UNIVERSITÉ DE STRASBOURG

Master 2
Réseaux Informatiques et Systèmes Embarqués

Presented by

Boris Grozev
boris@jitsi.org

# Final Report

Media recording for multiparty video conferences based on WebRTC

Supervised by:

Emil Ivov
emcho@jitsi.org

Hosting enterprise:

BlueJimp

blue jimp®

## Contents

**Abstract**

This document is submitted as part of the requirements for obtaining a masters degree in informatics at the University of Strasbourg. It describes work done during a 6-months long internship. A modern video conferencing application based on WebRTC (*Jitsi Meet*) is discussed, and a system for media recording, which I helped to design and implement, is examined in detail. The first section serves as an introduction to the work environment for the internship and the most important relevant standards and technologies. The rest of the sections examine specific parts of the media recording system.

## 1 Introduction

The first part of this section is about *BlueJimp*, the company which hosts my internship. Sections 1.2 to 1.5 are an overview of the important standards and software products involved in a *Jitsi Meet* conference. Finally, section 1.6 defines what we mean by ”media recording” and gives a general overview of the process.

### 1.1 *BlueJimp*

XXX expand *BlueJimp* section

XXX add dates?

*BlueJimp*[1] is a small company which offers support and development services mainly focused around the *Jitsi*[2] family of projects. The FLOSS (Free/Libre Open Source Software) nature of these projects makes for a slightly unusual business model. The company works with various kinds of customers who all have different use cases for *Jitsi* and need it adapted to their needs. While *BlueJimp* has no exclusivity on such adaptations, it is tightly involved in the development of the project and some of the related technologies and standards. This has helped the company acquire significant credibility and offer advantageous price/quality ratios.

In addition to orders from customers, *BlueJimp* also often works on internal projects that aim to enrich *Jitsi* and make it more attractive to both users and enterprises.

*BlueJimp* is registered in Strasbourg, but the development team is international, with people working from different geographic locations. Most communication happens over the Internet using e-mail, instant messaging and audio/video calls.

My position in *BlueJimp* is that of a software developer. Apart from development, my tasks also involve a fair amount of research, experimentation and optimizations. I have worked on *Jitsi* previously and when my internship began, I was able to quickly get accustomed to the environment.

### 1.2 The *Jitsi* family

*Jitsi* is a feature-rich internet communications client. It is free software, licensed under the LGPL[**?**]. The project was started by Emil Ivov in the University of Strasbourg in 2003, and it was then known as SIP Communicator. In the beginning SIP Communicator was only a SIP client, but through the years it has evolved into a multi-protocol client (XMPP, AIM, Yahoo! and ICQ are now also supported) with a very wide variety of features: instant messaging (IM), video calls, desktop streaming, conferencing (multi-party, both audio and video), cross-protocol calls (SIP to XMPP), configuration provisioning, media encryption (with SRTP, setup via SDES, ZRTP or DTLS/SRTP), many audio and video codecs, echo cancellation, session establishment using ICE, chat sessions encrypted using OTR, file transfers. Most of the development comes from *BlueJimp*.

XXX too much features listed? XXX clean the above paragraph

*Jitsi* is written mostly in Java and runs on a variety of platforms (Windows, Linux, MacOSX and Android). The various projects comprise a massive codebase– over 700 000 lines of Java code alone.

A big part of the code which was originally in *Jitsi* is now split into a separate library– *libjitsi*. This allows it to be easily reused in other projects, such as *Jitsi-Videobridge*. The code in *libjitsi* deals mainly with multimedia– capture and rendering, transcoding, encryption/decryption and

[1] https://bluejimp.com
[2] https://jitsi.org

transport over the network of audio and video date. It contains the RTP stack for *Jitsi* (partially implemented in *libjitsi* itself, partially delegated to FMJ).

*Jitsi-Videobridge* is a server-side application which acts as a media relay and/or mixer. It allows a smart client to organize a conference using an existing technology (for example, SIP or XMPP/Jingle), outsourcing the bandwidth-intensive task of media relaying to a server. The organizing client controls *Jitsi-Videobridge* over XMPP[3], while the rest of the participating clients need not be aware that *Jitsi-Videobridge* is in use (for example they can be simple SIP clients). Since the end of 2013 *Jitsi-Videobridge* supports ICE and is WebRTC-compatible.

One of the latest additions to the *Jitsi* family is *Jitsi Meet*[4]. This is a WebRTC application, which runs completely within a browser and creates a video conference using a *Jitsi-Videobridge* instance to relay the media.

## 1.3 WebRTC

WebRTC (Web Real-Time Communications) is a set of specifications currently in development, that allow browsers which implement them to open ”peer connections” to other browsers. These are direct connections between two browsers (a webserver is used only to setup the connection), and can be used to send audio, video or application data. The specifications are open and are meant to be implemented in the browsers themselves (as opposed to in additional plug-ins).

WebRTC is divided in two main parts: the JavaScript standard APIs, being defined within the *WebRTC* working group[?] at W3C, and the on-the-wire protocols, being defined within the *RTCWEB* working group[?] at the IETF.

If implemented, these standards provide web developers with a very powerful tool, which can be used to create rich real-time multimedia applications very easily. They also allow for some very interesting and more complicated use-cases.

The specifications are still being developed, but are already at an advanced stage. There is an open-source implementation of the network protocols, provided by Google with a BSD-like license. I will refer to this implementation as *webrtc.org* (which is the domain name of the project). Currently all browsers implementing WebRTC (Chrome/Chromium and Mozilla Firefox) use *webrtc.org* as their base. Because of this, *webrtc.org* is very important – it is used for all practical compatibility testing, making it the de-facto reference implementation.

## 1.4 XMPP and Jingle

Extensible Messaging and Presence Protocol (XMPP)[?] is a mature, XML-based protocol which allows endpoints to exchange messages in near real-time. The core XMPP protocol only covers instant messaging (that is the exchange of text messages meant to be read by humans), but there are a variety of extensions that allow the protocol to cover a wide range of use cases. Many such extensions are published as XMPP Extension Protocols (XEPs), and there are XEPs which allow for group chats, user avatars, file transfers, account registration, transporting XMPP over HTTP, discovering node capabilities, management of wireless sensors (provisioning, control, data collection), and, most relevant here, internet telephony.

*Jingle*[?, ?] is a signalling protocol, serving a purpose similar to that of SIP: it uses an offer-answer model to setup an RTP session. Many of the protocols often used with SIP, such as ICE[?], ZRTP, DTLS-SRTP, and RFC4575 can also be used with *Jingle*. Mappings have been defined between the two[?], which allow gateways or rich clients to organize cross-protocol calls.

COnferencing with LIghtweight BRIdging (COLIBRI[?]) is an extension developed mostly in *BlueJimp* for use with *Jitsi-Videobridge*. It provides a way for a client to control a multimedia relay or mixer, such as *Jitsi-Videobridge*. It works with the concept of a *conference*, which contains *channels*, separated in different *contents* (see fig.1). A client requests the creation

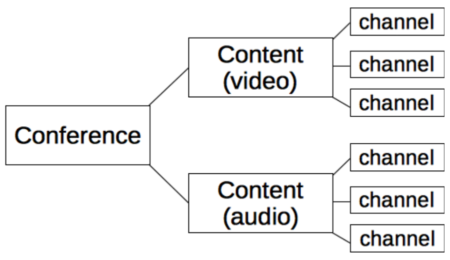


Figure 1: A COLIBRI conference.

[3]Although a REST API is also available.

[4]https://jitsi.org/Projects/Jitsi-Meet

of a *conference* with a specified number of channels. The mixer allocates local sockets for each *channel* and provides their addresses to the client. The client can then use these transport addresses as its own to establish, for example, a *Jingle* call. Instead of just allocating local sockets, the ICE protocol can be used, in which case the mixer provides a list of ICE candidates for each *channel*. The protocol works with a natural XML representation of a *conference*. After the *conference* is established, the client can add or remove channels from it, or change the parameters (such as the direction in which media is allowed to flow) of an existing *channel*.

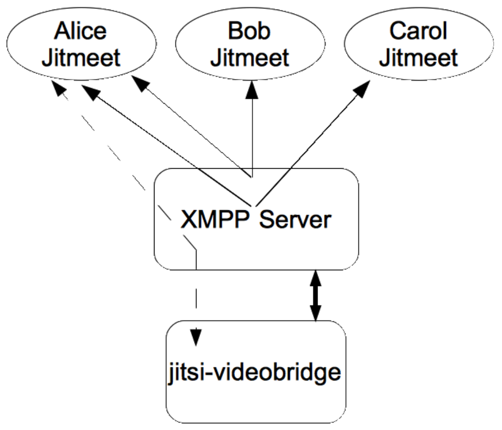
## 1.5 A *Jitsi Meet* conference

*Jitsi Meet* uses the above-mentioned technologies to create a multi-party video conference. The endpoints of the conference are simply WebRTC-enabled browsers[5] running the actual *Jitsi Meet* application. They all connect to an XMPP server and join a Multi-User Chat (MUC) chatroom. One of the participants (the first one to enter the chatroom) assumes the role of organizer (or focus).

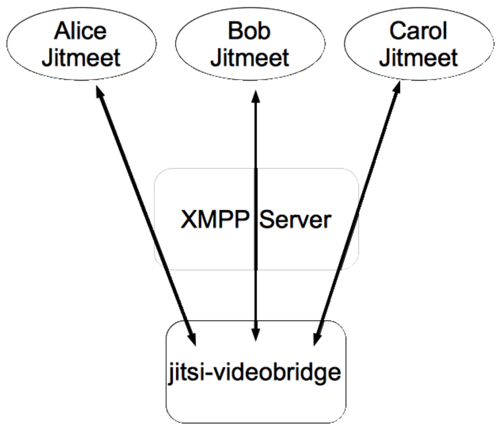
The focus creates a COLIBRI conference on a *Jitsi-Videobridge* instance *jvb*, and allocates two COLIBRI channels for each participant (one for audio, and one for video). Then, it initiates a separate *Jingle* session with each participant, using the transport information (i. e. the list of ICE candidates) obtained from *jvb* instead of its own. When the participants accept the Jingle sessions, they in effect perform ICE and establish direct RTP sessions with *jvb*.

The resulting connections are depicted in Figures 2a and 2b.

XXX update figures



(a) Signalling connections in a *Jitsi Meet* conference. The solid lines are XMPP/Jingle sessions, the dashed line is XMPP/COLIBRI. The thick line is an XMPP Component Connection.



(b) Media connections in a *Jitsi Meet* conference. The lines represent RTP/RTCP sessions.

On the user-interface end, *Jitsi Meet* aims to make it as easy as possible for a person to enter or organize a conference. Entering a conference is accomplished by simply opening a URL such as *https://meet.jit.si/ConferenceID* (where *ConferenceID* can be chosen by the user). If *ConferenceID* doesn't exists, it is automatically created and the user assumes the role of focus, inviting anyone who enters later on. If *ConferenceID* exists, the user joins it (possibly after entering a password for the conference).

When in a conference, the interface has two main elements: one big video (taking all available space) and, overlayed on top of it, scaled down versions of the videos of all other participants. Figure **??** is a screenshot from the actual application. The video shown in full changes, according to who is currently speaking.

XXX able to install your own *Jitsi Meet*

[5]Although at the present time only Chrome/Chromium and Opera are supported.

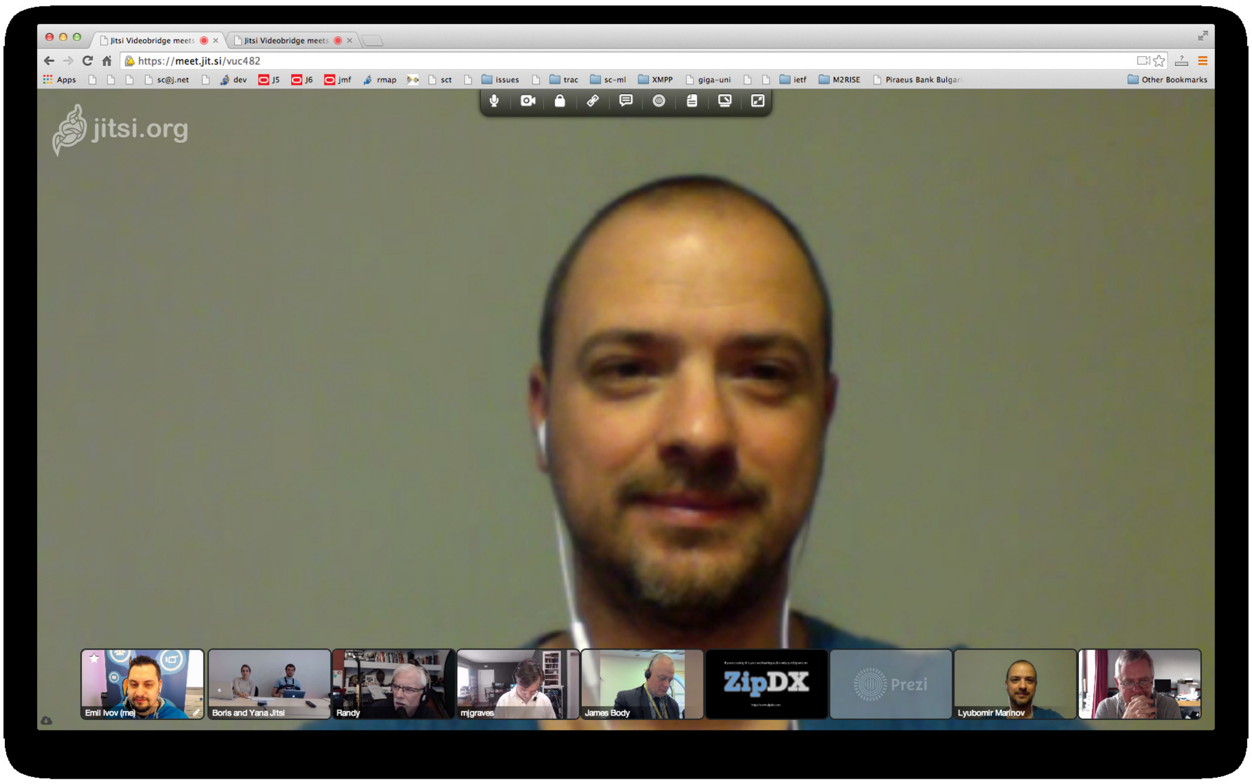


Figure 3: A screen capture from a *Jitsi Meet* conference.

## 1.6 Recording

By recording a multimedia conference in general we mean the following: all the audio and video involved in the conference is saved to disk in some format, in a way which allows the whole conference to be played back later on.

In our specific case, the recording of a conference has three main parts:

- Recording media
- Recording metadata
- Post-processing

Recording of the media is the process in which the audio and video RTP streams in the conference are converted to a convenient format and saved to disk. There are many different ways in which this can be done. Our final solution (and some of the ideas that didn't work) are discussed in detail in sections 3 and 4.

Recording of metadata means saving additional information (apart from the media itself) which is necessary to play back the conference later. This includes timing information, participant names and changes of the active speaker. There is a detailed discussion of the metadata that we use in section 5.

Post-processing in our case means taking all the recorded data and producing a single file with one audio track and one video track. The specifics depend on configuration, but generally all audio is mixed together, and the videos are combined in a way to resemble the *Jitsi Meet* interface. Appendix A discusses the post-processing application.

**Where does recording happen?** As can be seen in figure 2b, in a *Jitsi Meet* conference both *Jitsi-Videobridge* and all the participants have access to the RTP streams, and so could potentially perform recording.

Since the participating clients are running an application within their browser, if we want one of them to do recording, we would need modifications to the browsers (which is undesirable). To avoid this, a "fake participant" can be added to the conference, which does not not actually participate in the conference (does not send audio or video), and does not run in a browser.

Recording directly on *Jitsi-Videobridge* is more straightforward, so our initial implementation was focused on that. However, most of the code resides in *libjitsi*, allowing it to be easily reused.

Li Shunyang, a student from Peking University, is currently working, under the Google Summer of Code program, on a recording application[6] which uses the "fake participant" approach.

## 2 The WebRTC transport layer

XXX This section is about transporting media in webrtc. Uses ICE to establish a UDP (or TCP) connection, then makes a DTLS/SRTP (in the case of media) stream.

XXX Describe the libjitsi code structure (PacketTransformers)

### 2.1 RED

### 2.2 ULPFEC

ULPFEC (stupid idea with re-writing sequence numbers. why adding FEC arbitrary to the stream makes it impossible to decode correctly. what webrtc does. how we handle it by just removing my stupid code). Diagrams diagrams

The peer connections between browsers use the Real-time Transport Protocol (RTP[**?**]), and WebRTC mandates the use of Interactive Connectivity Establishment (ICE[**?**]) for session establishment.

WebRTC defines two mandatory to implement audio codecs: G.711[**?**] and Opus[**?**, **?**]. This guarantees that any two compliant browsers can establish an audio session. For video, however, the situation is not so simple, with no decision having yet been made on a mandatory to implement codec. Google's VP8[**?**] codec is the one implemented in *webrtc.org* and is therefore the de-facto standard at the moment.

### 2.3 Retransmissions

## 3 Recording video

The WebRTC standards do not define a video codec which has to be supported by all clients. There has been a very long discussion at the IETF (XXX add link?) about whether to make a codec mandatory to implement (MTI), and if so, which one. The four main options suggested were *(i) make the VP8 codec MTI*; *(ii) make the H264 codec MTI*; *(iii) have no MTI codecs*; *(iv) make both VP8 and H264 MTI.* No consensus has been reached. Nevertheless, currently VP8 is the de-facto standard codec for WebRTC, because it is the only codec supported by *webrtc.org*(and therefore *Jitsi Meet*).

VP8 is a video compression format, defined in RFC6386[**?**]. It was originally developed by *On2 Technologies*, which was acquired by *Google* in 2010. *Google* published the specification and released a reference implementation (*libvpx*) under a BSD-like opensource license. They also provided a statement granting permission for royalty-free use of any of their patents used in *libvpx*[7].

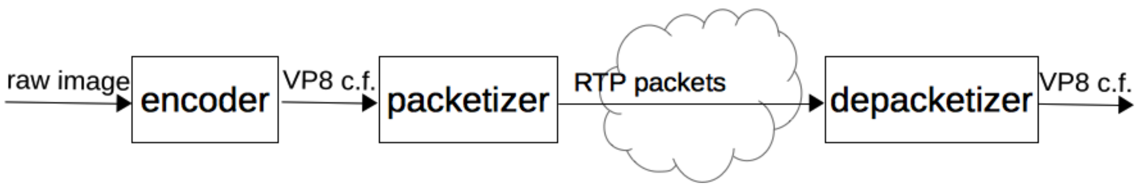
Both VP8 in general and *libvpx* work exclusively with the I420 raw (uncompressed) image format[8]. There is an encoder, which takes as input an I420 image and produces a "VP8 Compressed Frame". Similarly, a decoder reads VP8 compressed frames and produces I420 images.

A separate specification[**?**] defines how to transport a VP8 compressed frame over RTP. In short, the process involves optionally splitting a VP8 compressed in parts, adding a structure known as a VP8 Payload Decriptor to each part, and then encapsulating each part in RTP. This process is reffered to as packetization, and the reverse process (of collecting RTP packets and constructing VP8 compressed frames)– depacketization. Figure **??** provides a high-level overview of the use of VP8 with RTP.

[6]https://github.com/jitsi/jirecon

[7]http://www.webmproject.org/license/additional/

[8]http://www.fourcc.org/yuv.php#IYUV



*libjitsi* already has a VP8 implementation, which is used in the *Jitsi*client and consists of four parts: an encoder and decoder (wrappers around *libvpx*), a packetizer and a depacketizer. For the purposes of recording, we need only a depacketizer. We found that the existing depacketizer is not compliant with the specification, and also not compatible with *webrtc.org*. We decided to re-write it from scratch.

**The depacketizer** The VP8 Payload Descriptor contains, among others, the following fields.

- *S*-bit: start of VP8 partition, set only if the first byte of the payload of the packet is the first byte of a VP8 partition.
- *PID*: Partition ID, specifies the ID of the VP8 parition to which the first byte of the payload of the packet belongs.
- *PictureID*: A running index of frames, incremented by 1 for each subsequent VP8 frame.

Apart from this, the depacketizer also uses the following fields from the RTP header:

- *Timestamp*: a 32-bit field specifying a generation timestamp of the payload. For VP8, all RTP packets from a given frame have the same timestamp.
- *Sequence number*: An index of RTP packets.
- *M*-bit: set for the last RTP packet of a frame (and only for it).

We implemented the algorithm suggested in the specification: we buffer RTP packets locally, until we receive a packet from a new frame. At this point, we check whether the buffer contains a full VP8 frame, and if it does we output it. Otherwise, we drop the buffer and start to collect packets for the next frame.

In order to decide whether a received packet is from a new frame or not, the RTP timestamp and the *PictureID* fields are used (if either don't match, then it's a new frame).

The *PID* and *S* fields from the VP8 Payload Descriptor allow us to detect the first packet from a frame – the first, and only the first packet will have both the *S*-bit set and *PID* set to 0.

This allows us to easily check whether we have a full packet in the buffer or not: we have a full packet if: *(i) we have the beginning of a frame*; *(ii) we have the end of a frame (a packet with the M-bit set)*; and *(iii) we have all RTP sequence numbers inbetween.*

The following pseudo-code illustrates the procedure.

```
receive(Packet p){
    if (!belongsToBuffer(p))
        flush();
    push(p);

    if (haveFullFrame())
        outputFrame();
}

belongsToBuffer(p){
    if (bufferEmpty())
        return true;
    else if (bufferRtpTimestamp == p.RtpTimestamp
            && bufferPictureID == p.PictureID)
        return true;
    return false;
}

haveFullFrame(){
    if (! (buffer.first.S && buffer.first.PID == 0))
        return false;
    if (!buffer.last.M)
        return false;
    for (int i=buffer.first.seq; i<=buffer.last.seq; i++)
        if (!buffer.contains(i))
            return false;
    return true;
}
```

**Jitter buffer?**

**Container format** After depacketization, we are left with a stream of VP8 Compressed Frames. We needed to decide how to store them on disk. We considered three options:

- Use the *ivf* container format
- Define and use our own container format
- Use the *webm* container format

The *ivf* format is a very simple video-only, VP8-only storage format. It was developed with *libvpx* for the purposes of testing the implementation. It precedes each frame with a fixed-size header containing just the length of the frames and a presentation timestamp. The only advantage of using this format is the relative simplicity of it's implementation. The disadvantages include that it is not supported by many players (for example browsers support it), and that it's simplicity might be limiting our future needs.

Defining our own container format has one advantage, and that's the possibility to design it in a way that allows partial VP8 frames. The *libvpx* decoder has a mode which allows it to decode a frame even if parts of it are missing. In order to use this API, however, the decoder needs to be provided with information about which parts (which VP8 partitions) are missing, and this information would be lost if we use *ivf* or *webm*. The disadvantages of this approach are the complexity that it brings, and the inflexibility with regards to the players – we'd need to implement our own decoder (probably to be used in post-processing).

The *webm*[9] format is a subset of *matroska*[10]. It has been designed specifically to contain VP8 (possibly interleaved with audio) and to be played in browsers. It allows for much more flexibility than *ivf*. The main advantage is that it can be played by many players.

After some research, we found a library that would allow us to write *webm* files very easily, so we decided to ignore *ivf*, postpone the potential definition of a new format, and use *webm*.

XXX timestamps

XXX diagraam: transformers -¿ jitter buffer -¿ depacketizer -¿ webmdatasink

**Requesting keyframes** VP8 has two types of frames: I-frames (or keyframes) which can be decoded without any other context, and P-frames, whose decoding depends on previously decoded frames. Therefore a VP8 decoder needs a keyframe before it can decode frames. For this reason we want to always start the recorded files with a keyframe.

XXX Note: no decoding/encoding.

[9]http://www.webmproject.org/about/
[10]http://www.matroska.org/

## 4 Recording audio

We tried mixing streams live – difficult to sync

Decided to record them separately

Still problems with sync: lost packets cause ”holes” and in a long recording video starts to progressively lag more and more. Needed to add silence: allows for perfect sync (up to a sample)

Problem with too much silence (e.g when someone unmutes). Restarting streams when too much silence detected

We reencode to mp3 (because we have mp3 support ready). We want to do ogg/opus, and avoid decoding and encoding altogether. Problem: FEC

diagram: transformers -¿ jb -¿ decoder -¿ silence -¿ encoder + datasink

FEC (opus): how it works. when we reencode audio, we take advantage of it automatically (already implemented in the decoder).

diagram: how would it look with ogg/opus

## 5 Recording metadata

What is the metadata used for. Sample JSON

Important information: filenames and timestamps

speaker changed events

## 6 Synchronization

Purpose: audio and video from a single participant need to be perfectly synchronized. Need to use the timestamps from the source, because of network jitter.

How it works: RTCP sender reports and the info they contain.

My lovely Synchronizer class. Diagrams (I have some on paper, also sync.pdf somewhere)

## 7 Dominant speaker identification

Dominant speaker identification (DSI) refers to the process of continuously analysing a set of audio streams (which are assumed to contain human voice) and keeping track of the one which comes from the person currently speaking (or the person currently speaking ”the most”). Obviously the problem does not have a single solution, and so it is hard to measure objectively how well a system performs. A good description of the general problem and a proposed solution is available in [**?**].

In a *Jitsi Meet* conference there are two use-cases for DSI: for use ”live” in a conference and for recording. For ”live” use, the purpose is to have the client interface change during a conference, according to who is the dominant speaker (i.e. the video shown in full changes when the dominant speaker changes). In this case DSI is performed on *Jitsi-Videobridge*, which uses WebRTC data channels to notify the clients of the change.

For use in recording, the purpose is for the post-processing application to take into account the dominant speaker when combining the videos (in general, it follows the *Jitsi Meet* interface and renders the dominant speaker in full size). In this case, DSI is performed on the recording application, and changes of the dominant speaker are saved as metadata.

**Implementation** My task was to design and implement an API and a simple, non-optimized algorithm , which we could later improve upon.

I came up with a very simplified scheme, implemented it and tweaked its parameters until it seemed to work well in a conversation with two people. It only takes into account the audio levels (that is, the loudness of the sound) of the different streams. It works like this:

At all times one stream is considered active, while the rest are considered ’competing’. All streams have an associated score. When new audio levels are measured for a stream its score is recomputed, and the scores of all streams are examined in order to determine if one of the competing streams should replace the then active stream.

In order to be ’eligible’ to replace the active stream, a competing stream has to:

1. Have a score at least $C$ times as much as the score of the currently active stream.
2. Have score at least MIN_SCORE.
3. Have been active in the last MIN_ACTIVE milliseconds.

The first rule helps to avoid often changing the active when there are two streams with similar levels. The second prevents the active speaker from being changed during a pause of his speach, while no one else is speaking either (but someone else generates higher levels during "silence" than the speaker). The third is to make sure that when participants leave a conference, they aren't mistakenly chosen as active (this is due to implementation details)

The values for the parameters are very dependent on the exact rules for scoring. I chose the values based on a few uncontrolled experiments we performed. I used MIN_ACTIVE = 1000 and $C = 1.15$ (MIN_SCORE is too arbitrary to deserve mention).

Scores are computed as follows: $C_{recent} * avg(0, N_1) + C_{older} * avg(N_1, N_2)$ where $avg(X_1, X_2)$ is the average audio level for the interval $[now - X_2, now - X_1)$ (in milliseconds).

Currently the values of the parameters are: $C_{recent} = 2, C_{older} = 1, N_1 = 250, N_2 = 1250$.

We found that this solution works sufficiently well, at least for the purposes of testing recording and post-processing. One problem that we face is when one of the participants is generating constant noise (caused for example by a spinning fan near the microphone). In this case the algorithm tends to select this participant, even if they are not speaking. One suggestion for a fix is to measure the variance of the sound levels for a participant, and penalize the score if the variance is low. This depends on the assumption that a person speaking would produce sound with varying loudness (and that this variation can be detected with the 20ms resolution that we use). However, we have not yet tried to solve this problem.

Lyubomir Marinov has now taken over this task, and has implemented an improved algorithm that more closely resembles the one proposed in [**?**]. However, his implementation still relies on "audio levels" and not on other analysis of the audio samples. This offers a significant advantage in our use case, because this information (the "audio level" for a particular RTP packet) is already calculated by the sending client, and is included in the RTP packet in the format specified in RFC6464[**?**]. This allows *Jitsi-Videobridge* to do DSI very cheaply, without the need to decode the received audio streams.

## 8 future work

RTX, recording in ogg/opus, postprocessing, jitter buffers?

XXX TODO: support for RTCPCompoundPackets: does it fitsomewhere above?

XXX might scratch "future work" altogether?

## 9 Conclusion

XXX Crap, I almost forgot about this!

## A Post-processing

Jipopro