ELEN 6770 Next Generation Network Final Project

cl3469, Mick Lin
cw2952, Chih-Sheng Wang

Introduction

We target the implementation of a cloud-based network application as our final project. Specifically, we propose to build a live lecture streamer with the capability to let the remote audience who owns the shared link watch the lecture simultaneously. This project fits the requirement of streaming topic in implementation track.

Motivation

Online lectures are widely distributed in modern society, such as columbia CVN, MIT OCW, etc. However, the recording work is not easy. We propose to integrate the lecture recording with browser to broadcast lively and let audience be able to replay the lecture afterwards. It will save the lecturer's effort of repeating same content. Finally, this lets the knowledge distribute.

General approach

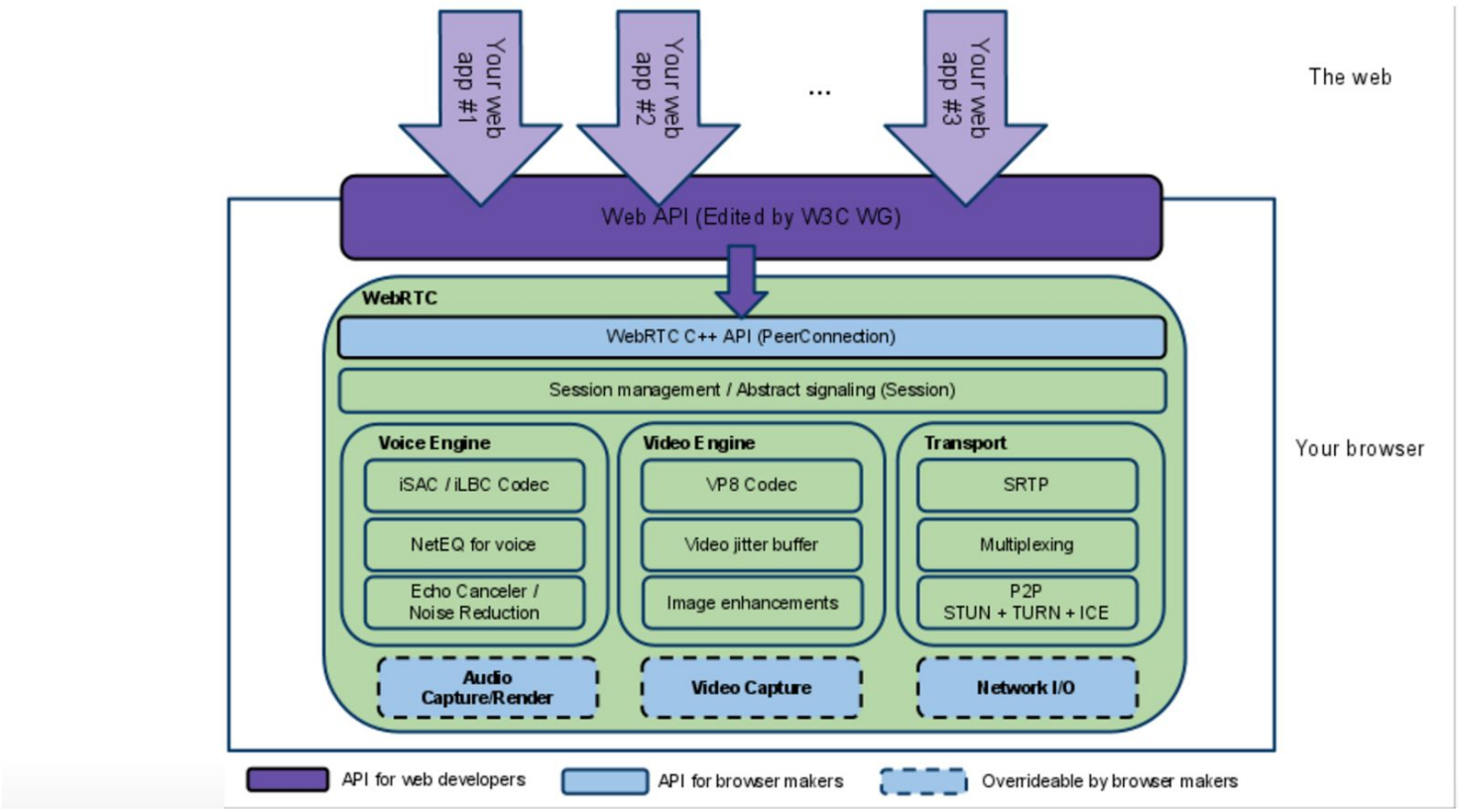
Our goal is to develop a web application using Node.js for the backend, the most popular video API in HTML5 currently is the WebRTC. After a bit of survey we quickly decided to dive into the world of WebRTC.

Although there were some available projects can achieve what we want such as FFserver+FFmpeg (https://www.virag.si/2012/11/streaming-live-webm-video-with-ffmpeg/), we decided not to use these third-party solution but build a streaming server on our own because this project is about what we can learn from building a cloud application.

The slides from Google(https://io13webrtc.appspot.com/#1) firstly give us a brief concept of what we can do and what problems can be addressed with WebRTC, such as that if we're building a P2P streaming application we need to use STUN/TURN server to handle the connections behind the NAT.

There are also some introduction to implement video streaming with WebRTC, such as:
http://forio.com/about/blog/create-video-conference-recorder-using-webrtc/
https://bloggeek.me/webrtc-broadcast/
All these project and introduction gave us a clear picture of what it takes to build a live streaming application.

WebRTC architecture



1.WebRTC

Low latency protocol, built on open standards, uses SRTP for transport, works in all browsers.

2.RTMP

Low latency TCP-based protocol originally built for Flash. Requires Flash player to run in the browser. (note, Tsahi got it wrong that this one is a high latency protocol)

3.HLS

High latency, non standard Apple-backed protocol

4.MPEG-DASH

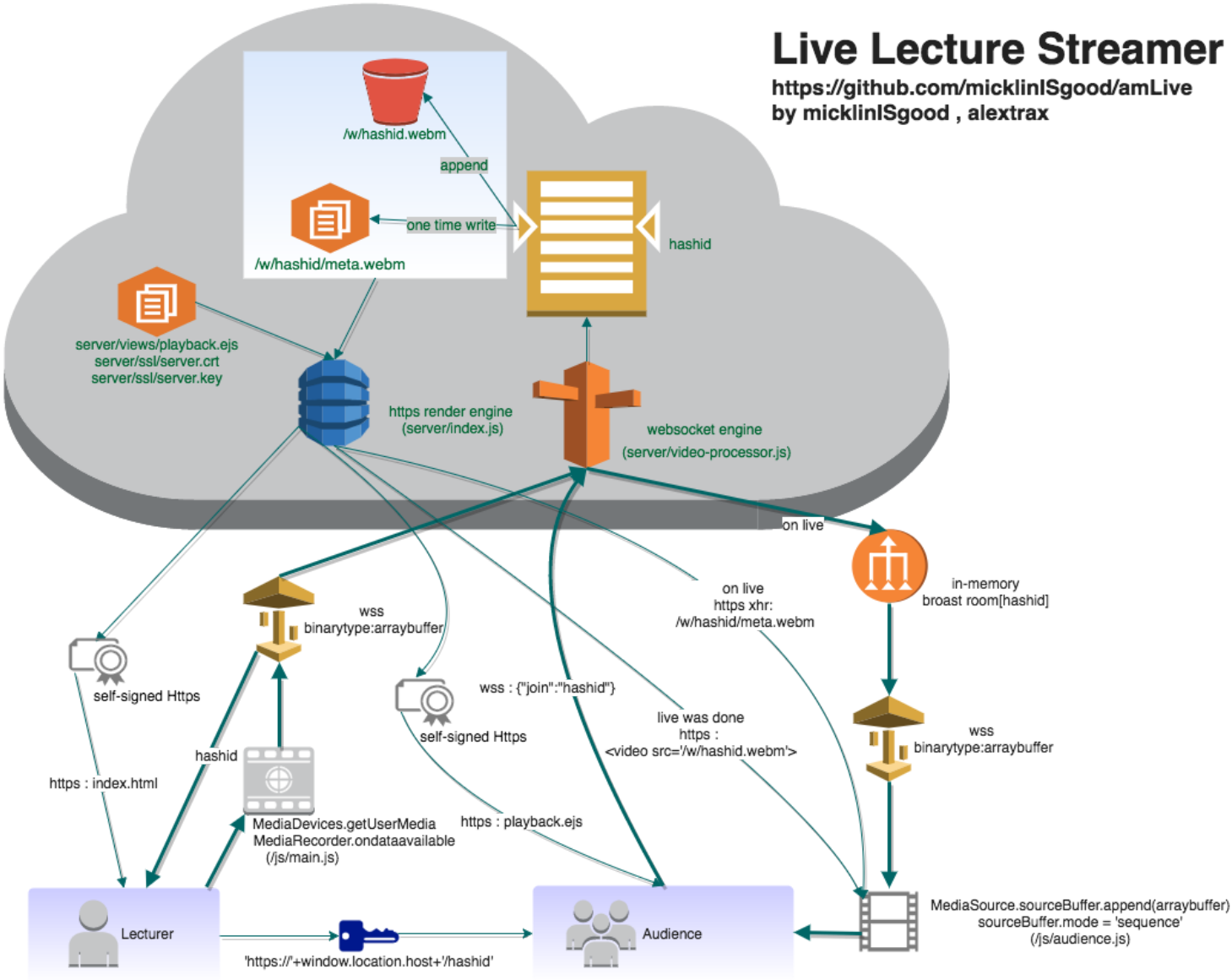
High latency Google-backed web standard, very similar and has many of the same faults as HLS This is a good run down of the main protocols used in live streaming products.

As you can see, the only two that deal well with low latency are RTMP and WebRTC. While RTMP has many solutions exist for scaling Flash based live streaming apps, unfortunately few have figured out how to scale WebRTC in this manner.

WebRTC is a well-defined API. Once you setup the peer connection between two browsers, then you can start the video chat. However, we decide to experience how streaming flow application works from a lower level perspective and build things with fine grained HTML5 components, such as MediaSource.

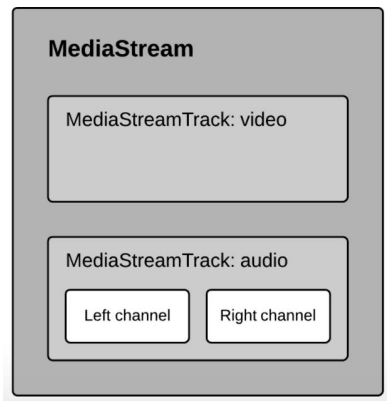
Implementation detail

Architecture diagram:



1.navigator.mediaDevices.getUserMedia():

A MediaStream object has an input and an output that represent the combined input and output of all the object's tracks. The output of the MediaStream controls how the object is rendered, e.g., what is saved if the object is recorded to a file or what is displayed if the object is used in a video element. A single MediaStream object can be attached to multiple different outputs at the same time.
src:https://github.com/micklinISgood/amLive/blob/master/www/js/main.js#L12



2.arraybuffer via websocket:

This is the critical part of uploading the user's stream to server. Websocket provides a keep-alive connection to upload the streaming data, which saves the redundant tcp connections comparing to HTTP post. After trial and error, we found that setting the socket channel as binaryType = 'arraybuffer' gets the best performance. In addition, because it's data buffer, we can save the data with assigning a file extension directly, no need to do additional data parsing.

The ArrayBuffer object is used to represent a generic, fixed-length raw binary data buffer. You cannot directly manipulate the contents of an ArrayBuffer; instead, you create one of the typed array objects or a DataView object which represents the buffer in a specific format, and use that to read and write the contents of the buffer.
Src: https://github.com/micklinISgood/amLive/blob/master/www/js/main.js#L54-L63

3.Dynamic playback url:

We initially used jade as our fronted template cooperating with Node.js, but then we found ejs has a clearer syntax which is beneficial to our frontend development.
Src: http://www.embeddedjs.com/

4.MediaSource API:

MediaSource.SourceBuffer is the critical component for appending real-time video data. For every opened client connection, we will try to open the MediaSource.SourceBuffer and then append the live streaming data to it. By utilizing this api, we can play dynamic data through a HTML5 <video> tag. Before using this api, we tried to replace the src of the video tag periodically and failed. The simply setting of src seems can only handle static and full file playback. But if we want to splice a video in different sections of video from multiple sources, it doesn't support. Here is why MediaSource api comes into play.
src: https://github.com/micklinISgood/amLive/blob/master/www/js/audience.js#L88-L92

The mode property of the SourceBuffer interface controls whether media segments can be appended to the SourceBuffer in any order, or in a strict sequence.
The two available values are:
segments: The media segment timestamps determine the order in which the segments are played. The segments can be appended to the SourceBuffer in any order.
sequence: The order in which the segments are appended to the SourceBuffer determines the order in which they are played. Segment timestamps are generated automatically for the segments that observe this order.
Src: https://developer.mozilla.org/en-US/docs/Web/API/SourceBuffer/mode

5.xhr get for meta.webm for client:

The first portion of a webm file is the "initialization chunk". It contains the container's header information and should always be the first segment added. Since we generated a unique id for each live streaming, the client will try to get the meta.webm while sourceBuffer is open.
Server
src:https://github.com/micklinISgood/amLive/blob/master/server/video-processor.js#L21-L25
Client src:https://github.com/micklinISgood/amLive/blob/master/www/js/audience.js#L23-L30

6.In-memory broadcasting:

This is the best websocket practice of MediaSource.SourceBuffer. In the beginning, we make a chat room for each live stream and broadcast every uploaded chunk's url to clients. When a client received a specific json key, it will fetch the live chunk by using xhr get. This method works but costly. For every xhr get, the client needs to initialize a tcp connection to server first and server needs to do disk reads to return chunk. However, by using the trick of setting websocket

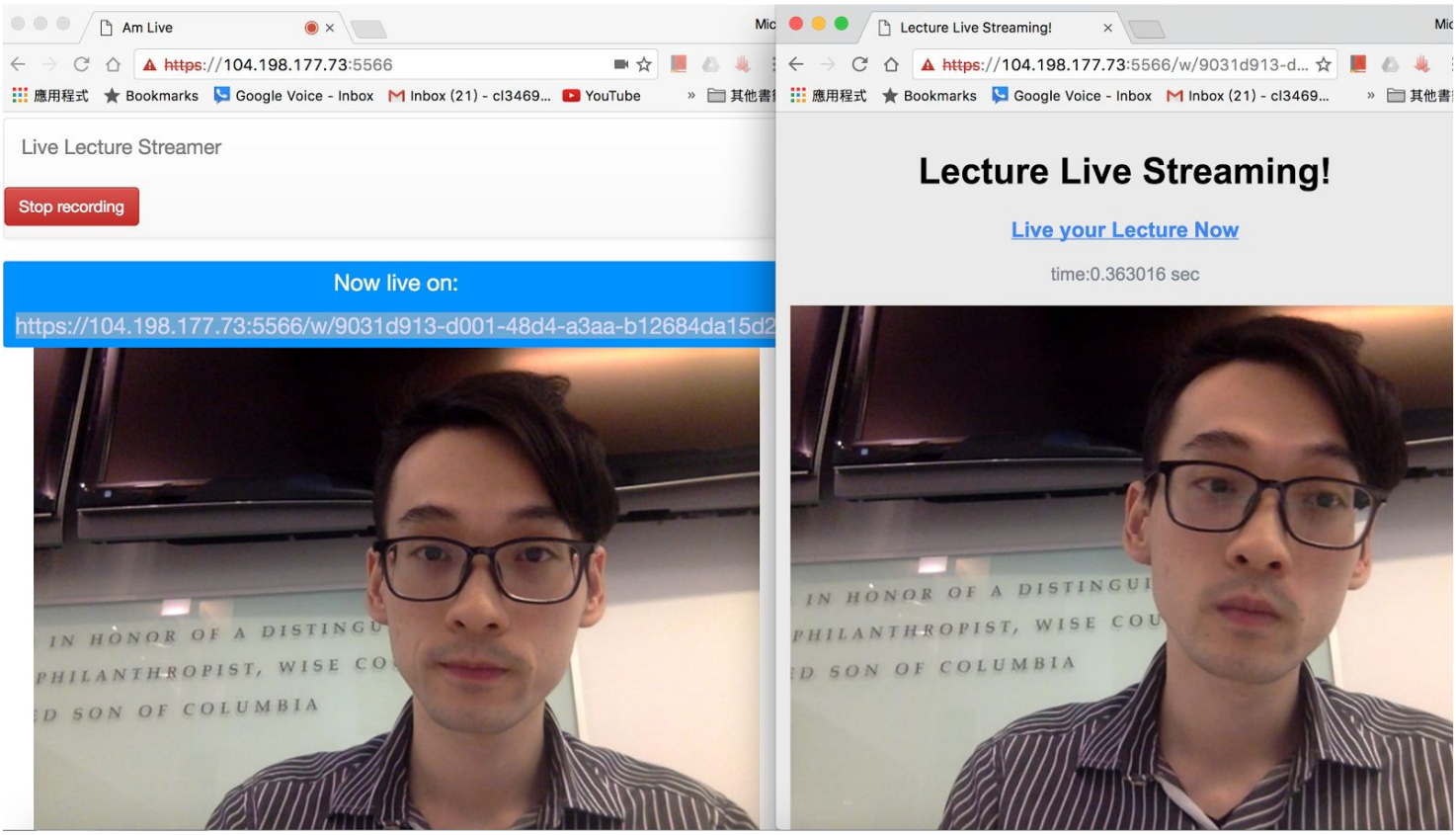
biniaryType after it opened, the client can successfully fetch the live chunk through websocket connection. In addition, since the live chunk can be broadcasted after server received, server doesn't need to save the live chunk. Finally, all the broadcasting is in-memory, which is faster than reading data from disk then returning to xhr get.
Server
src:https://github.com/micklinISgood/amLive/blob/master/server/video-processor.js#L45-L60
Client src:https://github.com/micklinISgood/amLive/blob/master/www/js/audience.js#L93

7.Clean up after live:

After a user has done a live, we still keep his video on server. In the beginning, we just broadcasted a signal to clients that the live is off and clients will reset the video src to lived video's url. However, the websocket connection of a client is still open because we didn't close those connections on the server side. And this causes the connection leak on the server as long as the client stays on the web page. We noticed this problem because we set the binarytype of the websocket and cannot use json to notify the clients. In the end, the server just simply close the subscription connections to show the live was done. And hence, no more connection leaks.
Server
src:https://github.com/micklinISgood/amLive/blob/master/server/video-processor.js#L145-L149
Client src:https://github.com/micklinISgood/amLive/blob/master/www/js/audience.js#L102-L114

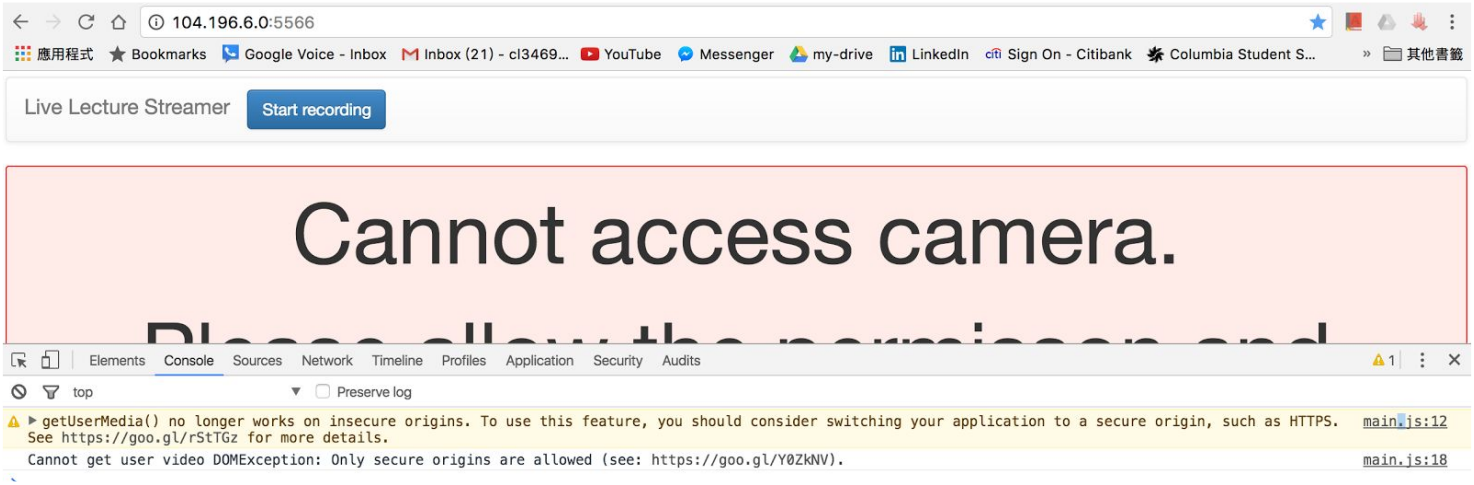
Website Snapshot:

Demo url: https://104.198.177.73:5566/
Video url: https://www.youtube.com/watch?v=w9pZ_rcmSvg



Lessons

1.HTTPS: getUserMedia() no longer works on insecure origin



While deploying our work on Google cloud, we faced that getUserMedia() no longer works on insecure origin. Google WebRTC API regards user stream as the data needs to be protected. For testing purpose, we resolved this issue by using a self-signed credential for HTTPS. Then now, all the stream data is safe, which will not be attacked by the sniffing. On the other side, the encoding may affect the performance.
src:https://github.com/micklinISgood/amLive/blob/master/server/index.js#L34-L35

2.Dynamic streaming:

A remote test between brooklyn area and columbia University revealed that the client will still try to download the high-resolution data from server even its bandwidth is low. In future work, we may enhance this feature with MPEG-DASH(mpd), which is an adaptive bitrate streaming technique that enables streaming of media content over the Internet delivered from conventional HTTP web servers. The main idea is simple. Server side will compress streaming data into different bitrate and put them into corresponding folder. Then, the client side will fetch the bitrate streaming data according to its bandwidth.

3.Video partitioning:

We initially face a critical problem which is if the video streamed from the server does not start from the first partition segmented in server side, the client can not playback the video successfully. We first considered it as a video codec constraint and started to implement a mechanism which guarantees the video segments will be transferred in sequence and start from the first segment. This was actually a work-around and we knew it’s performance is going to be bad since this mechanism always require streaming the whole video no matter at which point the client join in the live stream. To fix the problem completely, we dived into the details of Media Source API and figured out that there are actually two modes to reconstruct the video buffer from small segments, Sequence mode and Segment mode. Turned out that the Segment mode references the timestamp of each segment and try to combine them into a large video. So if the segments’ timestamp does not match the previous one, which is the case we have when client joins in the middle of the live stream, the video reconstruction process fails. To completely fix the problem, we change our reconstruction process by using Sequence mode and get the ideal result we wanted.

Teamwork

1.Mick Lin(cl3469):

- WebRTC getUsermedia & uploading to server via websocket
- Chat room mechanism between lecturer and audience
- Live stream data in-memory broadcasting
- Meta.webm setup on server side and xhr fetching on client side
- Self-signed Https setup on server side
- Main page UI: record button, alert box, navigation bar

2.Chih-Sheng Wang(cw2952):

- Dynamic playback url for different live session
- Debugging mediaSource
- Study MediaSource video partition mechanism : Sequence mode and Segment mode
- Frontend template testing with ejs and jade
- Frontend implementation