UNIVERSITÉ DE STRASBOURG

Master 2
Réseaux Informatiques et Systèmes Embarqués

Submitted by

Boris Grozev
boris@jitsi.org

Strasbourg, June 12, 2014

Final Report

Media recording for multiparty video conferences based on WebRTC

Supervisor

Dr. 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[2]. 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), media encryption (SRTP), session establishment using ICE, file transfers and more. Most of the development is financed by *BlueJimp*.

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

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

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". 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 (without the need for additional plug-ins).

WebRTC is divided in two main parts: the JavaScript standard APIs, being defined within the *WebRTC* working group[12] at W3C, and the on-the-wire protocols, being defined within the *RTCWEB* working group[4] at the IETF.

These standards provide web developers with a very powerful tool, which can be used to easily create rich real-time multimedia applications. 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, Opera 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)[7] 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 for group chats, user avatars, file transfers, account registration, transporting XMPP over HTTP, discovery of node capabilities, management of wireless sensors (provisioning, control, data collection), and, most relevant here, internet telephony.

Jingle (define in XEP-0166[13] and XEP-0167[14]) 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[6], ZRTP[**?**], DTLS-SRTP[**?**], and RFC4575[**?**] can also be used with *Jingle*. Mappings have been defined between the two[3], which allow gateways or rich clients to organize cross-protocol calls.

COnferencing with LIghtweight BRIdging (COLIBRI[15]) 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). In the most common use case a client requests the creation of a *conference* with a specified number of channels. The mixer allocates local sockets for each *channel* and provides

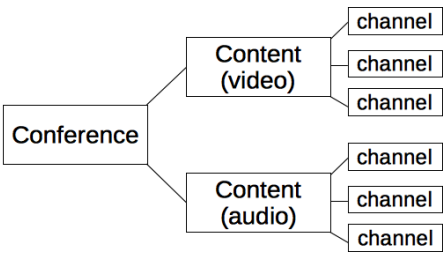


Figure 1: A COLIBRI conference.

[3]Although a REST API is also available.

[4]https://jitsi.org/Projects/JitsiMeet

their addresses to the client. The client then uses 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*.

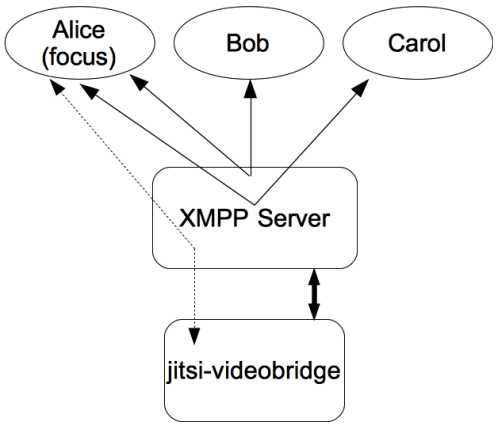
1.5 *Jitsi Meet*

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).

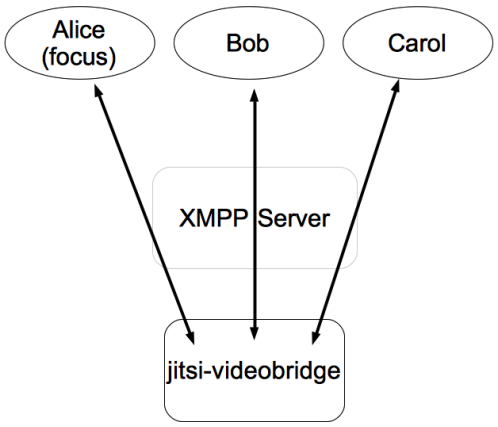
The focus creates a COLIBRI conference on a *Jitsi-Videobridge* instance (*jvb*), and allocates two COLIBRI channels for each participant (one for audio, and one for video). Then, it initiates a separate *Jingle* session with each participant, using the transport information (i. e. the list of ICE candidates) 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 for signalling and media are depicted in Figures 2a and 2b respectively.

XXX update pictures



(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. Current work is underway to use dominant speaker identification (see section 7) to change the video shown in full size to the person currently speaking.

In contrast to other products for video conferencing over the internet, the whole infrastructure needed to run *Jitsi Meet* can be installed in a custom environment. The only services which are needed are a web server (only serving static content), an XMPP server, and an instance of *Jitsi-Videobridge*. This makes *Jitsi Meet* very suitable for businesses (or even individuals) who want full

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

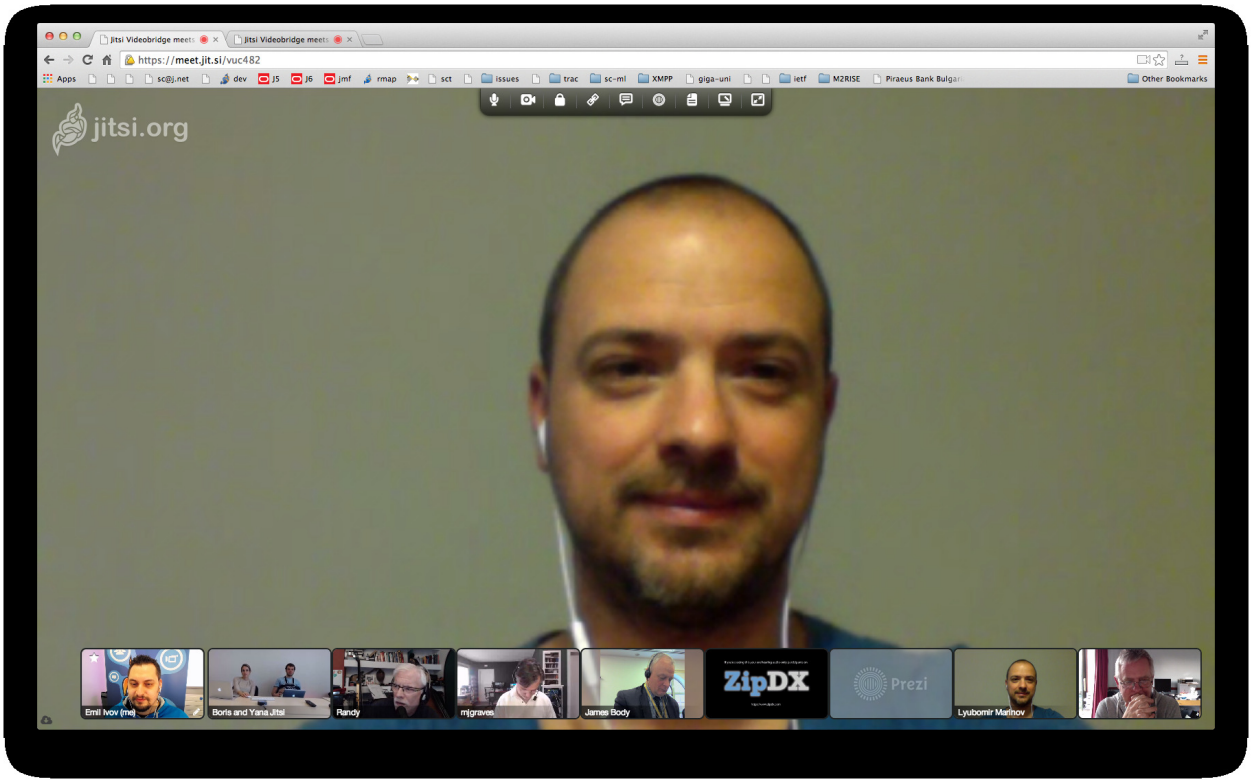


Figure 3: A screen capture from a *Jitsi Meet* conference.

control over their conferencing solution.

1.6 Recording

By recording a multimedia conference in general we mean the following: all the audio and video from 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 four main parts:

- Recording video
- Recording audio
- 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 filenames, 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, which does not not actually participate in the conference (does not send audio or video), and does not run in a browser. Still, it connects as a normal participant and establishes an RTP session with *Jitsi-Videobridge* (see figure 4).

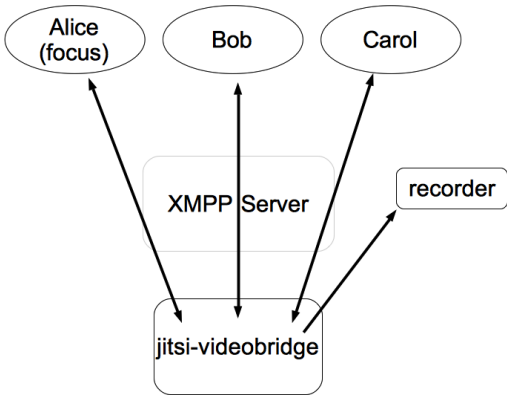


Figure 4: A recording application connected to a *Jitsi Meet* conference as a "fake" participant.

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 which uses the fake participant approach– *Jirecon*[6].

2 The WebRTC transport layer XXX rename (s/WebRTC//?)

The documents from the RTCWEB working group at the IETF specify how multimedia is to be transported between WebRTC endpoints. In the most part existing standards are reused.

It is mandatory to use the Interactive Connectivity Establishment (ICE[6]) protocol to establish a session. This assures that an endpoint will not send any media before it has receive consent (in the form of a STUN message) from the remote side. This protects against possible traffic augmentation attacks, in which a malicious web-server causes browsers to send large amounts of data (e.g. a video stream) to a target.

After a connection is established using ICE, a DTLS-SRTP session is started. This means that the endpoints use Datagram TLS (DTLS[**?**]) to exchange key material, which is then used to generate session keys for a Secure Real-Time Protocol (SRTP[**?**]) session. The procedure is defined in RFC5763[**?**]. In a *Jitsi Meet* conference, each participant's browser setups two SRTP sessions with *Jitsi-Videobridge* in this way.

SRTP provides an unreliable transport. For this reason *webrtc.org* uses a couple of mechanisms on top of SRTP to improve the quality of the media. These mechanisms and their implementation in *libjitsi* are discussed in sections 2.2 to 2.4. Before that section 2.1 gives an overview of the RTP stack used in *libjitsi*.

2.1 The RTP stack in *libjitsi*

libjitsi makes heavy use of the Freedom for Media in Java (FMJ[7]) library. This is an open-source implementation of the Java Media Framework (JMF) API, and it is used in *libjitsi* for a variety of tasks: capture and playback of media, conversion (transcoding) of media, and for handling of basic RTP streams. FMJ is highly extensible, and many components (such as media codecs, capture devices and renderers) are written in *libjitsi*.

The RTP stack which FMJ uses lacks some features: notably support for SRTP and for asymmetric payload type mappings (i.e. sending and receiving a given format with two different RTP payload type numbers). In order for *libjitsi* to implement these features, it intercepts the RTP packets from the actual socket in use, and processes them before passing them on to FMJ. Specifically, packets go through a chain of *PacketTransformer*s, which perform various tasks. Figure **??**

[6]https://github.com/jitsi/jirecon
[7]http://sourceforge.net/projects/fmj/

illustrates this scheme.

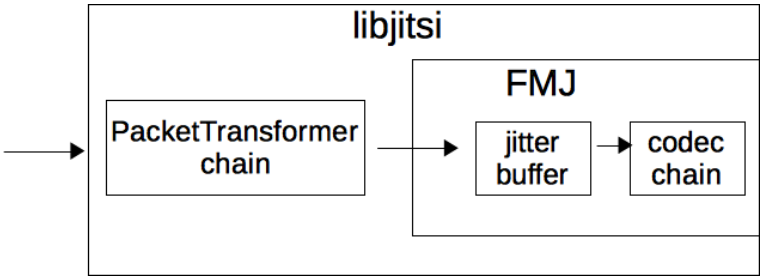


Figure 5: General scheme of the RTP stack in *libjitsi*.

*PacketTransformer*s provide a convenient interface to intercept RTP and RTCP packets at different stages of their processing and perform additional operations on them. Despite their name, *PacketTransformer*s don't need to change the packets in any way. That is, they can be used just for monitoring.

Figure 6 lists the currently used *PacketTransformer*s in *libjitsi*. A packet which arrives from the network goes through the chain downwards (and when packets are sent, they go through the same chain but in the other direction). The transformer labeled "RFC6464" is a filter, which allows packets marked as silence using RFC6464[**?**] to be dropped early in the process, thus saving processing time (this is an optimization implemented for the use case of *Jitsi-Videobridge*). The SRTP transformer decrypts SRTP packets. The "Override PT" transformer changes the payload type numbers of packets. It is used to implement asymmetric payload type mappings. The statistics transformer monitors RTCP packets and extracts statistics from them, making them available to other parts of the library.

The rest of the transformers (the ones in grey) were added with the implementation of the recording system, and will be discussed in the next sections.

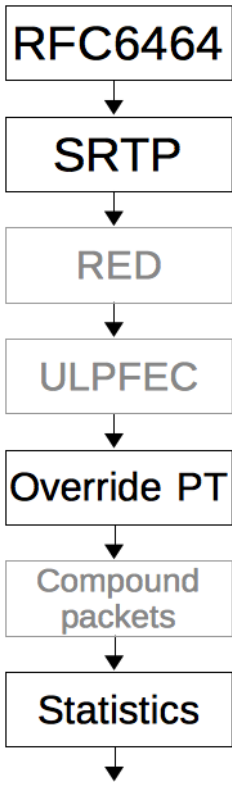


Figure 6: The chain of *PacketTransformer*s in *libjitsi*. The shaded elements are new additions added while implementing the recording functionality.

2.2 RED

RFC2198[**?**] defines an RTP payload-type that allows the encapsulation of one or more "virtual" RTP packets in a single RTP packet. It is intended to be used with redundancy data. Its use is negotiated as a regular media format and it does not have a static payload-type number, so a dynamic number is assigned during negotiation.

In *webrtc.org*, RED is supported and used for video streams. In the case of a *Jitsi Meet* conference, it is negotiated between the clients, and *Jitsi-Videobridge* has no way of affecting its use or its payload-type number, because it does not actively participate in the offer/answer procedure. This means that in order to record video, the recorder has to understand RED.

We decided that the best way to implement RED in *libjitsi* is as a *PacketTransformer*. There was one complication– *PacketTransformer*s work with single packets (they take a single packet as input and produce a single packet as output), while a RED packet may contain multiple "virtual" RTP packets which would need to be output.

We modified *libjitsi*, so that all *PacketTransformer*s work with multiple packets at a time– they take an array of packets as input and produce an array as output. This change was not easy, because we had to make sure that we didn't break existing code, but it proved useful later on when we added support for ULPFEC and RTCP compount packets.

We implemented a RED packet transformer following RFC2198 and inserted it in the transformer chain, after the SRTP transformer.

2.3 Uneven Level Protection Forward Error Correction

In general, Forward Error Correction (FEC) refers to a mechanism which allows lost data to be recovered without retransmission. It involves sending redundant data, in one way or another.

RFC5109[**?**] defines a specific RTP payload-type for redundant data called Uneven Level Protection FEC (ULPFEC). It is generic in the sense that that it can be used with any media payload-type (audio or video, no matter what the codec is).

In *webrtc.org*, ULPFEC is used with video[8], and while not strictly mandatory for our video-recording (as opposed to RED), it is important because by decreasing the number of irretrievably lost packets, it will improve the quality of the recordings.

ULPFEC (in the rest of the section we refer to it as simply FEC) is applied to an RTP stream (the "media stream", with "media packets") and adds additional packets to it ("FEC packets"). The basic idea is simple – take a set S of a few media packets and apply a parity operation (XOR) on it, resulting in a FEC packet f. If any one of the packets in S is lost, the receiver can use the rest of the packets in S together with f to reconstruct the lost packet[9].

Along with the parity data, a FEC packet contains two fields which are used to describe the set S from which it was constructed: a "sequence number base" field, and a bitmap field that describes the sequence numbers of the packets in S using the base. The packet f is said to "protect" the packets in S. This scheme allows FEC to work without any additional signalling (apart from the payload-type number negotiated during session initialization).

We decided to implement FEC as another *PacketTransformer*. This is how it works:

Two buffers of packets are kept: $bMedia$ and $bFEC$. With every FEC packet f is associated the number $numMissing(f)$ of media packets protected by f which have not been received.

On reception of a media packet, the values $numMissing(f)$ are recalculated for all f in the $bFEC$ buffer. Then, for all f in $bFEC$: if $(numMissing(f) == 0)$, then f is removed from $bFEC$. If $numMissing(f) > 1$, then do nothing. If $numMissing(f) == 1$, use f and $bMedia$ to reconstruct a media packet and remove f from $bFEC$.

On reception of a FEC packet f, $numMissing(f)$ is calculated, and the same as above is done according to its value.

$bFEC$ is limited to a small size and if a new FEC packet arrives while it is full, the oldest FEC packet is dropped. This prevents "stale" FEC packets (for which $numMissing$ will always be > 1, because more than one of their protected packets have been lost) to accumulate and cause needless computation.

RFC5109 does not place any restrictions on the placement of FEC packets within a stream, and in our architecture FEC packets are handled entirely in the FEC *PacketTransformer* and not passed on to the rest of the application. This presents a potential problem for the depacketizer (see section 3), because it cannot differentiate between a sequence number missing because a packet was lost and a sequence number missing because it was used by a FEC packet.

For this reason we initially implemented re-writing of the RTP sequence numbers of the media packets after they pass the *PacketTransformer*– we decreased their number by the number of FEC packets already received (see figure 7 for an illustration). This still leaves some problems, because a lost FEC packet might be incorrectly interpreted as a missing media packet, and because packets might be incorrectly re-numbered when a packet has arrived out of order.

Upon further research we found that *webrtc.org* restricts the placement of FEC packets in the stream by only adding them at the end of a VP8 frame, after the RTP packet with the M-bit set (see section 3). This restriction allows the depacketizer to distinguish between the case of a lost part of a frame and a sequence number being used for FEC, and makes re-numbering of the media packets unnecessary.

2.4 Retransmissions

RFC4585[**?**] defines a RTCP Feedback message type called NACK, with which a received can indicate to a sender that a specific RTP packet (or a set of packets) has not been received. Upon receipt of

[8]For audio, Opus' own FEC scheme which works differently than ULPFEC is used (and it is already supported in *libjitsi*).

[9]This is similar to how RAID5 works

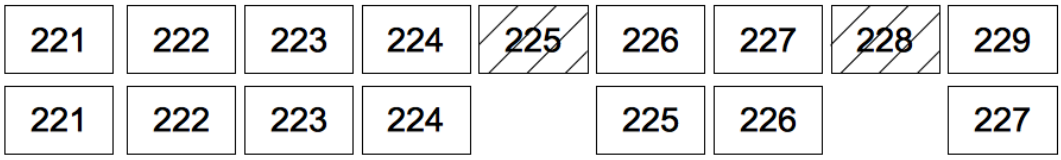


Figure 7: Re-writing sequence numbers after removal of FEC packets. The marked packets are FEC. The line above shows the sequence numbers before FEC is removed, the line below– after.

a NACK a sender might attempt to retransmit the lost packets.

In *webrtc.org* NACKs and retransmissions are used for the video streams. Current versions do retransmission by send the exact same RTP packets (without even re-encrypting, which causes some SRTP implementations to falsely detect a replay attack), but there's planned switch to using the payload format defined in RFC4588[**?**] to encapsulate retransmitted packets.

Currently *libjitsi* does not support RFC4588, but we plan to implement it (as another *PacketTransformer*). Retransmitted packets are not handled in a special way– we just ensure, for the purposes of recording, that we have buffers of sufficient size (see section **??**).

When the recording application runs on the *Jitsi-Videobridge*, requesting retransmissions with NACKs is not very important, because all RTP packets go through the bridge, and if the bridge is missing a packet, then so are the rest of the participants. The recorder can rely on the participants for sending NACKs, and just make use of the retransmissions themselves. However, when the recorder runs in a separate application (as a "fake" participant), this approach doesn't work, because packets might be lost between the bridge and the recorder. Our implementation does not yet support sending NACKs, but we plan to introduce it.

2.5 RTCP compound packets

RFC3550 specifies that two or more RTCP packets can be combined into a compoint RTCP packet. The format is very simple – the packets are just concatenated together and the length fields in their headers allow their later reconstruction.

Because of the lack of support for such packets in FMJ (and because *webrtc.org* makes use of them), we implemented it in *libjitsi* as a *PacketTransformer*.

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[8]. 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*[10].

Both VP8 in general and *libvpx* work exclusively with the I420 raw (uncompressed) image format[11]. 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[11] 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

[10] http://www.webmproject.org/license/additional/
[11] http://www.fourcc.org/yuv.php#IYUV

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.

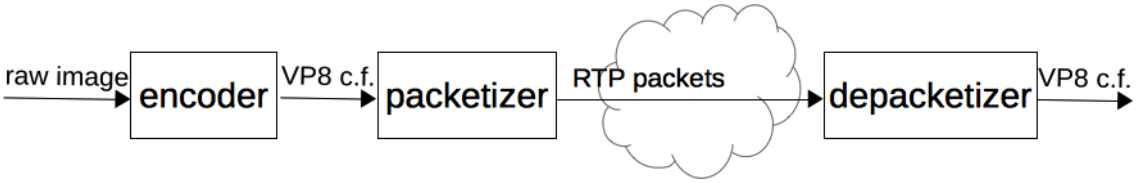


Figure 8: Using VP8 over RTP

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 *libjitsi*, the depacketizer is part of the codec chain (see fig. **??**).

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 outlines 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;
}
```

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*[12] format is a subset of *matroska*[13]. 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.

We found a library with a simple API 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 refer to lj diagram, depack lives in the codec chain

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.

We

[12] http://www.webmproject.org/about/

[13] http://www.matroska.org/

Jitter buffer? XXX add note that no decoding/encoding happens when we record video

4 Recording audio

Contrary to our initial expectations, recording audio turned out to be more time-consuming than video, with more problems coming up along the way.

Our first implementation

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

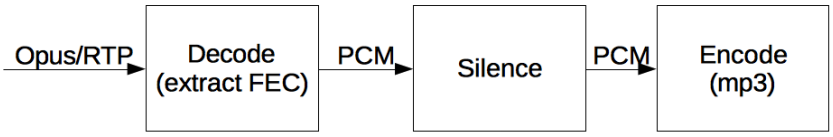


Figure 9: XXX

5 Recording metadata

In order for the post-processing application to correctly process the recorder audio and video files, it needs some additional information– a list of file names, at the very least. In the context of our recording system, we call this information metadata.

We decided to keep all metadata in a single file, and to use a JSON format to store it. This would allow to easily extend it with new fields if the need arises.

We defined a JSON object, which we call a *RecorderEvent*, and the metadata file consist of two arrays of such objects: one array for audio and one for video.

All *RecorderEvents*s have two mandatory fields: an event type, and an instant. The instant is an integer that represents the time at which the event took place, in milliseconds. No specific clock is used: only the relative time between the events in a single metadata file is taken into account. There are three types of events:

RECORDING_STARTED events signify that a recording began in a specific file (given by the "filename" field). They also contain an "ssrc" field, giving the SSRC associated with the recording. This field is used by the post-processing application to associate different files with a single participant, and is necessary in case recording of a stream is split among more than one file. These events also contain a "mediaType" field used to distinguish between audio and video, and two optional fields for additional information about the participant: "participantName" and "participantDescription". This is used in post-processing to overlay some text identifying the participant on top of their video.

RECORDING_ENDED events indicate that a particular recording ends at a specific time. These events have "filename", "ssrc" and "mediaType" fields with the same meaning as for RECORDING_STARTED events. The post-processing application uses these events to decide when to remove a certain audio or video stream from the final mix, but they are optional, because it can determine this from the actual media file.

SPEAKER_CHANGED events are used to signal that the dominant speaker in the conference has changed (at a specific time). They include an "audioSsrc" field which specifies the SSRC of the audio stream of the new dominant speaker, and an "ssrc" field which specifies the SSRC of the video stream associated with the new dominant speaker. The post-processing application uses SPEAKER_CHANGED events to change the video shown in full size.

A new examples of *RecorderEvent*s follow, and appendix B contains the full contents of a metadata file from an actual recording of a conference.

```
{
"instant" : 1395767658389,
"type" : "RECORDING_STARTED",
"filename" : "3360910907.webm",
"ssrc" : 3360910907,
"mediaType" : "video",
"aspectRatio" : "16_9",
"participantName" : "Jane Doe",
"participantDescription" : "
}

{
"instant" : 1395767704521,
"type" : "RECORDING_ENDED",
"filename" : "1277672956-2.mp3",
"ssrc" : 1277672956,
"mediaType" : "audio"
}

{
"instant" : 1395767658527,
"type" : "SPEAKER_CHANGED",
"ssrc" : 500778727,
"audioSsrc" : 1277672956
}
```

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)

XXX cname

The mappings that we have saved for each SSRC and each endpoint are illustrated on figure **??**. Given the previously saved values for $rtp0, ntp0, ntp1$ and $local1$, and given the RTP timestamp $rtpX$ for SSRC A, we calculate the local time $localX$ which corresponds to $rtpX$:

$$localX = local1 + (ntpX - ntp1)$$

where $ntpX$ is the source's wallclock time corresponding to $rtpX$, for which we have

$$ntpX = ntp0 + (rtpX - rtp0)$$

Simplifying:

$$localX = local1 + (ntp0 - ntp1) + (rtpX - rtp0)$$

Of course we have to take into account the different formats of the timestamps.

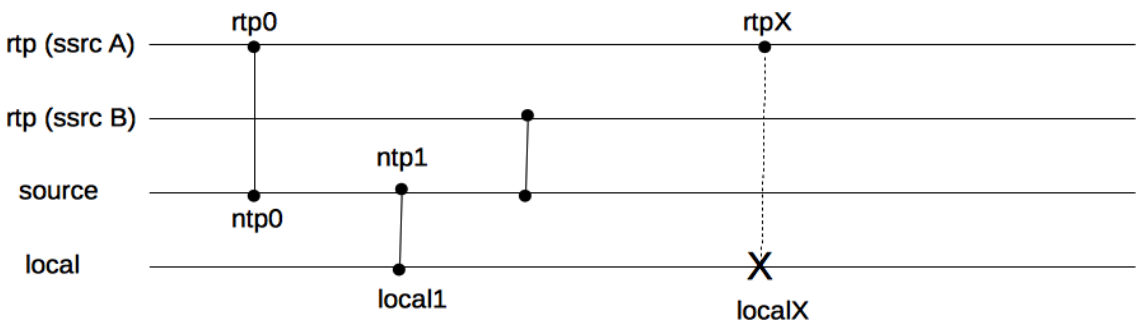


Figure 10: The mappings XXX

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 [16].

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 [16]. 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 with 4588

requesting RTX with NACK

recording in ogg/opus

postprocessing

jitter buffers

scratch "future work" altogether?

9 Conclusion

XXX Crap, I almost forgot about this! Crap!

A Jipopro

Jipopro (short for Jitsi Post-Processing) is an application which was developed by Vladimir Marinov specifically for post-processing recordings for *Jitsi Meet*. It is tied with the media formats and the metadata produced by the recording system discussed in this document.

On a very high level, *jipopro* does four things:

1. Reads a metadata file
2. Process video producing a single file
3. Process audio producing a single file
4. Merges the audio and video files together

The audio and video processing are independent (and could be done in parallel).

Video ffmpeg (command line, not API), a fixed frame rate used for output
decode (transcode to MJPEG)
do magic with sections
concat sections
encode

Audio decode, mix, encode audio:sox

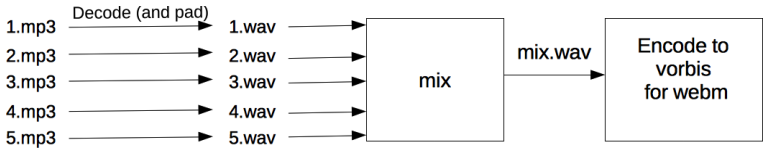


Figure 11: XXX

Merging ffmpeg configurable (mp4, webm)

Performance performance
future

B A full metadata file


```
{
  "audio" : [
    { "ssrc" : 2473760775,
      "filename" : "2473760775.mp3",
      "type" : "RECORDING_STARTED",
      "instant" : 1401807483964,
      "mediaType" : "audio" },
    { "ssrc" : 2452647913,
      "filename" : "2452647913.mp3",
      "type" : "RECORDING_STARTED",
      "instant" : 1401807484421,
      "mediaType" : "audio" },
    { "ssrc" : 1107392327,
      "filename" : "1107392327.mp3",
      "type" : "RECORDING_STARTED",
      "instant" : 1401807501709,
      "mediaType" : "audio" },
    { "ssrc" : 1565260729,
      "filename" : "1565260729.mp3",
      "type" : "RECORDING_STARTED",
      "instant" : 1401807524062,
      "mediaType" : "audio"},
    { "ssrc" : 1107392327,
      "filename" : "1107392327-1.mp3",
```

```
    "type" : "RECORDING_STARTED",
    "instant" : 1401807527409,
    "mediaType" : "audio"},
  { "ssrc" : 2452647913,
    "filename" : "2452647913-1.mp3",
    "type" : "RECORDING_STARTED",
    "instant" : 1401807532821,
    "mediaType" : "audio"}
],

"video" : [
  { "ssrc" : 2612617218,
    "filename" : "2612617218.webm",
    "aspectRatio" : "16_9",
    "type" : "RECORDING_STARTED",
    "instant" : 1401807484011,
    "mediaType" : "video"},

  { "ssrc" : 9413050,
    "filename" : "9413050.webm",
    "aspectRatio" : "16_9",
    "type" : "RECORDING_STARTED",
    "instant" : 1401807484013,
    "mediaType" : "video"},

  { "ssrc" : 2612617218,
    "audioSsrc" : 2473760775,
    "type" : "SPEAKER_CHANGED",
    "instant" : 1401807484078,
    "mediaType" : "video"},

  { "ssrc" : 9413050,
    "audioSsrc" : 2452647913,
    "type" : "SPEAKER_CHANGED",
    "instant" : 1401807485275,
    "mediaType" : "video"},

  { "ssrc" : 1190123626,
    "filename" : "1190123626.webm",
    "aspectRatio" : "16_9",
    "type" : "RECORDING_STARTED",
    "instant" : 1401807500796,
    "mediaType" : "video"},

  { "ssrc" : 2612617218,
    "audioSsrc" : 2473760775,
    "type" : "SPEAKER_CHANGED",
    "instant" : 1401807503515,
    "mediaType" : "video"},

  { "ssrc" : 2612617218,
    "filename" : "2612617218.webm",
    "type" : "RECORDING_ENDED",
    "instant" : 1401807509187,
    "mediaType" : "video"},
```

```
    { "ssrc" : 9413050,
      "audioSsrc" : 2452647913,
      "type" : "SPEAKER_CHANGED",
      "instant" : 1401807512310,
      "mediaType" : "video"},

    { "ssrc" : 3879530045,
      "filename" : "3879530045.webm",
      "aspectRatio" : "16_9",
      "type" : "RECORDING_STARTED",
      "instant" : 1401807522745,
      "mediaType" : "video"},

    { "ssrc" : 3879530045,
      "audioSsrc" : 1565260729,
      "type" : "SPEAKER_CHANGED",
      "instant" : 1401807524818,
      "mediaType" : "video"},

    { "ssrc" : 1190123626,
      "filename" : "1190123626.webm",
      "type" : "RECORDING_ENDED",
      "instant" : 1401807534861,
      "mediaType" : "video"},

    { "ssrc" : 9413050,
      "audioSsrc" : 2452647913,
      "type" : "SPEAKER_CHANGED",
      "instant" : 1401807536296,
      "mediaType" : "video"},

    { "ssrc" : 3879530045,
      "audioSsrc" : 1565260729,
      "type" : "SPEAKER_CHANGED",
      "instant" : 1401807541869,
      "mediaType" : "video"}
  ]
}
```

References

[1] G.711 : Pulse code modulation (PCM) of voice frequencies. http://www.itu.int/rec/T-REC-G.711/e.

[2] GNU Lesser General Public Licence. https://www.gnu.org/licenses/lgpl.html.

[3] Interworking between the Session Initiation Protocol (SIP) and the Extensible Messaging and Presence Protocol (XMPP): Media Sessions. https://tools.ietf.org/html/draft-ietf-stox-media.

[4] Real-time communication in web-browsers working group. http://tools.ietf.org/wg/rtcweb/.

[5] RFC3550: RTP: A Transport Protocol for Real-Time Applications.

[6] RFC5245: Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols.

[7] RFC6120: Extensible Messaging and Presence Protocol (XMPP): Core.

[8] RFC6386: VP8 Data Format and Decoding Guide.

[9] RFC6716: Definition of the Opus Audio Codec.

[10] RTP Payload Format for Opus Speech and Audio Codec. https://tools.ietf.org/html/draft-ietf-payload-rtp-opus-01.

[11] RTP Payload Format for VP8 Video. http://tools.ietf.org/html/draft-ietf-payload-vp8.

[12] Web real-time communications working group. http://www.w3.org/2011/04/webrtc/.

[13] XEP-0166: Jingle. http://xmpp.org/extensions/xep-0166.html.

[14] XEP-0167: Jingle RTP Sessions. http://xmpp.org/extensions/xep-0167.html.

[15] XEP-0340: COnferences with LIghtweight BRIdging (COLIBRI). http://xmpp.org/extensions/xep-0340.html.

[16] Ilana Volfin and Israel Cohen. Dominant speaker identification for multipoint videoconferencing. *Computer Speech Language*, 27(4):895 – 910, 2013.