GSoC 2018 Proposal for Jitsi Meet: Client-side Recording in Jitsi Meet

Basic Personal Information

- Name: Tianlei Zheng (a.k.a Radium)
- University: The University of Melbourne, Australia
  (3rd year CS undergraduate)
- GitHub: https://github.com/ztl8702/
- Email: ztl8702@msn.com
- My Resume: https://drive.google.com/open?id=1ARslWJnuuk_qxrf3mNmEm5BxE7kbZYZw

My Programming Experience

In my spare time, I like to play around with different technology. And I am also passionate about putting my skills to use, particularly projects that match my value and can create a positive impact.

I have familiarised myself with front-end development (JavaScript, React and Angular) through my involvement in language revitalisation. Specifically:

- Foochow Idioms (intro, source code) which is a Web App + Cordova mobile app for collecting, annotating and querying the idioms and proverbs in the language of Foochow.
- The digitalisation of an old Foochow dictionary. (intro) In this project I experimented with OCR and computer vision, built an MTurk-like crowdsourcing website (ASP.NET Core for backend and jQuery for frontend; link; source code) to let volunteers review the OCR results.

I am also a big fan of the “cross-platform”, “write once run everywhere” idea. That is why I have been learning Xamarin (C# with native Android / iOS bindings) and built a few apps for my own use (here is one). I am not just enjoying learning to use the frameworks but also understanding the underlying paradigms and architecture, such as MVVM and Reactive Programming. Recently, I also started to learn a bit about IPFS and the distributed web, and WebAssembly. I just managed to port a substantial C++ library `librime` to WebAssembly, and now people can type Chinese in the browser without installing any program (demo; source code).

In the past 6 month, I have been (because of my previous research project) mainly working with container orchestration (e.g. Docker) and writing automation scripts (Ansible and Fabric). A while ago, I had been doing some competitive programming (ACM-ICPC). I mainly used C++ and Python for those problems. My school projects mainly use Java.

Generally I am excited to work on frontend, networking, and anything cross-platform.

Project Idea

A custom idea: Implementing client-side recording in Jitsi Meet

Implement distributed client-side recording mechanism in Jitsi Meet, where each device records and only records the audio of the participant on that device (directly using the stream from audio recording device), and after the recordings are merged offline to form a single track.

Background & Use Cases

I have been co-hosting a podcast[1]. One challenge we faced is that we are all located in different cities. The way we currently workaround this is like:

- each of us will have recording software (Audacity and VoiceMeter) running on his/her computer to record only the microphone audio;
- after the audio call, we manually collect the recorded files from all computer and merge them into a single audio track.

The reason we do this instead of having one person recording the entire call is to ensure audio quality so that in the final product we can have high-fidelity audio, and without any jitter. In other words, we want every sound bit recorded to come *directly from the microphones*, rather than transmitted over the Internet. This idea is not uncommon among podcasters who need to collaborate remotely. For example, Scott Hanselman wrote about "don't use the phone/Skype track" in his introductory blog post to podcasting.

Now this works to some extent, but it is cumbersome:

- What if one of us is away from home and can only join using his/her mobile phone?
- What if we want to invite guests to our show? We cannot require every guest to have the same local-recording setup as we do.

We need a solution that is simple to use (preferably "one-click") and customisable. And add this as a feature to Jitsi Meet[2] would empower many audio creators.

Potentially, many scenarios where jittering or even reduced audio quality is not desirable, can benefit from this feature. In such scenarios, centralised recording solutions, either working on a videobridge or as a virtual participant, are not ideal, because they are not guaranteed to receive original-quality, non-intermittent audio streams. Examples include:

[1] In case you are wondering: the show is about language revitalisation in Foochow. It is not live yet, but will be soon.

[2] I plan to start as a fork of Jitsi Meet, and then identify the common piece that can be merged upstream

1. Hosting a podcast by remote collaboration.
2. Interviewing people remotely, with the intention of using the recording in a radio / TV program afterwards. Something BBC often does.[3]
3. Collecting stories and oral history. Something like StoryCorps DIY but without the need to meet physically.
4. Virtual choir, where multiple people sing the same song together but remotely. Audio quality is crucial here.

In all these scenarios the audio / video streams over WebRTC can be used as a way to coordinate and communicate among participants, but are not ideal as the final output. Also, these scenario does not require the recording to be immediately available, so some offline processing after the audio/video session is acceptable.

For this project, I plan to focus on audio recording only. But potentially this will apply to video recording as well.

[3] OK, I admit this one was a live interview. But in many cases interviews does not have to be conducted live and can benefit from a high quality post-processed recording.

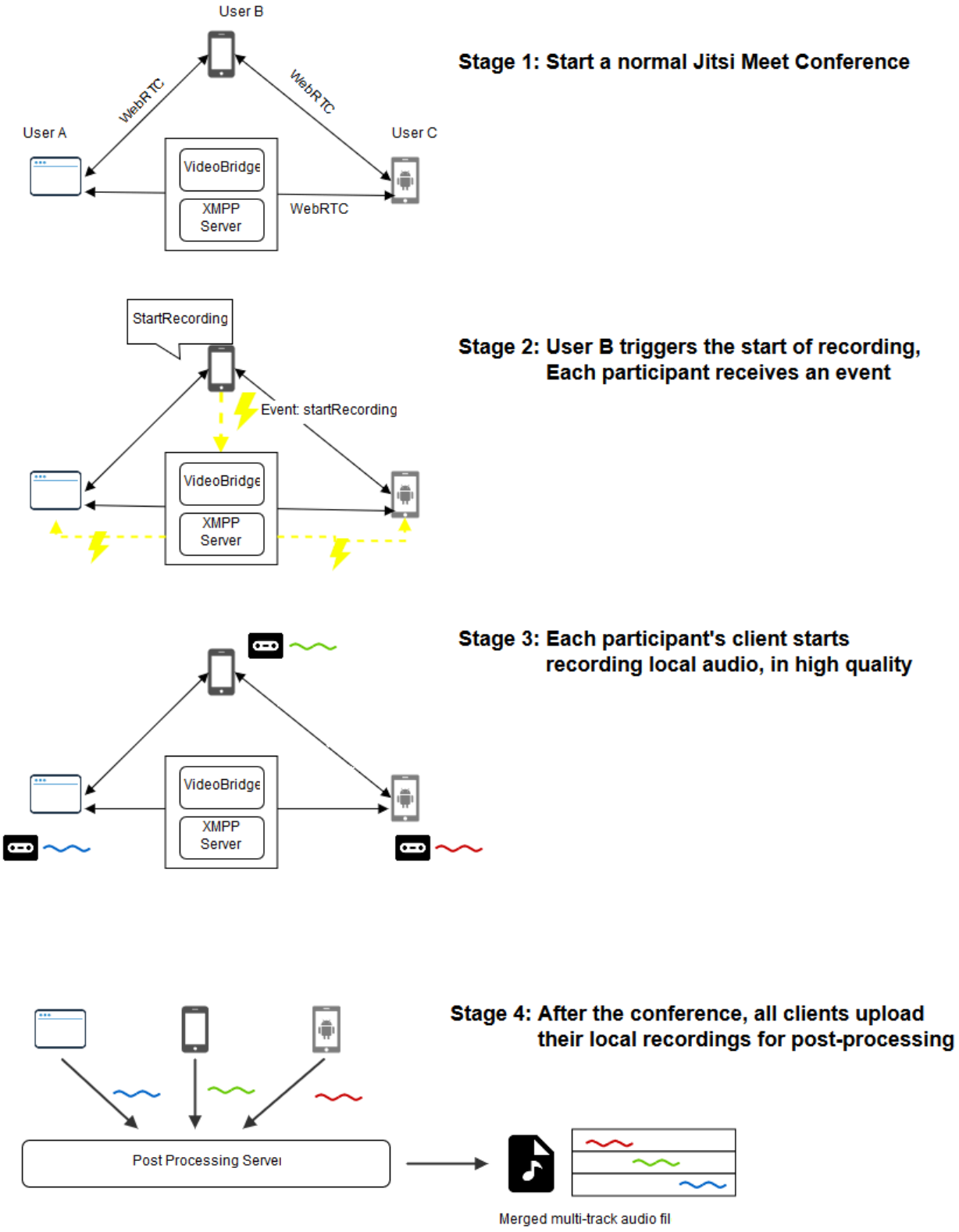

Figure 1: An illustration of how local recording should work

Alternative Projects

If my custom idea is not considered interesting enough for the Jitsi community, I am also happy to work on these ideas:

- Mutual mute – Automatically detect when a video conference has echo caused by two microphones close to each other.
    - I also find this annoying. A naive idea: detect repeat occurrence of certain frequency patterns in the audio. Can be implemented either on clients or on the videobridge.
- Native settings panel – Implement a native settings panel for the Jitsi Meet Electron app.

Why I Choose This Project

Short answer: because this is an idea that came from my own needs.

Long answer:

- I can clearly picture myself (and potentially many others with similar needs) benefit from this feature. Just imagine how much can be done when audio chat over internet can be recorded at high-quality and jitter-free, as if it were recorded face-to-face.
- I love the technology that is driving this: mainly WebRTC and React Native. WebRTC is proving more and more useful, not only in audio and video conference, but also things like p2p file sharing and distributed content delivery network. React Native is the most popular cross-platform framework that I always want to learn and use it on a real world application.
- It is reasonably doable using Jitsi Meet, whose cross-platform and open source codebase means that this feature can be implemented with less “reinventing the wheel” and maximum maintainability.

The Plan: Technical Details

My initial investigations show that both jirecon and jibri are insufficient for this use case, because they both work as a “virtual participant” to the conference, and are subject to network intermittence and do not receive original-quality audio. To be specific, we want the local-recording solution to (ideally) be:

1. **Network-partition tolerant.** Even if one or more clients are disconnected during the conference, the local recording should work normally, and the resulting audio will not have any gaps.
2. **Crash-resistant.** If a client crashes (or browser window accidentally closed), local recording should automatically resume when the client restarts. Only the part of speech during the crash will be lost, the rest of the audio (from the same client) can still be merged with the audio tracks of other people.
3. **No single point of failure.** This is an extension of point 2. In traditional, centralised recording solutions, if the recording server went down, everything during that down time is gone. Local recording is naturally decentralised and audio are easier to recover in case of a failure.

There are three subtasks of this project that need to be implemented in order to have a smooth user experience:

1. **Local recording functionality.** In the browser, on Android and iOS.
2. A **Signalling & synchronisation** mechanism, for coordinating the start and stop of the recording operation, and for making sure that individual-recorded audio clips can be synced up afterwards.
3. **Offline processing**, including collecting recording files from each device and merging them into a single audio file.

For the timeframe of GSoC, I think the minimum is to have a relative stable solution for 1; a naive solution for 2; and working prototype for 3. More can be done if time permits.

I will go through the three subtasks one by one:

Local Audio Recording

Initial experiments have proved it doable in the browser, high-quality local recording can go on simultaneously as the Jitsi Meet conference. I have written a simple Greasemonkey script to demo this: https://gist.github.com/ztl8702/d3b586697b91836e190983b46cc768bf

On mobile platforms, it is trickier. We need to make some changes the React Native Module level`, because Jitsi Meet client code itself does not have direct access to the audio bytes (also it would be not efficient to pass raw audio via React Native JS Bridge). Generally I can think of two types of approaches:

1. Use native audio API to create local recordings, side-by-side with the native WebRTC library. This approach might create conflicts with the WebRTC library.
    - One possible (hacky) solution is to use a separate library like https://github.com/jsierles/react-native-audio to record the audio. Experiments need to be done with this to see if it will create conflict with react-native-webrtc which Jitsi Meet is using.
    - *(Android)* It is not possible to have two Java MediaRecorder instances working at the same time.
    - *(iOS)* It is possible to have multiple AVAudioRecorder at the same time. Therefore, theoretically we can record audio simultaneously as well. Patching the iOS implementation of `react-native-webrtc` will do the trick.
2. Use a "callback" in WebRTC Android/iOS SDK, to fill a buffer when new audio bytes arrive. This approach is safer, and should work nicely together with WebRTC.
    - There are some discussions at https://github.com/oney/react-native-webrtc/issues/99. Someone mentioned a workaround for both iOS and Android but did not show the code.
    - *(Android)* While `react-native-webrtc` does not support recording yet, the `org.webrtc.audio.AudioDeviceModule.SamplesReadyCallback` API in the WebRTC Android SDK seems a good candidate for retrieving audio bytes. This does not require patching or rebuilding the WebRTC library, just some modification in `react-native-webrtc`.
    - *(iOS)* I have not found much examples nor documentations for receiving audio bytes using the WebRTC iOS SDK. Will keep looking.

These are all intended to be easy-to-implement workarounds. Eventually, once `react-native-webrtc` implements the MediaRecorder API of the WebRTC draft, implementations can be replace by the standard ones.

References:

- https://developer.apple.com/library/content/documentation/Audio/Conceptual/AudioSessionProgrammingGuide/AudioSessionBasics/AudioSessionBasics.html
- https://webrtc.googlesource.com/src/+/master/sdk/objc/Framework/Headers/WebRTC/RTCAudioSession.h#118
- https://stackoverflow.com/questions/8492818/multiple-avaudiorecorder-instances
- https://webrtc.googlesource.com/src/+/master/sdk/android/src/java/org/webrtc/audio/WebRtcAudioRecord.java

I am still experimenting with recording on mobile platforms, if Android / iOS implementation turns out the be too difficult for GSoC, we can instead focus on desktop platform (browser and Electron) only, which will still be quite useful.

Signalling & Synchronisation

One of the participants need to signal the start of the recording and all other participants' clients, upon receiving that signal, will need to start their own recording. Same with stopping recording.

Jitsi Meet already has an XMPP based signalling mechanism, so I do not have to reinvent the wheel. Specifically in `lib-meet-jitsi` there are API methods:

- `JitsiConference.sendCommandOnce()` and
- `JitsiConference.addCommandListener()`

We can create custom commands like `` `startLocalRecording` `` and `` `endLocalRecording` ``. This will work across all platforms. I have tested this with the example app (code):

https://imgur.com/TMg6HK4

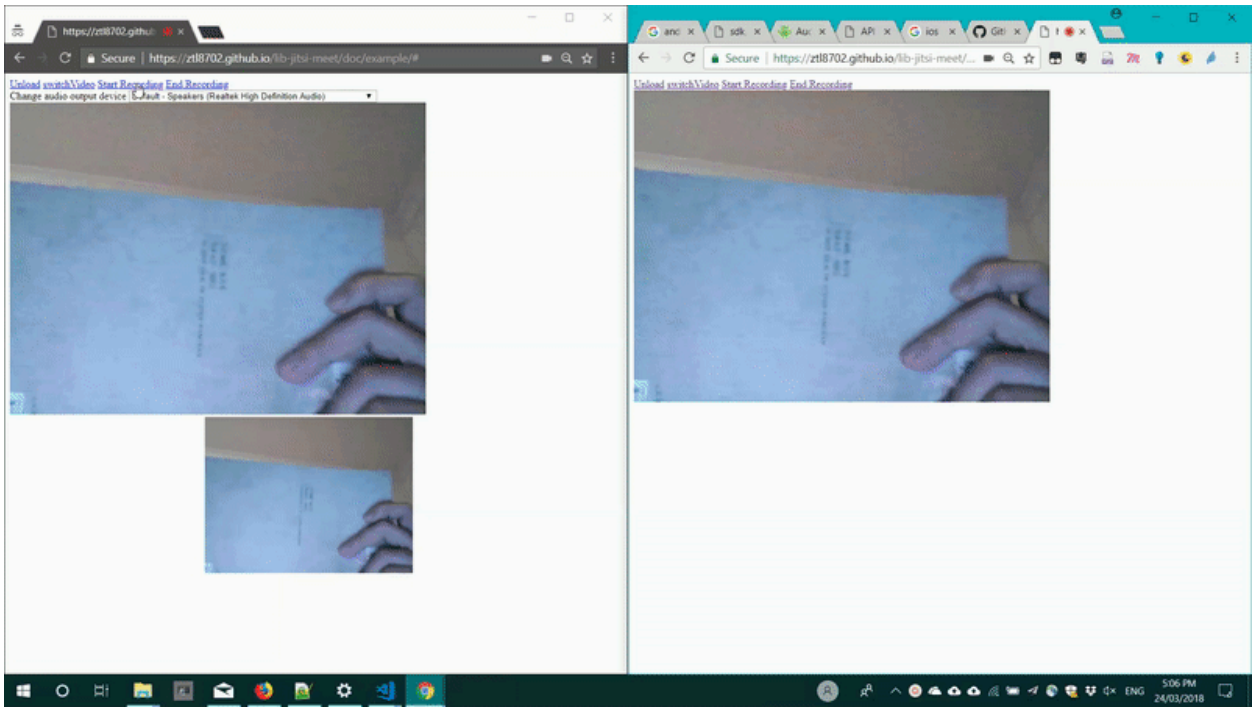

Figure 2: Demo of sending custom events via XMPP using Jitsi Meet API

For synchronisation of the audio, for the sake of ensuring that audio clips can line up. A naive approach would be to first perform a time sync (something like https://github.com/enmasseio/timesync will be useful) of all participating clients. Then each client will periodically stamp the stored audio blobs with the synchronised time. This also

provides redundancy, in case any part of the recorded audio on one client is lost, the rest can still be synced up with other peers.

Offline Processing

After the recording stops or the conference ends, clients should upload their recordings to a post-processing server. As a prototype I will consider implementing a simple HTTP server in Python or Node.js that accepts HTTP uploads (multipart/form-data):

It will expose a simple REST API like:

```
POST /uploadAudio/<conference-id>/<participant-id>/<audio-segment-id>
```

Each client's audio will be segmented by size (e.g. every 1MB) or by duration (e.g. every 30s) and each segment assigned an id (by the client). This will allow for uploading in parallel, removing uploaded segments to save space, and fault tolerance in case some segments get lost due to errors.

The HTTP server will invoke `ffmpeg` to merge the segments into a single audio file, which will also provide an endpoint for users to view or download the (merged) audio recordings.

Other Considerations

- Especially for the browser, we need to provide a mechanism for **persisting the recorded audio segments**, in case the page refreshes or the user accidentally closes the browser window. localStorage might be a solution. Should be accompanied by on-the-fly backup / uploading. This fulfils the crash-resistant requirement. See Figure 3.
- **Divide recorded audio into segments and uploading on the fly.** Since an audio conference might last hours, it might be too much to store for each client. Ideally, the client should be able to save audio segments by a fixed length (e.g. every 30s) and then upload those segments individually. After uploading each segment, the local copy can be deleted to save space.
- **Allow users to change the quality of recording** (sample rate, sample size, etc.). Perhaps as an argument that is sent together with the `` `startLocalRecording` `` custom command. Provide a simple UI for the user to configure their recording quality.
- **Clean abstraction/encapsulation of the local recording logic**, so that the actual recording mechanism on each platform can be easily interchangeable without affecting the rest of the code.
- (Optional) **Integration with the electron app**. There limited codec choices and a upper bound of audio quality (44 kHz) in the browser. With jitsi-meet-electron we can even make use of native Windows/Linux/macOS audio for local recording. In that case, we

can achieve whatever audio quality your microphone / audio interface can provide and the sky is the limit!

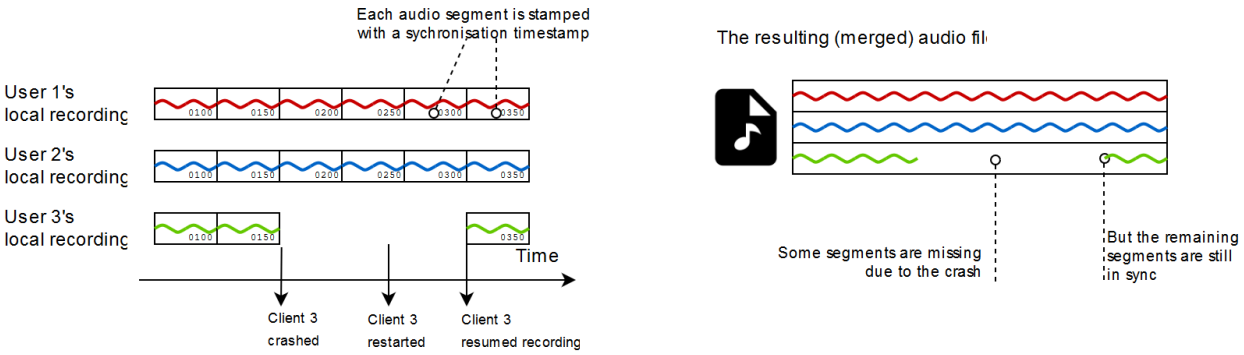

Figure 3: How to be crash-resistant

Deliverables

1. A fork of `react-native-webrtc` containing the workarounds for audio recording on Android and iOS. Make pull requests to upstream if appropriate.
2. A fork of `jitsi-meet` containing:
    a. UI change, such as a button for starting/ending local recording, an entry in the settings page for the URL of the offline processing server.
    b. The client logic for signalling the start/end of recording, uploading recorded files.
    c. The client logic for handling time synchronisation and “stamping” audio segments.
    d. The client logic for persisting not-yet-uploaded local recording segments. (e.g. `localStorage` in the browser and file-based approach on Android / iOS).
3. A simple offline-processing server implementation in Python or Node.js

The Plan: Timeline

There is at least 12 full weeks of time during GSoC 2018. Some initial experiments can also be done during the community bonding phase. The following is a rough execution plan based on my estimation of how long each task will take. It is subject to changes.

Week 1 (15 May -)

- (Local recording) Implement prototype local recording in the web client
- (Offline processing) Implement a simple program to merge local audio files.

Week 2 (21 May -)

- (Synchronisation) Experiment with time synchronisation.
- (Local recording) Implement minimal UI on the web client.

Week 3 (28 May -)

- (Local recording) Experiment with recording on Android.
- (Offline processing) (Deliverable 3) Implement a simple HTTP server.

Week 4 (4 June -)

- (Deliverable 2a, 2b) Have a working browser-to-browser solution.
- (Local recording) Refine web client UI and recording logic.

Week 5 (11 June -) First Evaluation

- (Local recording) Continue experiment with recording on Android.
- (Local recording) (Deliverable 2d) Start implementation of cross-platform segmentation logic
- (Synchronisation & Offline processing) (Deliverable 2c) Refining time synchronisation of audio.

Week 6 (18 June -)

- (Local recording) Experiment with recording on iOS.
- (Local recording) Testing implementation of cross-platform segmentation logic
- (Offline processing) (Deliverable 3) Handle on-the-fly audio segment uploads

Week 7 (25 June -)

- (Local recording) Continue experiment with recording on iOS.
- (Local recording) (Deliverable 1) Add Android recording feature into our fork of `react-native-webrtc`.
- (Offline processing) (Deliverable 3) More fault tolerance, handle missing recording segments

Week 8 (2 July -)

- (Local recording) (Deliverable 1) Add iOS recording feature into our fork of `react-native-webrtc`.
- (Deliverable 2a, 2b) Implement necessary mobile UI components.
- Have a working mobile-to-mobile and mobile-to-browser solution.
- Design new tests in jitsi-meet-torture.

Week 9 (9 July -) Second Evaluation

- (Local recording) More testing with recording on possible faulty situations (browser refresh, mobile app crash, etc.)
- Improving code quality. Test on more devices.

Week 10 (16 July -)

- (Synchronisation) Test synchronisation of audio.
- (Local recording) Submit pull request to merge implementation with upstream `react-native-webrtc`.

Week 11 (23 July -)

- Continue the work on tests.

Week 12 (30 July -)

- Cleaning up code. Submit PR to `jitsi-meet` if appropriate (i.e. all tests green, upstream `react-native-webrtc` PR accepted).

Week 13 (6 August -) Student Evaluation

- Finalising and submitting evaluations

Week 14 (13-15 August) Wrapping up

Technologies Involved and Challenges

For shared logic and the UI part:

- JavaScript: I am very familiar with it. One of my most frequently used language.
- Web Audio APIs: Not particularly familiar, but is gaining speed by reading the documentations.

For mobile clients:

- React Native: I am not quite familiar with it. I have only used React (not Native) before. But I expect to have a working knowledge of RN before the actual code phase. As an exercise, I have tried and built Jitsi Meet's Android client from source.
- Android: I have a working knowledge of Android platform architecture and SDK.
- iOS: I am not particularly familiar with iOS development, largely because I don't really own a Mac. That being said I have at least managed to compiled a Xamarin.iOS keyboard app on a virtual macOS machine (on MacInCloud) and run it on a real iPhone. Given time I believe this can be overcome.
- WebRTC: I am learning it, having read the basic documentations on the web and watched a few WebRTC conference talks. Although this project won't actually touch on the networking part of WebRTC, but a good knowledge of WebRTC internals will help me navigate through library source code.

And depending on the implementation of the offline processing server:

- Some server side Python or Node.js (Express): I am comfortable with it.
- ffmpeg: I have not used extensively before, but its use in this project will be limited to very basic command line invocations and there are many documented examples to learn from.

Time Commitment

The Australian semester works differently from that in the northern hemisphere. There is little I can do about this, but I will try my best to cope with the GSoC schedule and carefully estimate how much time I can devote to this project:

- Community bonding phase & Week 1 - 3: 10-20 hrs/ week (devote time equivalent to a major subject in university)
- Week 4 - 6: 20-30 hrs/week (still have classes, might have exams)
- Week 7 - 10: 40 hrs/ week (as a full time job)
- Week 11 and 12: 20-30 hrs / week in August. (School starts on July 24th. But I won't have much pressure during the first few weeks.)

I do not have any other major summer plans. If I am accepted, I will focus on this project solely.