navigate	Poll a dozen users both familiar and unfamiliar with DAW interfaces, timing them on performing basic functions such as join room, create track, start recording etc.
comparative usefulness avg satisfaction > 7	survey members of ensemble groups who use our application, asking them to rate various functions, overall audio quality, and overall usefulness from a scale of 1-to-10

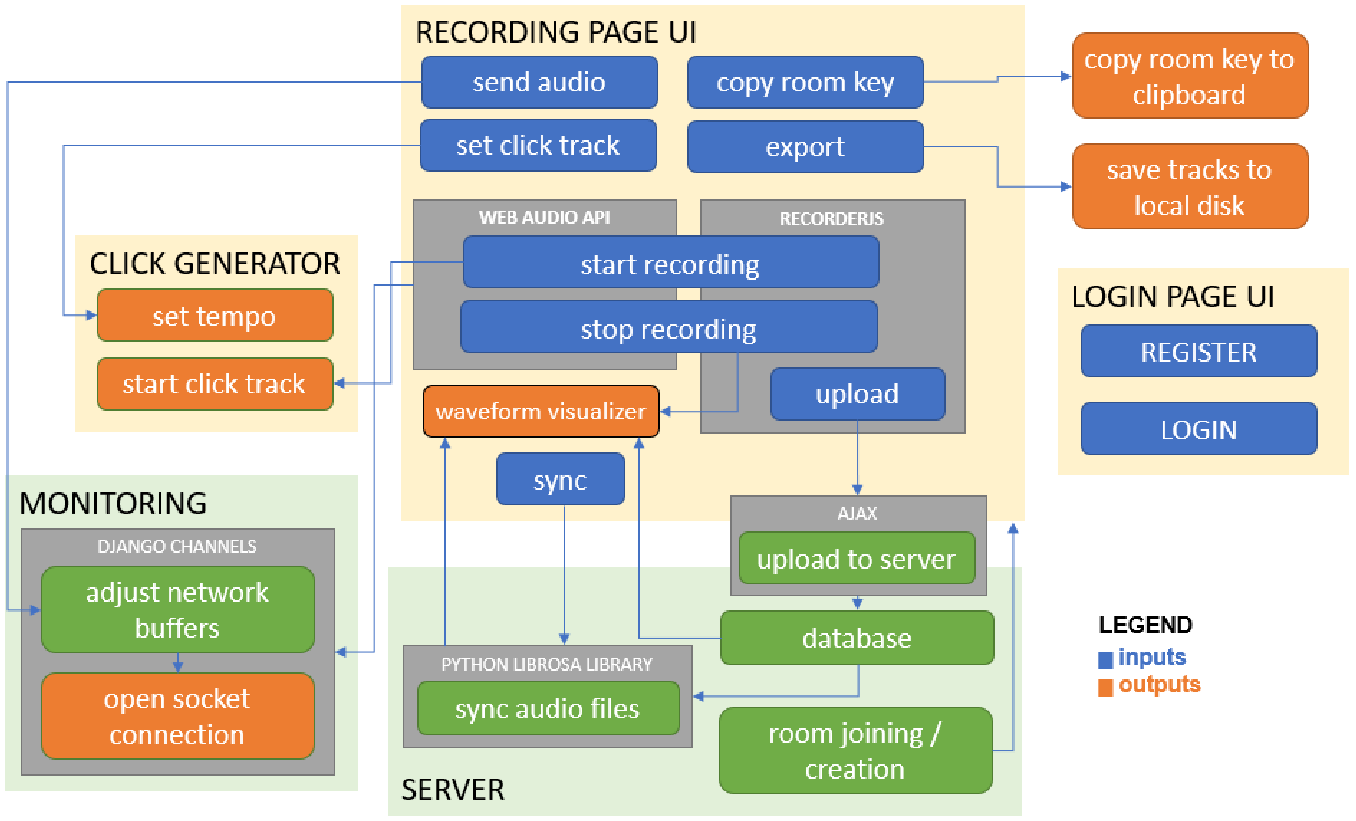


Fig. 1. system block diagram

III. Architecture and/or Principle of Operation

The web application has two primary pages, the login and the UI for recording. When logging in, users are able to create or join rooms with other users, mapped to a specific hash in the URL. This hash is implemented as a Universally Unique Identifier (UUID), and is randomly generated at the time a group is created. In the recording interface, users may set up their low latency peer-to-peer connections for monitoring and record their tracks individually before uploading them all to the server. Afterwards, users of a room can decide which recordings they want to keep and tell the server to sync the beginnings of their selection, using the click track as a metric for timing.

Acapella runs on the Django framework, which provides efficient ORM (Object Relational Mapping) that simplifies storing and accessing databases. One of the essential web application techniques is WebSockets, which enables two-way communications between the server and a browser. Django Channels is critical to handle web sockets and integrate Django ORM with Web Sockets.

Fig. 2 shows how the users and the server interact with each other during the recording process. The users are able to record their audio while monitoring the audio of the other users who are concurrently recording. Django Channel interferes with this functionality.

Once the user uploads the audio file to the server, the server is responsible for integrating all the audio files from each user and dispatching the integrated audio file to each user. Keeping track of timing info of the audio, such as start time, end time, and click track is critical to accurately synchronize audio files from multiple users at the right tempo.

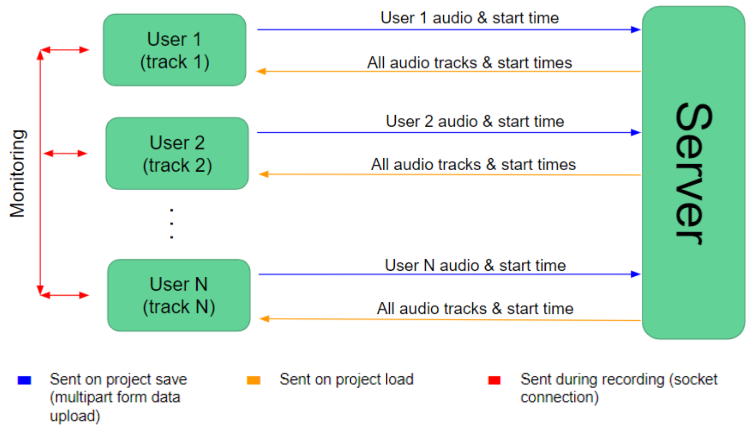

User 1 (track 1)
User 1 audio & start time
All audio tracks & start times
User 2 (track 2)
User 2 audio & start time
All audio tracks & start times
User N (track N)
User N audio & start time
All audio tracks & start time
Monitoring
Server
Sent on project save (multipart form data upload)
Sent on project load
Sent during recording (socket connection)

Fig. 2. server interaction diagram

Monitoring during the recording is accomplished with a low latency peer-to-peer connection, while the actual recording takes place with a separate recording API, to preserve a higher audio quality and sample rate. Recorded audio is then uploaded to the server to be processed.

To create and maintain low latency monitoring, three buffers will be placed between each user and the other (detailed in Fig. 8): the sample buffer, which controls packet size; the network buffer, which controls the number of packets to be sent; and the jitter buffer, which controls the number of packets to be received. Decreasing the packet size allows for lower latency but also increases the risk of losing more data which impacts the quality of the audio. While it is not too important to maintain 100% packet integrity for monitoring, staticky and clipped segments of audio are distracting and may interfere with other musicians' ability to play their part. To circumvent this somewhat, the network and jitter buffer controls the speed of the outgoing and incoming audio correspondingly, allowing the computer some time to complete each packet before being sent at the cost of some latency. The inspiration for this implementation comes from Soundjack, a web app for musicians to rehearse and perform together remotely [2]. Normally users will have to manually adjust each other's buffers to reach a minimized latency, but for our implementation, this process will be automatic. The WebRTC API controls these parameters and decides on connection routes automatically to minimize latency for real-time communication.

Peer-to-peer connections are established by WebSocket signalling and maintained by WebRTC. The moment the group page is loaded, the user's browser generates a session description, or SDP, which contains a list of that user's ICE candidates. In short, these represent all the possible methods for connecting to that user, in the form of public-facing IP/Port combinations. In order to set up a peer-to-peer connection between two users, each user needs the other user's SDP. The process of exchanging SDPs happens over WebSocket connections with the server, and is referred to as *signalling*. When signalling, one user will send an offer requesting to connect to another user. This offer contains the SDP. The other user, upon receiving an offer, sends back an answer containing their SDP. Finally, once both users have each other's SDP, a connection can be established between the two peers using WebRTC, and audio can be sent between users without going through the server first. Fig. 3 below shows the timing for WebSocket signalling.

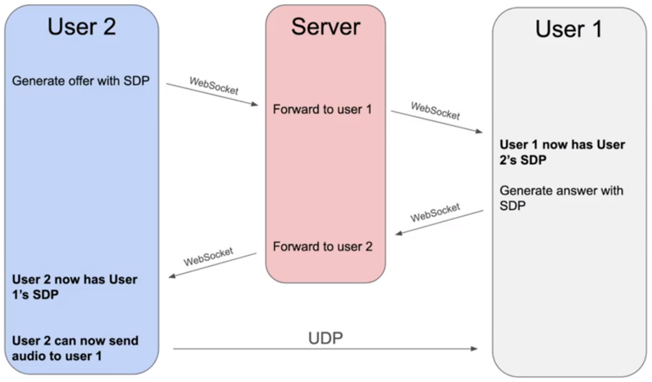


Fig. 3. one-to-one monitoring connection

After the recording is finished, the tracks are uploaded to the server via AJAX, along with the timing of each beat of the click track. The latter information comes from the click generator: when each beat of the click track is played, its timing is obtained and added to an array. Upon upload, all tracks attributed to the room key will appear on the page by hitting refresh. Users can then select the tracks they want to keep and forward to syncing. When the user clicks the sync button, the onsets of each track are matched to the beat times, and each audio file is updated to reflect this.

IV. Design Trade Studies

Previously, we discussed sending the audio to the server and back to the other users as it is being recorded in real time; in other words, combining the real-time monitoring aspect and the recording upload into one process. Unfortunately, upon closer examination, all of this will add massively to the overhead and result in unusable latency, as both the recorded audio and the audio played back for monitoring will have to go through the server.

The above implementation would also make it easier for us to synchronize each chunk of audio as it is recorded, to create an end result account for minute changes in the user's connectivity. However, synchronizing every single chunk would be tedious and oftentimes a waste of computation. As we have tested on other apps that do remote performance, our network will be relatively stable most of the time, making synchronization with each packet unnecessary; most likely the end result will only need a small adjustment to the start time of each track for all of them to line up.

In the end, we decided on separating monitoring and recording into two modules: the first, for monitoring, will not go through the server at all, which will allow configuration for lowest possible latency while sacrificing some of the quality. The second, for recording, will be designed to have maximum possible quality, with latency not being an issue at all since processing of recorded audio does not occur in real-time.

To summarize, the audio is recorded to CD quality specifications, and the high quality file is stored before uploading to the server, while the audio being sent between users for monitoring can be a significantly lower quality to ensure the lowest possible latency between peers.

A. UDP vs TCP

The choice between UDP vs TCP is perhaps the hardest decision with the most important implications in our project. On one hand, UDP has very small overhead (less than half the size header as TCP) making it incredibly useful for driving latency down. Packets are sent without any error checking, as quickly as possible. On the other hand, TCP is far more reliable. Connections are established between users before transmission, packets are guaranteed to be received in the correct order, and the slicing of packets is determined by the network speed, rather than manually as is the case with UDP. Thus, we initially decided to use TCP to send audio between users. acknowledging that UDP would allow for slight improvements in latency. During implementation however, we realized that the WebRTC API which automatically sizes the buffers (as discussed in section III) sends media over UDP. Furthermore, UDP has become a standard in real-time communication specifically because of its latency minimizing capabilities. And finally, the packet loss trade-off is rather small. For these reasons, we ultimately decided to use UDP to send audio between users for monitoring.

B. *HTTPS vs HTTP*

HTTPS is utilizing the same HTTP Protocol. The difference is that HTTPS is running on HTTP Protocol with SSL encryption, making HTTPS much trickier to configure than HTTP. The big advantage of HTTPS is that users do not need to worry about their information and packets broadcasted to malicious interceptors. Furthermore, we had no choice but to configure our website in HTTPS because the RecorderJS API only runs on secure websites and RecorderJS is a crucial part of our project to process user's input [4].

There are numerous ways to set up HTTPS using Apache and AWS (Amazon Web Service) settings. One method is to have an AWS load balancer sitting in between the Apache server and clients. The load balancer will redirect all clients to the HTTPS server. In addition, the load balancer will control traffic by equally distributing the requests among multiple servers. However, since we do not expect much traffic from the outside world and running multiple servers are cost-inefficient, we decided not to have an AWS load balancer in between our Apache Server and clients. Instead, the Apache server will directly listen to one specific port (port 6379) where clients submit requests. In addition, we modified our Apache configuration to always redirect users to HTTPS even if the users access our website through HTTP.

C. *Audio Processing*

Initially we proposed a system which did a significant amount of audio processing on the back end after audio was already uploaded. This would allow for mixing (changes in volume), the combination of tracks, and effects to all be implemented in Python on the server using NumPy or other Python signal processing libraries. After some attempts at audio file upload with Django, we realized that adjusting for different file formats on the back end can be very tedious. Even among files of the same type, encodings can be different. For example, .wav files alone can have different bit depths or be signed or unsigned. In addition, this is just inefficient since all takes (including ones with mistakes that the user has no intention of keeping) would have to be stored on the server, and all computations would have to use server resources.

Because of these issues, we eventually settled on the current model, where most of the audio processing takes place in the user's browser. The Web Media API takes care of input from the microphone and can create any desired file format. Thus, we can send only one format to the server and not have to account for other ones. This new approach also solves the efficiency problem, as files can be discarded by the browser when they are no longer needed. And finally, the Web Media API actually supports quite a lot of sophisticated audio processing: everything from reverb to EQ to adaptive noise cancellation is already implemented. Signal flow is intuitive, and no additional libraries are needed since the Web Media API is supported by your browser already.

V. System Description

The first thing users see is the login/registration page. On registration, the new username and password are sent to the server. Once approved, a user is entered into the SQLite database, and a corresponding User object is created in Django. Alternatively, existing users can simply log in, where their session is mapped to their existing User object. If an existing user object does not exist in the database, an error message is sent back to the user.

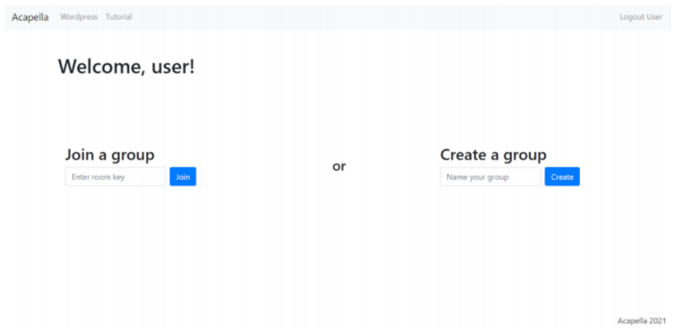


Fig. 4. User interface: landing page

Upon logging in, users are given the option to create a new group or join an already existing one. User's can join a room by obtaining a unique room key (a UUID) from the room's creator, the trailing string of characters in the group's URL

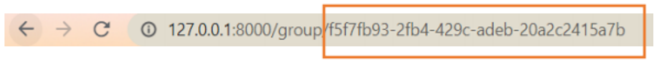


Fig. 5. The room key, displayed in the URL

The DAW interface allows users to interact with the bulk of the functionality with our web app. There are two banks which display the recorded track and all recorded tracks of the room respectively. Above that are buttons which take in user input.

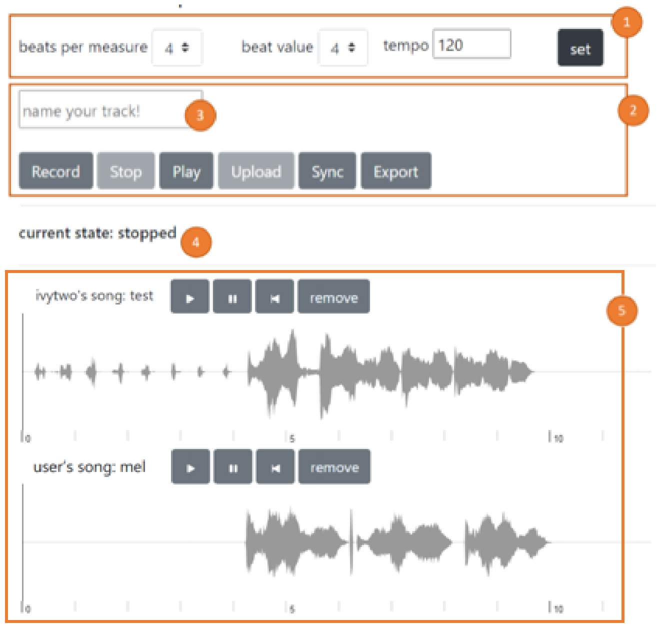


Fig. 6. User interface: recording

1. **Click generator**: The group leader can set a tempo for the recording, by selecting the number of beats per measure and a tempo from 20 to 200 beats per minute. Clicking the set button plays a three-bar sample beat. The click track automatically starts upon recording.
2. **Recording UI**
   a. **Record**: Starts the recording as well as the click track.
   b. **Stop**: The stop button will be enabled after the record button is clicked. Upon stopping, recorded audio is displayed in **(4)** as a waveform.
   c. **Play**: Play back the track you've just recorded.
   d. **Upload**: Uploads the recorded track in **(4)** Users must enter a name for the track in **(3)**. All tracks uploaded from that room will appear in **(5)** upon refreshing the page.
   e. **Sync**: Syncs the start times of all uploaded tracks. Clicking refresh afterwards updates the page to reflect these changes.
   f. **Export**: Creates download links for all uploaded under **(5)**. Recorded tracks are downloaded in .wav format.
5. **Group tracklist:** All uploaded tracks are displayed here and may be updated by hitting refresh. Users can playback and remove any track they do not want to be synced. Removing is permanent and will delete the track from the database.

To preserve recording quality, monitoring is done separately over a peer-to-peer connection. The monitoring console sits at the top of the page and displays each group member.

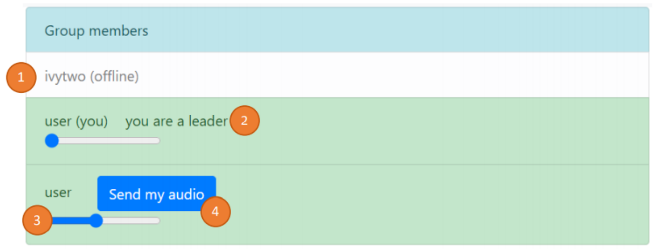


Fig. 7. User interface: monitoring console

1. **Member status**: Each row displays the name of a group member and their status (online / offline).
2. **Leader status**: The creator of the room becomes the room's leader. They are tasked with starting the recording for everyone.
3. **Volume slider**: Adjusts the monitoring volume of said member. This does not affect the volume of recorded audio.
4. **Send my audio**: Opens a peer-to-peer connection with that member, allowing them to hear you in real-time.

Since monitoring between users will be heavily dependent on latency minimization, we can afford to lose some audio quality when monitoring. As mentioned previously, our solution here is to use two different systems for monitoring and recording. This also means the track synchronization will be processed on the server after recording. The tracks, along with the time the audio starts relative to the beginning of the project (for synchronization) are uploaded to the server via AJAX and stored in the server's static file folder. Each uploaded file is mapped to only one project, so that a project can be revisited later, and all the previously uploaded files will be sent back to the user when they open the project again.

A. *Monitoring Over Peer-to-Peer Connections*

To implement monitoring, we began with a proof of concept, creating a connection that supports only two users. To set up monitoring, we used the signalling process discussed in section III and detailed in Fig. 3. We used WebRTC for peer connections which roughly follows the network buffer process as described in Section III.

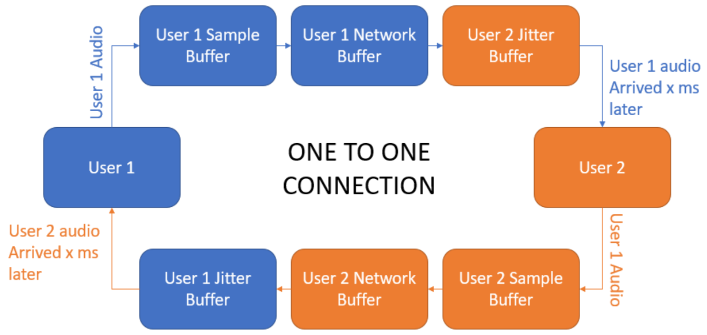


Fig. 8. one-to-one monitoring connection

To review: the sample buffer controls packet size, the network and jitter buffer control outgoing and incoming number of packets correspondingly.

The manual process for doing this involves lowering User 1's sample buffer and User 2's jitter to as low as possible without the audio completely dropping out. Then User 1's network buffer can be raised until the transmitted audio quality is to User 2's liking. To minimize latency some more, User 2 can lower their jitter buffer again. This step and the previous can be repeated until a connection with decent speed and audio quality is established. Finally the entire process is repeated again, but with User 2 transmitting the audio to User 1 [3]. Each user requires different values to their buffers because everybody's ISP and hardware are different: some could be already using a latency-minimizing audio interface while others may just be using a simple USB microphone.

The equation for the amount of times this process must be repeated is:

$$n^2 - n, \text{ } where\ n = number\ of\ users$$

With each additional user, set up time takes increasingly longer and longer. Furthermore, already established connections may need to be modified to account for additional user's connectivity.

During implementation, we realized the complexity of this problem is greater than just the sizes of the buffers. Another factor which is often far more important is the connection path between the two users. Luckily, WebRTC will automatically pick the lowest latency reliable path between two users when setting up peer connections. WebRTC also automates the buffer sizing problem dynamically, allowing for latency minimization to occur in real-time during communication. This is the implementation we opted for.

B. In-Browser Recording/Playback

Audio recording takes place entirely on the front-end. The recording page points to a JavaScript file, which contains all the necessary code for recording. This is accomplished using Web Media API built-ins.

Four things must be initialized before recording. The first and most obvious is storage for data chunks as recording takes place. This can be done with a JavaScript array, which fills only as new data becomes available. The second is a MediaRecorder object, which is JavaScript's construct to capture audio or video. Our system needs only the audio component. The third is an HTMLMediaElement object which will point to the URL of the recorded audio once recording is completed. And lastly, the browser needs some kind of way to keep track of state (i.e. STOPPED, RECORDING, or PLAYING) so the recorder doesn't bug out from the user clicking a button one too many times.

The state of the page operates like a finite state machine. It will begin at STOPPED. To change the state, a user can either begin the recording with the "record" button (changing the state to RECORDING), or play back already recorded audio with the "play" button (changing the state to PLAYING). While recording, playback should not be possible, and while playing back, recording should not be possible. Thus, the only state which can be entered while recording or playing is the STOPPED state. This transition happens either manually by stopping the recording or playback with the "stop" button, or automatically at the end of playback.

To store data as it's being recorded, the MediaRecorder needs an event listener for the "dataavailable" event, which triggers every time new data is received from the microphone or other audio input. When this event is triggered, the recorded data is pushed to an array of audio chunks. When recording is finished, the audio chunks array is turned into an audio blob, which is then uploaded to the server with AJAX. Moving the recorded tracks from the front end to the database proved to be tricky; only .webm formatting is supported across all browsers, but we needed our tracks to be in .wav format in order to access them with Python on the backend. In the end, we decided to go with the RecorderJS library [5], a wrapper for the functions described above, which configures our file to the preferred .wav format before uploading to the server.

C. Click Generator

Similar to recording, the click generator is implemented within the browser with the Web Audio API. Two audio contexts are first initialized, each to play short tone at a unique frequency. The first, using a slightly higher frequency signifies the first beat of a bar. The second, at a lower frequency, signifies the following beats. The tempo and beats per measure is obtained by inputs from the user. Once the click track begins, these parameters are used to calculate the timing of each beat to the millisecond and to determine which of the two sounds to be played. The timing of each beat is tracked by pushing Date.now() into an array each time one is being played.

To maintain a stable beat, setTimeout() was used recursively to create a delay between the current beat being played and the next. However, the function is not entirely accurate; timing issues become more and more evident after successive setTimeout() calls. If a beat is delayed for longer than the set tempo for any reason, the following beats will inherit the delay, causing the metronome to stray further from the correct timing the longer it runs. To mitigate this issue, the delay for every other beat was recalculated to make up for any extra time it took for the previous beat to be played.

D. Syncing

After the users have selected the tracks they want to keep, a request is sent to the server containing those file names. Furthermore, the server fetches the beat timings and tempo previously uploaded by the room leader from a dictionary. With this data, we can now begin extracting timing information from the recordings.

We used the [6] Librosa Python library to extract musical features from our audio, specifically Librosa's onset detection function to parse the beginnings of each recording. Normally recordings have a period of silence before the musician begins playing, however onset detection will react to the smallest bit of noise, irregardless of whether it was intentional; thus, we needed a way of differentiating between these onsets and the onsets of the actual notes being played.

To do this, we used another one of Librosa's functions to calculate the root mean square (RMS) of energy for the entire audio track. Since accidental bumps are generally lower in volume (and therefore lower in energy), we then use thresholding to remove the time values where RMS was below 0.1. The onset times which are also inside of this RMS threshold are the onset times of the intended recording.

After obtaining the onset times, we begin matching the first onset to the closest beat, by finding the difference between two adjacent ones. There is a special case with pick-up notes that we must make sure to take into account, as it would be unfortunate if the pickup-note was matched with other parts that only played at the beginning of the first bar. To do this, we've set a Tolerance parameter for how close the onset has to be from a beat to be considered on the beat; if the onset fails to meet that metric, we look at the next. This tolerance is set to about a third of the tempo, to account for triple meter, although it defaults to 0.2 if the tempo is set to a little over 120 bpm. This value was to be modified depending on the monitoring latency metric we get after deployment, however as we were not able to get the monitoring working with deployment, it is left as it is.

Afterwards, the difference between the onset and corresponding beat is converted from seconds to samples which are then added or subtracted from the beginning of the .wav file in order to align it to the beat. This file is then updated in the database to reflect this change.

With that being said, different instruments have different envelope characteristics, and not all of them are easily detected with RMS thresholding. For instance, a drum playing a sparse beat may behave a lot like background noise. For this scenario, this algorithm may not be as reliable. A more robust algorithm would probably involve machine learning to detect the instrument being played, but this is beyond the scope of our project.

E. *Cloud Deployment*

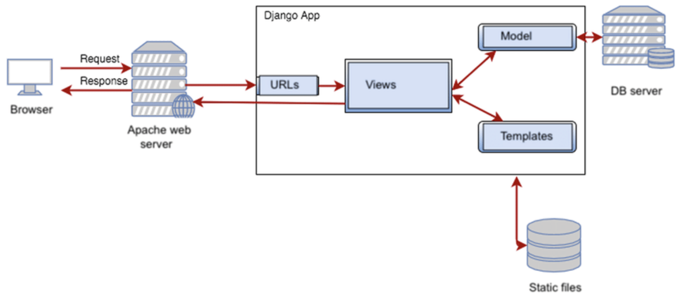


Fig. 9. Django Apache Architecture [8]

In our production setting, we utilized Apache version 2 and Django version 3.1.7. Apache web server listens to browser's HTTPS request and forwards the requests to the application server, Django. Since Django is a python-based application, HTTP requests must be translated in a form that Django can understand. One of the modules from Apache, mod_wsgi, helps communicating with Django Application over WSGI (Web Server Gateway Interface) specification. Then, Django will parse the request and look up the function in urls.py file. views.py file will perform tasks by communicating with the database through models.py and rendering html in static files [9].

Although Apache is a perfect fit for the static website development, Apache does not support asynchronous requests like WebSockets and HTTP/2, which are relatively new technologies. Thus, Apache does not support ASGI (Asynchronous Server Gateway Interface), meaning that Apache does not handle one of our main features, real-time communication between users.

In order to make our application available online, we used Amazon Web Service (AWS) to host our Apache and Django server as a SaaS (Software as a service). AWS EC2 is a virtual computer that provides secure, resizable compute capacity, operating system, networking, and storage [10]. For our virtual computer, we utilized Ubuntu 20.04 as our operating system and t2.large as our instance type. Additionally, we utilized Amazon Route 53 to register a domain name and manage our application domains so that users can easily access our website using a domain name instead of an ip address number.

VI. Test and Validation

In section II, we outline our metrics and validation, with justifications for all of our requirements. In this section, we follow those requirements. Though much of the testing we intended to do was never able to be completed due to issues with cloud deployment, we were still able to implement a number of tests which could, at the very least, be used locally to give us a rough idea of our latency and quality.

A. *Results for Latency*

To evaluate latency, we first must define what exactly is being measured. Our requirement is to keep latency below 100 ms, with the justification that this is the rough threshold for a reaction to be perceived as "real-time." Thus, we should define latency to mean the time between when a sound is made by one user, to the time the sound is heard by another user. For our project, however, much of this path cannot be measured. WebRTC hides much of this information under the hood. For example, the time from when a sound is first made to the time when the recorded audio is placed into buffers to send. This would definitely be a valuable time, but we determined that it would be very small compared to the far more important measure of the end-to-end latency just over the network. I.e. the time from when an audio packet is sent by one user to the time that audio packet is received by another. So when we say "latency," we are referring to this measurement.

This end-to-end latency can be measured easily by sending packets at regular intervals containing the time at which they are sent. This time is then compared to the time at which they are received, and the difference is the latency.

Performing this test locally showed latency numbers mostly below 10 ms, far below our threshold. Though this result is very promising, we were unable to finish the portion of cloud deployment dealing with monitoring, so we could not verify that this requirement was met over an internet connection.

Departure from the beat after syncing also follows the same metric. Because the offset from different recordings will probably be affected by the monitoring latency, we have set the latency for synchronization to be less than 100 ms as well. To measure this metric, we ran our code for onset detection on the processed recordings, printed out the onset times, and calculated the difference between recordings of the same session. Doing this locally resulted in an average difference of 20 ms, which is well within the metric. While our code for this part would likely operate the same way after deployment (and thus would not change these results), we cannot say for certain if this is the case.

B. *Results for Audio Quality*

Audio quality tests are far simpler than latency. We are measuring the packet loss rate, which can be calculated from data that WebRTC provides in their stats API. Namely, WebRTC gives the number of packets sent and the number of packets lost. From these numbers, we can divide the number of packets lost by the total number of packets sent. This gives us our packet loss rate.

As described in section II, we require packet loss rate to be below 5% for our application to be considered successful. Offline, the packet loss is virtually 0%, passing the test easily. However, as mentioned above, we were unable to perform this test over an internet connection, and thus we cannot say for certain that our audio quality requirements are met.

C. Results for UI Intuitiveness

To measure the intuitiveness of our user interface, we planned to poll people at random to use our project and ask them to perform simple tasks like creating a group, joining an existing group, recording their audio, beginning monitoring with another user, etc. We would then measure how long it took them to figure out how to complete these tasks. For our UI to be considered sufficiently intuitive, we determined that users should take no more than five seconds to figure out how to complete a simple task.

This testing would require one of two things. The first is a working, deployed version of our project for users to log onto and perform these tests remotely. And the second is the ability to meet with many people in person to perform these tests locally. The first option could not be completed due to unexpected issues and delays with deployment, and due to the COVID pandemic, we had to rule out the second option as well. Thus we could not complete this test.

D. Results for Comparative Usefulness

Similar to the test for intuitiveness, we planned to poll users to measure usefulness, by asking a random selection of users for a very general satisfaction ranking from 1-10. However, for the same reasons as the intuitiveness test, this could not be completed either.

VII. Project Management

A. Schedule

Our team worked separately on our assigned tasks during the week. We planned to deploy our implementation on the cloud every weekend to verify that everything works correctly in practice. This did not pan out however, since deployment requires altering many aspects of the backend. Because of this, we waited until most of the implementation was finished before beginning deployment to the cloud.

The WebSocket signalling and peer-to-peer monitoring began the week of March 15. Also at this time, the click generator feature was implemented to provide timing information for the users and for the server. Timing information from the click generator is important for the server to synchronize audio files from multiple users at the correct tempo, so the click track was implemented prior to synchronization. We created the UI for the sound recording page all throughout the semester as we implemented specific features.

We planned to test our latency metrics early on, but since this requires cloud deployment, we were unable to get to this, as cloud deployment was never fully completed due to challenges with using an asynchronous WebSocket implementation. We planned to first reach our minimum viable product (which performs perfect audio synchronization and transmission between four users) before adding sound editing features, which allow users to adjust the tone, pitch, and speed of their audio file. Since the minimum viable product was never finished in time, audio effects were not implemented either, with the exception of gain control for each user.

A simplified version of our updated Gantt chart can be found in Fig. 9 at the end of the paper.

B. Team Member Responsibilities

Jackson was responsible for in-browser recording and playback, setting up peer-to-peer connections via WebSockets, sending audio between users over these connections, as well as website functionality like team formation. Ivy was responsible for creating the click track generator, uploading audio recordings to the server via AJAX, and synchronizing the audio files to the click track. Christy was responsible for waveform visualization, implementing basic website functionalities like login and registration, and cloud deployment. We planned to work together to integrate and finally implement audio effects such as reverb, EQ, noise cancellation, etc. However, integration was unable to be completed fully due to issues with cloud deployment for an asynchronous WebSocket server.

C. Budget and AWS Credit Usage

ITEM	**COST**
AWS Credits	$50

We would like to thank AWS Cloud Computing Services for providing the resources to get our web app online. Because the product must work on the web, we require an AWS server for hosting our application, handling data transfer, storing the database, and computing resources. High quality audio files can grow large, and we already utilized 0.3 GB of Amazon Elastic Block Storage snapshot storage, which is 30% of usage limit. Although we are able to avoid extra payment for the Block Storage usage for now, we might need to expand the storage limit as more users access our website and store their data.

In order to establish secure communication over a computer network for our website (HTTPS), the first step was to purchase a unique domain name for our website. AWS provides such networking service in 53 Route where we can register our domain name. We paid 12 dollars to register our domain name, www.acapella2021.com, and the domain name will expire one year after our purchase, which is on May 7th, 2022. After registration, we utilized AWS Hosted Zone to connect domain name with our IP address, which ended up costing 0.5 dollars. After the purchase of our domain name and setup for Hosted Zone, there was no additional expense for deployment because the SSL certificate was issued via a free open source website, named ZeroSSL. Also, we chose to utilize an Apache Web Server, which is a free, open source HTTP server that supports Ubuntu operating system.

Figure 11, for AWS cost management, is located at the end of our paper. In reference to the AWS cost management, since May 2nd when we first deployed our server through AWS, daily cost for t2.large instance is 2.2272 dollars. Since we did not stop our EC instance since we launched the server, the expense makes sense. Until May 14th, costs for the EC2 instance sums to 26.454 dollars. In addition, costs for DNS Queries vary everyday between the range of 0.00006 dollars to 0.0003 dollars, but they are relatively negligible compared to the costs for EC2 the instance.

Since our project is designed to work with your computer's built-in audio input, purchasing new hardware cannot be justified. However, between the three of us, we already have a variety of different audio input types, including USB microphones and USB audio interfaces capable of recording with standard microphones or a direct line input.

D. Risk Management

Our team planned to take an iterative and incremental development approach throughout the project. Instead of deploying our web application on cloud right before the deadline, we intended to deploy and test our work on the AWS Cloud Computing Service every week, in order to make sure everything works fine in practice. Iterative web development processes can also help identify the bottleneck of the website, which could potentially increase latency. However, since cloud deployment requires changes in the backend settings, we decided that deploying every week would require a lot of extra and unnecessary work. This came at a cost though; we had issues with cloud deployment that we did not catch until the final days of the project, and they never got resolved. Specifically, we had a lot of trouble trying to get the Apache server to accept asynchronous WebSocket requests. The Django Channels documentation [7] details a number of ways to deploy an ASGI server, but we were unable to get one working in time for our demo.

We also had to be careful about not overspending on AWS credits. If we were to use a t2.large instance, which costs 0.0928 per hour, we could have 538 hours of website running time, which is about 22 days. We will close down the instance once we are done with our project.

One big problem we still have is that the latency could possibly not be low enough to be useful for real-time monitoring. Since we were never able to test latency, this is something we cannot verify will work with our implementation. If, after deployment issues are resolved, latency is above the previously specified threshold of 100 milliseconds, it will not only remove helpful timing cues for other recording musicians, but will also create incorrect timing cues, making the problem worse than simply having no monitoring at all. If latency is too big because too much data is being sent, we can greatly reduce the amount of data by lowering the sample rate of the entire project significantly. If, for example, we change the sample rate from 44.1 kHz to 16 kHz, we cut the amount of data by more than half, while still maintaining enough information for speech to be intelligible. This would make our product not viable for professional quality recordings, however, as a sample rate of at least 40 kHz is required to represent frequencies up to 20 kHz, the top of human hearing range, and a sample rate of 44.1 kHz is the standard in professional audio.

Another potential risk was the WebSocket signalling implementation not working properly at all. We are all new to socket programming, and though we were able to get it working locally, we never got it working on the cloud. In order to manage this risk, we decided that in a worst case scenario, our product can still be useful for recordings without the monitoring feature. This is not at all ideal, but if the monitoring must be removed from the final product, we still need some kind of way to give users pitch cues (they will already have rhythm cues from the click track), which can be done by allowing the user to upload a backing track containing at least one pitched instrument or voice. Audio will then only be sent between users by the server after recording, allowing for our editing features to still work.

VIII. Ethical Issues

While our project is geared towards the very specific area of collaborative music, and we were unable to make even our minimum viable product work within our requirements, we still have a number of ethical concerns we believe are worth addressing for future attempts at this technology.

Firstly, if the concept of our project was ever to replace recording studios and practice spaces, a few issues could arise. Suppose a similar application to Acapella was developed at a large scale. Small recording studios could lose business as our technology gains more users. But perhaps more importantly, since our project requires a very high quality internet connection (to keep latency at a minimum as is required for music or any domain where timing is vital), the process of making music could become more expensive and lead to an upward transfer of wealth to large ISPs with the resources to provide internet fast enough for viable use of Acapella. The existence of our project could also influence people in certain geographical areas without internet connections to be less likely to create music due to high barriers of entry. Though we think this is unlikely to happen with a technology as domain-specific as ours, it is still worth thinking about, since one of our main goals is to make music more accessible; the last thing we want is to have the opposite effect. This is tough to mitigate, since it is just a consequence of the need for high speed internet, but if we were able to implement all of our latency minimizing strategies (such as downsampling monitoring audio), the internet speed requirements could be relaxed a bit to compensate.

Secondly, since our app has a peer-to-peer audio chat function, there are some security issues to think about. One group of issues is related to copyright and creative property. An example of this kind of issue is if the connection is intercepted at any step of the way, people may be able to hear someone else's unreleased music, which could allow them to steal it and lead to legal problems. Another security concern is that our audio chat may not only be used for music purposes. Since we are starting small, we have created a difficult-to-track communication platform which could enable malicious users to plan illegal activities. There are a few approaches we can think of to mitigate this. If we sufficiently design our site entirely to encourage users to make music, it becomes unlikely that people will even think to use it for malicious activity. For the copyright issue however, we can either work hard to improve the security of a group (by encrypting everything being sent, adding passwords to groups, etc), or simply telling users that our app is not intended for music where copyright is a major concern. We have implemented our server to work using HTTPS, which adds a layer of security to connections. But none of us have expertise in the area of computer security, and there are likely many vulnerabilities that we do not have the knowledge to recognize. However, Django application provides default security for Cross-site scripting attacks (XSS) where attackers inject malicious script to a form field. Django will defend the website by escaping potentially harmful HTML special characters.

IX. Related Work

We were inspired by the application called Soundjack, a real time communication system providing any quality and latency relevant parameter to the user. The application provides real time/online jam solutions where musicians interact as if they were in the same room. The difference is that, while Soundjack provides video streaming along with audio streaming in the manner of Zoom, it has no recording or editing interface like the one we have implemented with Acapella. Still, we aim for the similar goal: providing an environment for real-time audio communication between multiple users who remotely practice. Soundjack gave us basic guidelines about what tools are needed for virtual audio recording, which includes a click generator and timing adjustment/synchronization. Although we were not able to learn about the back-end of this application, we got a good idea about what we have to implement.

X. Summary

The system specified in this document is likely to meet the design requirements outlined in sections I and II. We are hopeful that the latency and quality requirements are both possible to be achieved using our implementation of monitoring, while the audio synchronization requirements can be met by post-processing on the server. An obvious limitation is the latency, which even in a best case scenario will still likely be noticeable for professional musicians. This is inherent in web-based audio monitoring, but given more time, certain improvements could be made. For example, the audio sent could be downsampled before sending to reduce the amount of data sent, and then upsampled before playback, or just played back at a different sample rate. The completely in-browser system makes this difficult, since the Web Audio API allows only one sample rate for recording and playback, which is why this is not already a part of the proposed project.

A. Future work

Beyond the semester, a number of additional features could greatly improve the quality of the product. Firstly, cloud deployment of an ASGI server is necessary, and can be done by using Daphne or Heroku instead of Apache. With this addition, we believe that our implementation could meet the minimum requirements we outline in sections I and II.

The DAW could support a much bigger variety of editing tools similar to professional DAWs such as Pro-Tools or Ableton, which both allow for panning, groups of tracks to be processed together, send/return tracks, and most importantly an effects chain for each track or group of tracks. Most audio software uses VST or Audio Units plugins to process audio with effects such as reverb, delay, distortion, compression, and more. Our system has the potential to support this as well. This would also allow for third party developers to create plugins for our site.

B. Lessons Learned

We have learned a few important lessons from this project. For one, this project is heavily reliant on networking protocols for real-time audio communication, which was the most involved part of the process. In our group, none of us had prior experience with networks, which meant that a lot of time had to be spent researching networking protocols, sockets, and their implementations on the web.

Additionally, we were anticipating more interesting audio signal processing when choosing an audio related project. However, just the process of narrowing our scope down to a minimum viable product ended up removing most of the DSP we were initially excited about, including sound synthesis and effects processing. If future student groups are interested in audio signal processing (or any other specific domain), it is important to choose a project whose scope is based in that domain. For example, something like a web synthesizer or web effects processor removes the tough networking problems while maintaining both signal processing and web programming.

And finally, we learned that cloud deployment for asynchronous web applications with a WebSocket layer (using an ASGI) is very different from deploying a standard WSGI web app the way we were taught in the CMU Web Apps course (17-437). Furthermore, best practice would be to keep track of the deployment resources and settings needed for our APIs and code libraries so we know what we need beforehand. If future teams are looking to do any web applications which require asynchronous WebSockets, we would strongly recommend you leave plenty of time for deployment, especially if you have no prior experience deploying ASGI web applications. The Django channels documentation gives a quick example, but leaves out many key details that we were never able to figure out.

Glossary of Acronyms

AJAX - Asynchronous Javascript and XML
API - Application Programming Interface
ASGI - Asynchronous Server Gateway Interface
AWS - Amazon Web Services
DAW - Digital Audio Workstation
EQ - Equalization
HTTP(S) - Hypertext Transfer Protocol (Secure)
ICE - Interactive Connectivity Establishment
IP - Internet Protocol
ISP - Internet Service Provider
ORM - Object Relational Mapping
RMS - Root Mean Square
SDP - Session Description Protocol
TCP - Transmission Control Protocol
UDP - User Datagram Protocol
UI - User Interface
URL - Uniform Resource Locator
UTC - Coordinated Universal Time
UUID - Universally Unique Identifier
VST - Virtual Studio Technology
WebRTC - Web Real-Time Communication
WSGI - Web Server Gateway Interface
XML - Extensible Markup Language

References

[1] Miller, Robert *Response time in man-computer conversational transactions* AFIPS '68 (Fall, part I): Proceedings of the December 9-11, 1968, fall joint computer conference, part I, December 1968

[2] Avant, Bob *Soundjack: Current Thoughts, Understandings and Guidance of this Real - Time Communication and Collaboration Tool.* 2020 Dec 6

[3] Howel, Ian L. DMA *Report of the Voice and Sound Analysis Laboratory Voice Pedagogy: SOUNDJACK GUIDE* 2020 Dec 20

[4] "What Is HTTPS?" CloudFlare, 2020, www.cloudflare.com/learning/ssl/what-is-https.

[5] Diamond, Matt *Recorderjs* 2016 Github Repository https://github.com/mattdiamond/Recorderjs

[6] *Librosa* 2021 Github Repository https://github.com/librosa/librosa

[7] *Django Channels* 2018 https://channels.readthedocs.io/en/latest/

[8] *Bianca, Joao. "Django Structure."* ResearchGate, Apr. 2019, www.researchgate.net/figure/Specific-Django-architecture_fig1_332023947.

[9] Ellingwood, Justin. "How To Serve Django Applications with Apache and Mod_wsgi on Ubuntu 14.04." Community, 2015, www.digitalocean.com/community/tutorials/how-to-serve-django-applications-with-apache-and-mod_wsgi-on-ubuntu-14-0.

[10] "Amazon EC2." AWS, aws.amazon.com/ec2/?ec2-whats-new.sort-by=item.additionalFields.postDateTime&ec2-whats-new.sort-order=desc.

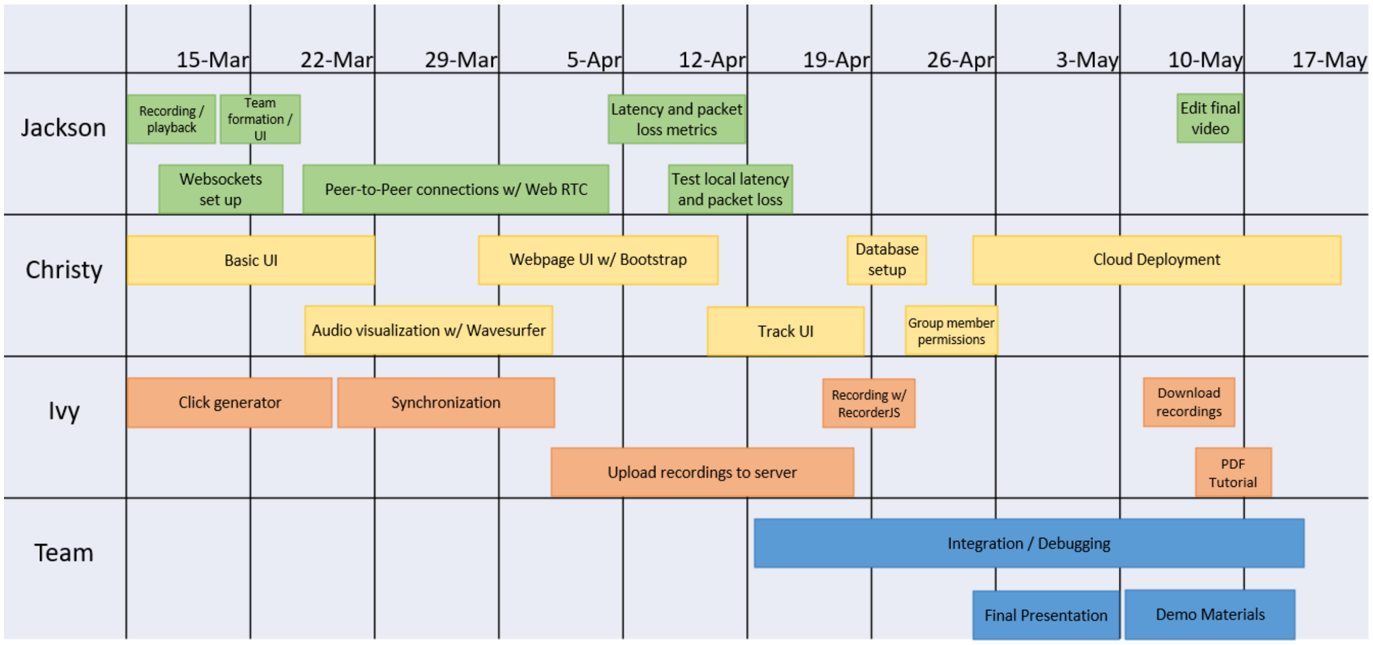
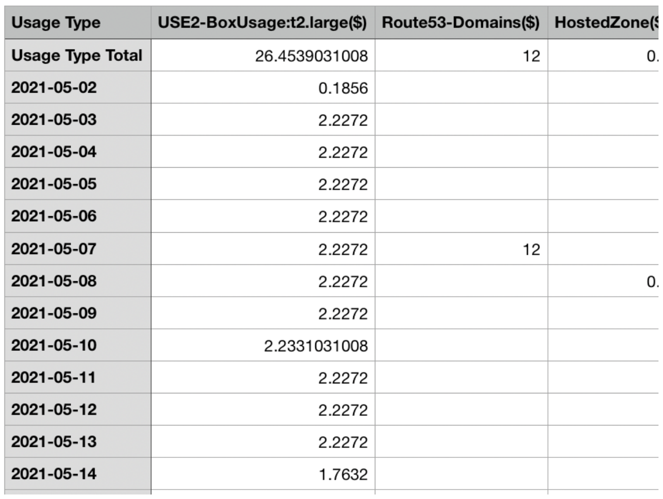


Fig. 10. Schedule and Division of Labor



Usage Type	USE2-BoxUsage:t2.large($)	Route53-Domains($)	HostedZone($
Usage Type Total	26.4539031008	12	0.
2021-05-02	0.1856		
2021-05-03	2.2272		
2021-05-04	2.2272		
2021-05-05	2.2272		
2021-05-06	2.2272		
2021-05-07	2.2272	12	
2021-05-08	2.2272		0.
2021-05-09	2.2272		
2021-05-10	2.2331031008		
2021-05-11	2.2272		
2021-05-12	2.2272		
2021-05-13	2.2272		
2021-05-14	1.7632		

Fig. 11. AWS cost management