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; current 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 will use a DAW interface to create and edit tracks, receiving each other's audio real-time via websockets. The implementation of this monitoring must reduce audio latency to 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 will be processed separately from this monitoring. After they are finished playing, users will upload their individual tracks and the web app will sync them to the click track based on the timing information. These recorded tracks must have a sample rate of 44100 kHz, and must be within 100 ms off the corresponding beat. Finally these recordings will be saved on the users' local harddrive in accessible file formats (.wav, .mp3).

II. Design Requirements

Users will be able to monitor audio from other users in real-time during recording sessions. 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 will 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, each audio packet sent between users will contain the time at which the audio is sent. Upon reception, the time sent will be compared to the time the audio is received, and the difference is the latency. We will use UTC time 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," which as stated above can not be achieved unless each recorded track is aligned within at most 100 milliseconds of the beat of the song. The beat of the song will be determined by the click track, and this synchronization error can be measured by again using the difference between the time of the first click and the first packet sent.

In addition to the timing requirements, the audio quality must also be maintained in monitoring. When streaming audio in real-time, it's 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, 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 received. Corresponding packets may also be compared against each other to check for accuracy.

And finally, the system must be easy to use. Anyone familiar with standard audio editing software should know immediately how to use Acapella without having to read an instruction manual. As such, we've 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 options available, we will survey some ensemble groups who've used our web app, asking them to rate our site's features out of 10, compared to other similar applications they've used so far.

METRIC	**VALIDATION**
Latency < 100 ms	For monitoring - Send time (UTC) with each packet sent and compare that to the UTC when it is received For synchronization - Compare timing of each tick to the timing of the audible click in each audio track
Audio quality < 5% packet loss	Compare number of sent packets with number of received 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

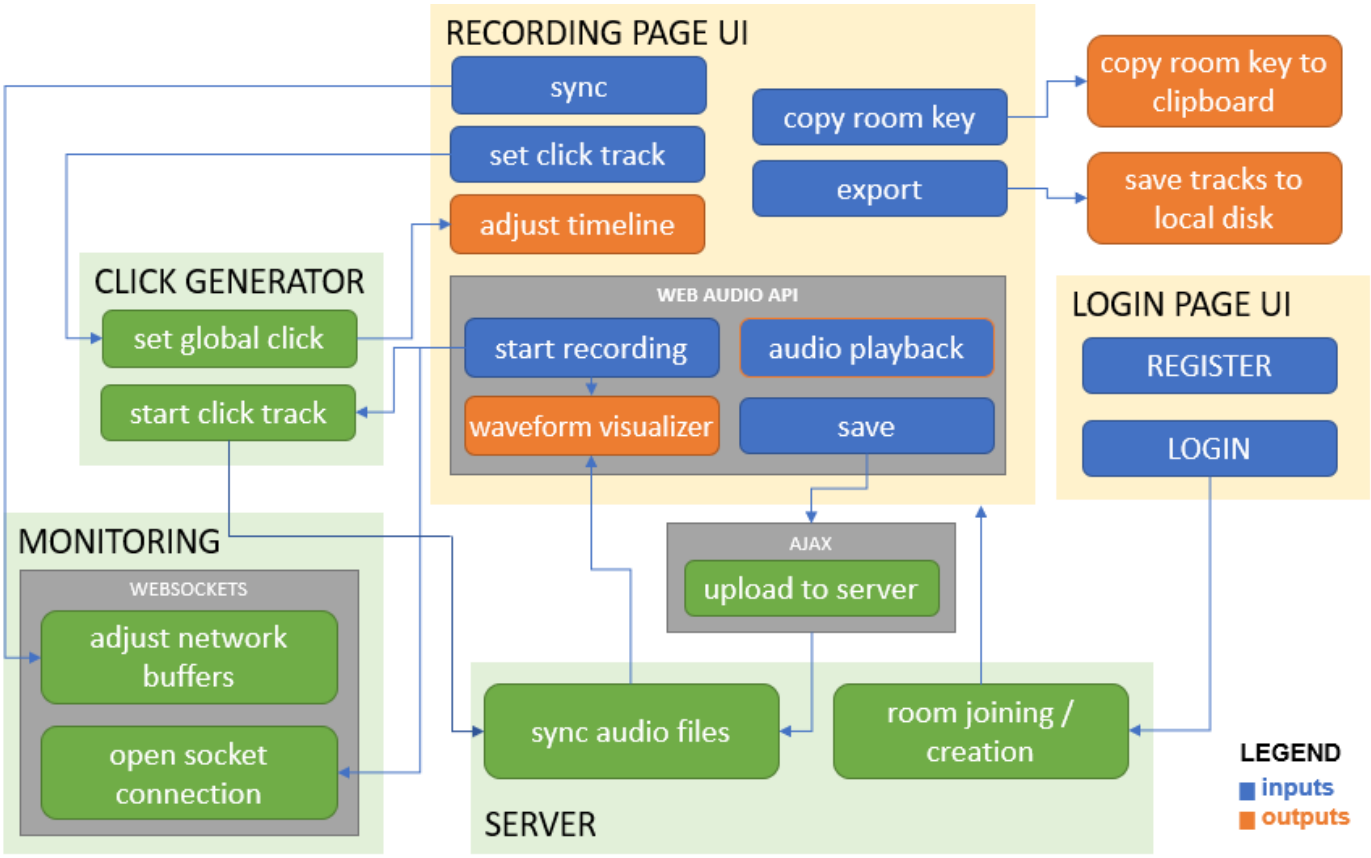

Fig. 1. system block diagram

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 will be able to create or join rooms with other users, mapped to a specific hash in the URL. In the recording interface, they will be able to create tracks for each of their audio inputs, set up their low latency connections, set a click track and finally begin recording. After recording has been finished, their individual tracks will be uploaded to the server via a file form upload, where they will be mixed, using the click track as a timing metric.

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 Web Sockets. Web Sockets enables two-way communications between the server and a browser, resulting in low latency connection. Django Channels will be 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 will be able to record their audio while monitoring (listening) to 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.

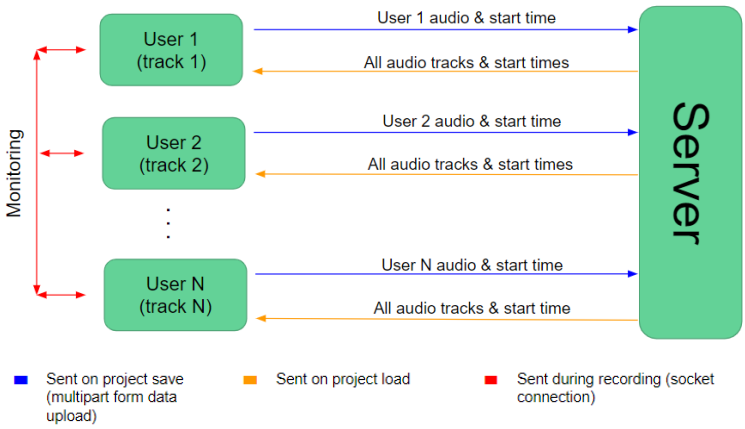

Fig. 2. server interaction diagram

Monitoring during the recording will be accomplished with a TCP connection over websockets. The actual recording will take place separately, to preserve a higher audio quality and sample rate. This will then be uploaded to the server to be processed when finished.

To create and maintain low latency monitoring, three buffers will be placed between each user and the other (Fig 3) : 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 will allow for lower latency, however will also risk losing more data, impacting the quality of the audio. 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 webapp 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.

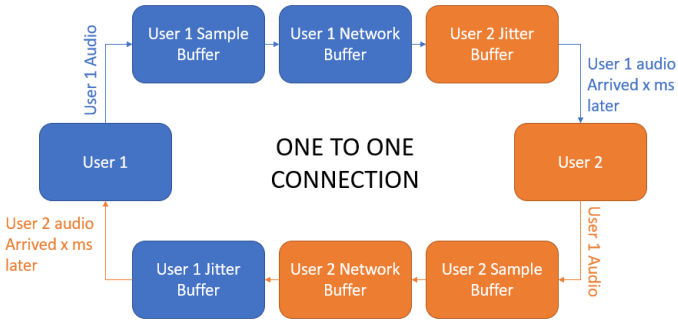

Fig. 3. one-to-one monitoring connection

After the recording is finished, the tracks will be uploaded to the server with Ajax file form upload, along with the timing of each beat of the click track. The latter information comes from the click track. Setting up the click track initializes a global clock that will run according to the inputted BPM. An algorithm will synchronize all these tracks together, by looking at the time codes for each 'tick' of the clock and comparing that to the first noticeable click in each user's recording. After synchronization, these clicks will be detected in the time domain and then filtered out.

IV. Design Trade Studies

Previously, we discussed sending the audio to the server and back to the other users as it's 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, just 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, make 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, 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. During recording, this module will only be focused on retrieving as much data as possible, as the track synchronization will be done afterwards.

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.

A. *UDP vs TCP*

The decision between UDP vs TCP is probably the hardest one 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 however, 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 have decided to use TCP to send audio between users. However, we acknowledge that UDP would allow for slight improvements in latency.

B. *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 and can 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 all 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 automatic noise cancellation. 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 will see is the login/registration page. On registration, the new username and password are sent to the server, and 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 on success, or an error message is sent back to the user on error.

The DAW interface allows users to interact with the bulk of the functionality with our webapp. There is a scrolling timeline in which their recorded tracks will be displayed visualized as waveforms. Above it is a horizontal toolbar that takes in user input (button clicks) to perform the functionality of our DAW.

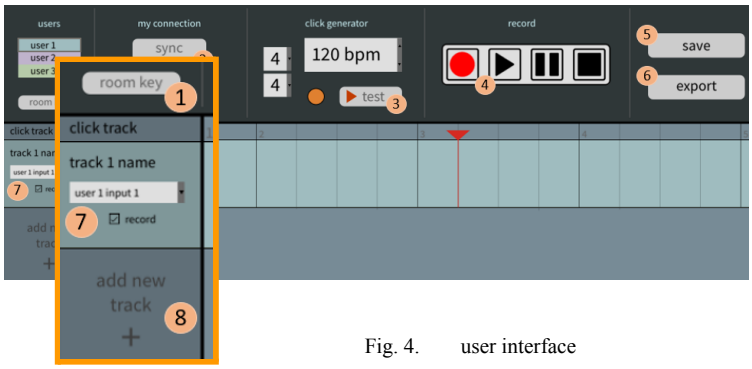

Fig. 4. user interface

1. copies room key url to clipboard to share with other group members
2. sets up latency minimizing socket connection between all peers
3. sets up metronome bpm
4. recording and playback fsm
5. uploads recorded tracks to server
6. export recording to local disk in .wav or .mp3 format
7. track sidebar: manages track input and toggles recording
8. adds new track

To preserve recording quality, monitoring will be done separately over a socket connection. Since monitoring between users will be heavily dependent on low latency, we can afford to lose more audio quality over this socket connection as compared to the actual 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) will be 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 Sockets*

To implement monitoring, we will begin with a proof of concept, creating a connection that supports only two users. We will be using the network buffer process as described in Section III, figure copied here for reference.

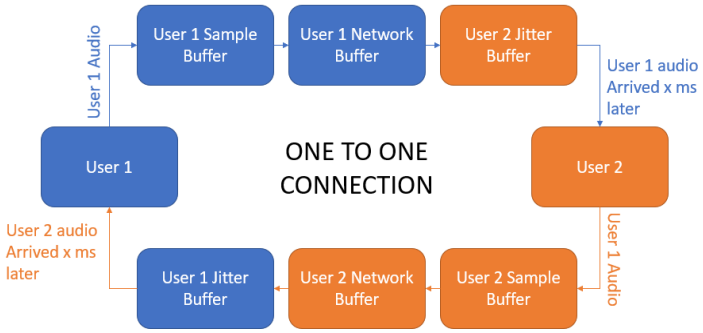

Fig. 5. one-to-one monitoring connection

To review: the sample buffer controls packet size, the network and jitter buffer controls 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:

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

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. To take the load of the user, we plan to automate this.

To do so, we will play a simple tone over each person's connection, calculating latency by comparing the time each packet is sent to the time said packet is received. Because we will be using TCP, packet loss should be automatically detected by our implementation. With each adjustment of a buffer, the program will compare the current latency and packet loss rate to the previous, and undo the change if either takes these values further away from our specifications of 100 ms of latency and 5% packet loss, prioritizing the former over the latter. When the algorithm for between two users has been established to work, we will translate it to work with multiple users, using the user with the slowest connection as a baseline.

B. In-Browser Recording/Playback

Audio recording will all take place on the front-end. The recording page will point to a JavaScript file, which will contain 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).

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 "dataavailable". This will push recorded data to the array of audio chunks only when new data is available. Because this happens within an event listener, it can occur asynchronously, allowing other code to be run while recording. This is critical, since recording must take place simultaneously with sending and receiving data from other users.

C. Click Generator

Setting up the click track involves an HTML form submission, which allows the user to set a tempo from 20 to 200 bpm and pick time signatures with 2, 3, 4, or 6 beats per bar with a eight, quarter, and half note equal to one beat. When pressing the play icon in the metronome, the form is uploaded to the server, where these inputs will be copied to global variables that will determine the timing information for the entirety of the recording. The tempo global variable, specifically will be used to construct a clock that will run at the specified bpm. Each tick of this clock corresponds to each beat of the click track. The python playsound module will be used to alternate between a beat and bar indicator sound.

At the same time, pressing the metronome's play icon will play back a 3 measure sample beat to the users. The click track bar in the DAW UI will adjust to accommodate the selected number of beats per measure. When pressing record, this beat pattern will be looped infinitely until recording stops. Timing information for each tick will be collected with perf_counter(), appended to a list as the recording happens.

VI. Project Management

A. Schedule

Our team will separately work on our assigned tasks during the weekdays. On the weekend, we will deploy our implementation on cloud to verify that everything works correctly in practice. The date that we plan to deploy our code on AWS Cloud is basically our milestones. On the week of March 15, we are aiming to set up Websockets to enable audio transmission between two users. In addition, the click generator feature will be implemented to provide timing information for the users and for the server. Timing information from click generator is important for the server to synchronize audio files from multiple users at the correct tempo. In addition, sound synchronization will be installed to synchronize audio files. Skeleton UI will be implemented for the sound recording page. On the week of March 22, we are going to wrap up the socket audio transmission and test our latency metrics. We are going to come up with a latency minimization algorithm in order to keep a satisfactory latency rate even after we add more users who record simultaneously. After we reach our minimum viable product, which performs perfect audio synchronization and transmission between four users, we are going to add sound editing features, which allow users to adjust the tone, pitch, and speed of their audio file.

B. Team Member Responsibilities

Jackson is responsible for the in-browser recording and playback, as well as sending audio to and from the server. Ivy is responsible for creating visual/auditory tools to correctly measure timing info of the audio and synchronizing the audio files from multiple users to create an ensemble. Christy is responsible for creating visual UI for the recording UI page and implementing convenient website functionalities, such as chatting, team forming, and registration. We are going to work together to implement the socket communication and to reduce latency as low as possible, which involves increasing AWS Cloud resource capacity, possibly downsampling the monitoring audio, setting up the correct network buffer, and decreasing the amount of server-side work. In addition, sound editing functionality will be divided so that each person can work on EQ, pitch, or speed adjustment.

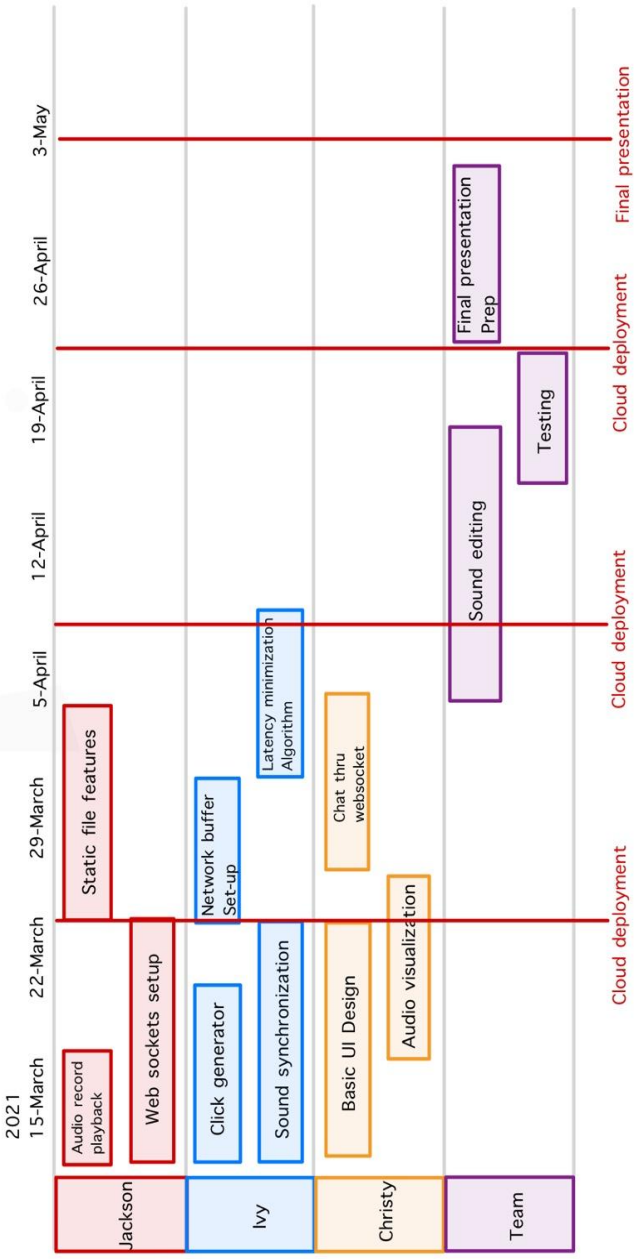

Fig. 6. Schedule and Division of Labor

C. Budget

ITEM	COST
AWS Credits	$50

We are going to spend our budget on utilizing AWS Cloud Computing Services. Because the final product will work on the web, we will require an AWS server for deployment in order to store things like the database of users, projects, and each recorded track saved by the user. High quality audio files can grow large, so we will likely need the extra space provided by the paid servers.

Since our system 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 is going to take an iterative and incremental development approach throughout the project. Instead of deploying our web application on cloud right before the deadline, we are going 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. We also have to be careful about not overspending 50 dollars AWS credits. If we were to use a t2.large instance, which costs 0.0928 per hour, we can have 538 hours of website running time, which is about 22 days. We will make sure to terminate the resources in a timely manner as soon as we verify our implementation works in practice.

One big risk is that the latency will not be low enough to be useful for monitoring while recording. If 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 heard. This would make our product inviable 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.

Another potential risk is the websockets implementation not working properly at all. We are all new to socket programming, and though we would like to figure out how to use it to send audio this semester, there is the possibility that we cannot. In order to manage this risk, we've decided that in a worst case scenario, our product can still be useful for recordings without the monitoring feature entirely. 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, allowing for our editing features to still work.

VII. 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 and singers 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. 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 rpm adjusting controls. 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.

VIII. Summary

The system specified in this document meets the design requirements outlined in sections I and II. The latency and synchronization requirements are both possible to be achieved using the websockets implementation of monitoring, while the audio quality requirements can be met by standard file upload to 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, a separate application could be used for monitoring using UDP instead of TCP, which could minimize latency further. Additionally, 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 for one sample rate for recording and playback, which is why this isn't just 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. 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

Though we are only at the halfway point in the semester, we have learned a few important lessons already. For one, this project is heavily reliant on networking protocols for real-time audio communication, which is likely going to be the most involved part of the process. In our group, none of us had prior experience with networking, 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. 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.

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