UNIVERSITÉ DE STRASBOURG

Master 2
Réseaux Informatiques et Systèmes Embarqués

Submitted by
Boris Grozev
boris@jitsi.org

Strasbourg, June 14, 2014

Final Report

Media recording for multiparty video conferences based on WebRTC

Supervisor
Dr. Emil Ivov
emcho@jitsi.org

Hosting enterprise
BlueJimp

blue jimp®

I want to express my gratitude towards Emil Ivov, who, in the last two years, has provided me with a tremendous amount of help and support. As my teacher he has been an inspiration. As my boss he has allowed me to work on many interesting problems, patiently listened to my usually not very well expressed ideas, and often trusted me more than I trust myself. As my supervisor during the internship and the writing of this report, he has provided invaluable advice, always specific and spot-on. As none-of-the-above he has always been especially understanding and encouraging. Thank you, Emil!

I am also grateful to my family for their continued love and support.

Contents

Abstract

This document describes the implementation of various media recording-related features in a modern video conferencing application based on WebRTC. We start by introducing the work environment and the various technological components involved in the project. We then proceed with a detailed description of the different stages of the process including the way audio and video data is transported over the network, the way it is persistently stored, the way multiple audio and video files are organized and combined in a single flat audio/video file, and how synchronization is ensured. The work on the project is still actively pursued and we therefore also present a number of planned future steps and improvements that will likely be implemented in the near future.

1 Introduction

The ever decreasing cost of bandwidth and processing resources have in the recent years made multi-party video conferencing over the internet viable for personal use. The advent of the WebRTC technology that adds audio/video communication capabilities to web browsers, has made the development of conferencing applications (or the addition of conferencing features to existing applications) simpler than ever before.

The work described in this document is about the development of video recording features within *Jitsi Meet*: an existing WebRTC video conferencing application. Throughout the rest of this section we introduce the working environment and the most important standards and software products used in *Jitsi Meet*. In section 2 we define what we mean by recording a conference and introduce the possible general approaches to the problem. In sections 4 through 8 we examine specific parts of the recording process in detail. In section 9 we review accomplished work, and in section 10 we review new features and optimizations to existing features which we plan to develop in the near future.

1.1 *BlueJimp*

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.

[1] https://bluejimp.com

[2] https://jitsi.org

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[31]. 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).

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*[32]. 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. *Jitsi Meet* is discussed in more detail in 1.6.

[3] Although a REST API is also available.

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[38] at W3C, and the on-the-wire protocols, being defined within the *RTCWEB* working group[36] 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. There is also the possibility to pass arbitrary data in an application-defined format. These 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)[18] 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 (defined in XEP-0166[7] and XEP-0167[8]) 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[16], ZRTP[26], DTLS-SRTP[3], and RFC4575[17] can also be used with *Jingle*. Mappings have been defined between the two[19], which allow gateways or rich clients to organize cross-protocol calls.

1.5 COLIBRI

COnferencing with LIghtweight BRIdging (COLIBRI, defined in XEP-0340[4]) is an XMPP extension developed mostly in *BlueJimp* for use with *Jitsi Videobridge*. It pro-

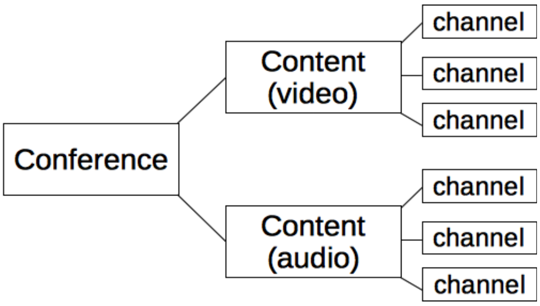


Figure 1: A COLIBRI conference.

vides 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 *Channel*s, separated in different *Content*s (see fig.1).

In the most common use case a client requests the creation of a *Conference* with a specified number of *Channel*s. The mixer allocates local sockets for each *Channel* and provides their addresses to the client. The client then uses these transport addresses as its own to establish, for example, a *Jingle* call with another participant. 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 a *Conference* has been 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*.

The protocol is being extended for the purposes of *Jitsi Meet* (see the next section), and now also has support for establishing *Channel*s which use DTLS[14], and establishing special *Channel*s for use with WebRTC data channels. It also supports the starting and stopping of the recording for a specific *Conference* (which was a small extension implemented as part of the recording effort described in this document).

1.6 *Jitsi Meet*

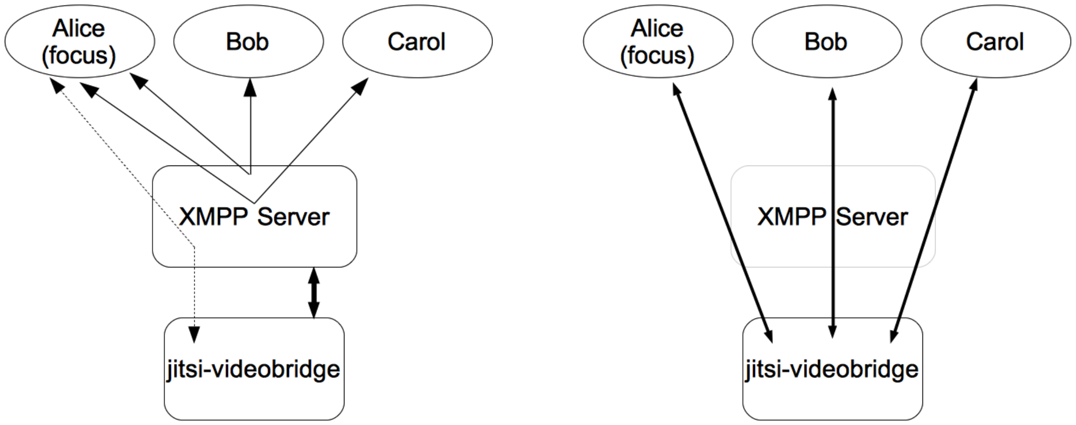
Jitsi Meet uses the above-mentioned technologies to create a multi-party video conference. The endpoints of the conference are simply WebRTC-enabled browsers[4] running the actual *Jitsi Meet* application. They all connect to an XMPP server and join a Multi-User Chat (MUC) chatroom. One of the participants (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

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

direct RTP sessions with *jvb*.

The resulting connections for signalling and media are depicted in figures 2a and 2b respectively.



(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.

The *Jitsi Videobridge* instance runs as a relay (as opposed to a mixer) for both video and audio (meaning that it only passes RTP packets between the participants, without considering their payload).

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 3 is a screenshot from the actual application. Current work is underway to use dominant speaker identification (see section 8) 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 control over their conferencing solution.

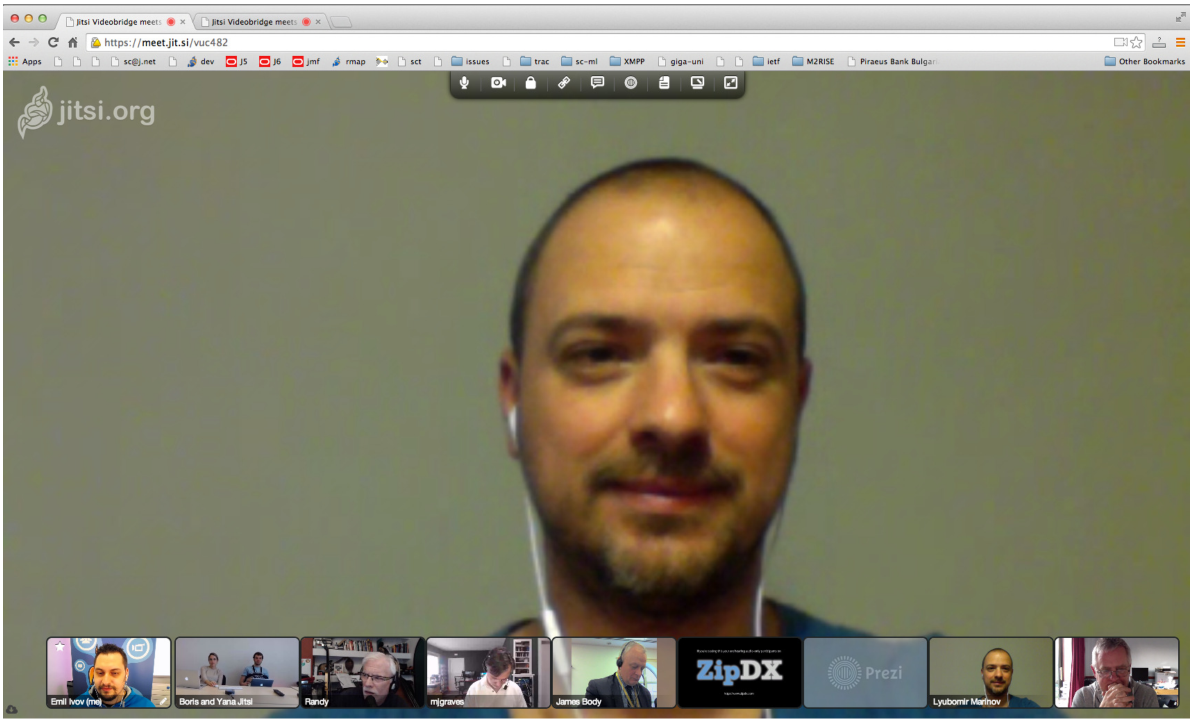


Figure 3: A screen capture from a *Jitsi Meet* conference.

2 Recording a *Jitsi Meet* conference

2.1 What we mean by recording

By recording a multimedia conference in general we mean the following: all audio and video flows exchanged during a conference are 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**: storing all video flows (0, 1 or more per participant) in separate, single track files.
- **Recording audio**: storing all conference audio flows (usually 1 or more per participant) in either a single mix or separate single track files.
- **Recording metadata**: persisting all non-media information that is important for the reconstitution of a conference.
- **Post-processing**: combining all recorded audio, video and metadata in a single audio/video file.

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 4 and 5.

By metadata we mean all additional information (apart from the media itself) which is necessary to play back the conference later. This includes participant names, filenames, synchronization source (SSRC) identifiers, flow start and end times, etc. A detailed discussion of the metadata that we use is provided in section 6.

Post-processing in our case means taking all the recorded data and producing a single file with one audio track and one video track, which can be easily viewed, manipulated and uploaded and viewed at popular video streaming platforms[5]. The details can vary, 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.

2.2 Which entity performs recording

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. This is inconvenient because users would need to use modified browsers, and because in most use-cases the recordings are going to be stored (and post-processed) on a server, so they would have to be transferred there somehow. 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 runs on a server (and without the need for a browser at all). Still, it connects as a normal participant and establishes an RTP session with *Jitsi Videobridge* (see figure 4).

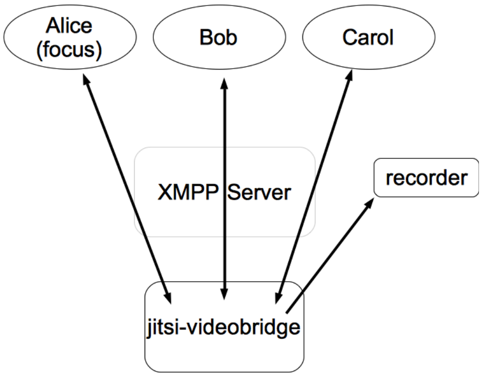


Figure 4: A recording application connected to a *Jitsi Meet* conference as a "fake" participant.

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.

Shunyang Li, a student from Peking University, is currently working, under my guidance and within context of the Google Summer of Code program, on *Jirecon*[33]– a standalone XMPP container for the recording application described here.

[5] Such as YouTube and Daily Motion

3 Implementing support for the WebRTC transport layer

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, which made our task of implementing support for the WebRTC transport significantly more manageable.

It is mandatory to use the Interactive Connectivity Establishment (ICE[16]) 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[14]) to exchange key material, which is then used to generate session keys for a Secure Real-Time Protocol (SRTP[2]) session. The procedure is defined in RFC5763[3]. In a *Jitsi Meet* conference, each participant's browser sets up two secure RTP sessions with *Jitsi Videobridge* in this way.

RTP provides an unreliable transport. For this reason *webrtc.org* uses a couple of mechanisms on top of RTP to improve the quality of the media. Implementing support for these mechanisms in *libjitsi* was one of the most significant efforts during our recording project. These efforts are discussed in detail in sections 3.2 to 3.4. Before that, however, section 3.1 gives an overview of the existing RTP stack in *libjitsi*.

3.1 The RTP stack in *libjitsi*

libjitsi makes heavy use of the Freedom for Media in Java (FMJ[29]) 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*.

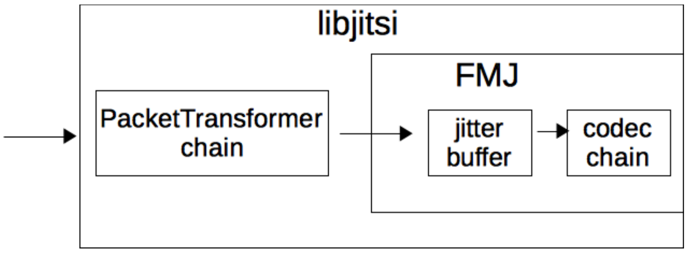


Figure 5: General scheme of the RTP stack in *libjitsi*.

The RTP stack used by FMJ 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 5 illustrates this scheme. *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, 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 labelled "RFC6464" is used to extract the audio level information from packets, which include this information in an RTP header extensions defined in RFC6464[5]. The audio levels are used for, among other things, performing dominant speaker identification (see section 8). This transformer also serves as a filter, dropping packets with audio marked to contain silence (in order to avoid unnecessary processing, which is why the transformer is first in the chain). 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.

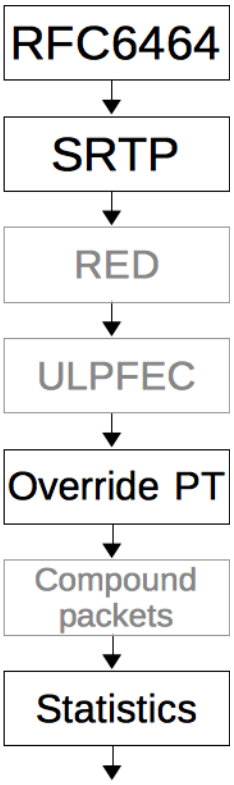


Figure 6: The chain of *PacketTransformer*s in *libjitsi*. The shaded elements are new additions.

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.

3.2 RED

Our first task was to implement support for implement support for the RED payload format for RTP. This format is defined in RFC2198[12] and 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.

3.3 Uneven Level Protection Forward Error Correction

Our next task was to implement support for the Forward Error Correction (FEC) format used in *webrtc.org*. In general, (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[6] 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 for the video streams[6], 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"). When a sender uses FEC for a particular stream, it 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[7].

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).

The sender can control the amount of FEC packets it adds to a stream by changing the number of protected packets, and it can do this dynamically, adapting to network conditions. This is the most common usage of FEC, and the one currently employed by

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

[7] This is similar to how RAID5 works.

webrtc.org: the sender allocates a given fraction of the configured bandwidth to FEC, and this fraction changes depending on the packet loss statistics received with RTCP. The aim is to mitigate the effects of packet loss without the need for retransmissions.

Another way to use FEC is for probing the available bandwidth. When the sender detects stable network conditions, it wants to increase its sending bitrate, in order to improve quality. However, this risks causing a congestion, and therefore packet loss. The sender initially increases its sending rate by significantly increasing the amount of FEC. In this case, even if a congestion occurs, the receiver is more likely to be able to reconstruct the media packets (without the need for retransmissions). The sender then monitors the following RTCP reports. If they indicate a high percentage of packet loss, the sender goes back to the previous, lower rate. Otherwise, the sender keeps the total bitrate, but decreases the rate of FEC, using the available bitrate for the encoder instead, thus improving the video quality. This scheme is examined in [10] and it may represent a surface for future improvement of the *Jitsi Videobridge* and *Jitsi Meet* platforms.

3.3.1 Implementation of ULPFEC

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

We keep two buffers of packets: $bMedia$ and $bFEC$. With every FEC packet f we associate the number $numMissing(f)$ of media packets protected by f which we have not received (but we could receive later on).

When we receive a new media packet, we recalculate the values $numMissing(f)$ for all f in the $bFEC$ buffer. Then, for all f in $bFEC$: if $(numMissing(f) == 0)$, then we remove f from $bFEC$. If $numMissing(f) > 1$, then we do nothing. If $numMissing(f) == 1$, we use f and $bMedia$ to reconstruct a media packet and then we remove f from $bFEC$.

When we receive a FEC packet f, we calculate $numMissing(f)$, and apply the same procedure as above.

We have limited $bFEC$ to a small size, and if $bFEC$ is full when we receive a new FEC packet, we drop the oldest packet from it. In this way, we prevent "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.

3.3.2 Re-writing RTP sequence numbers

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 4.2), 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 we might incorrectly interpret a lost FEC packet as a lost media packet, and because we might incorrectly renumber some packets if they arrive out of order.

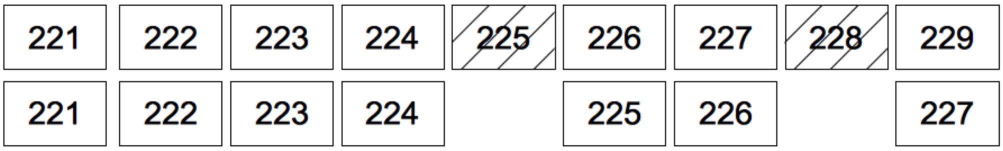


Figure 7: Re-writing sequence numbers after removal of FEC packets. The marked packets are FEC. The line above shows the sequence numbers before FEC is removed, the line below– after.

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 4). This restriction allows their depacketizer to distinguish between the case of a lost part of a frame and a sequence number being used for FEC, and allows their implementation to work without the unnecessary and awkward complication of rewriting sequence numbers.

We updated our implementation – all that was required was to remove the code which does the sequence number change.

3.4 Retransmissions

Our next task was to understand how *webrtc.org* uses RTP retransmissions, and implement support for them in *libjitsi*. We found that *webrtc.org* uses RTCP NACK messages (an RTCP Feedback Message type defined in RFC4585[11]) from the receiving side, in order to notify the sender that a specific RTP packet (or a set of packets) has not been received. When a sender received a NACK message, it attempts to retransmit the lost packets (i.e. retransmits the lost packets, if they are still in its buffers).

The *webrtc.org* code uses NACKs and retransmissions only for the video streams. Current versions do retransmissions by sending 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[15] to encapsulate retransmitted packets.

Currently we do not support RFC4588 in *libjitsi*. We plan to add support for it, in the form of a *PacketTransformer*. It will strip the additional RFC4588 headers and pass on exact copies of the original packets. Then, in the rest of the library, we will handle the packets in the same way as we currently handle non-RFC4588 retransmission, which is nothing special: we just ensure that we have buffers of sufficient size so that we don't drop retransmitted packets because they arrive too late (see section 4.5).

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.

3.5 RTCP compound packets

RFC3550 specifies that two or more RTCP packets can be combined into a compound 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*).

4 Recording video

4.1 The VP8 codec

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 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[1]. 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 open-source license. They also provided a statement granting permission for royalty-free use of any of their patents used in *libvpx*[27].

Both VP8 in general and *libvpx* work exclusively with the I420 raw (uncompressed) image format[40]. A component called a VP8 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[25] defines how to transport a VP8 compressed frame over RTP. In short, the process involves optionally splitting a VP8 compressed in parts, prefixing each part with a structure called a ”VP8 Payload Descriptor”, and then encapsulating each part in RTP. This process is referred to as packetization, and the reverse process (of collecting RTP packets and constructing VP8 compressed frames)– depacketization. Figure 8 provides a high-level overview of the use of VP8 with RTP.

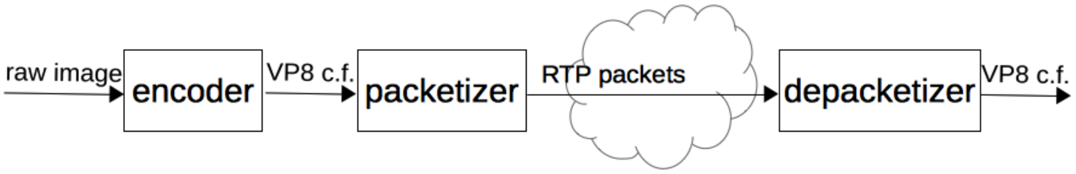


Figure 8: Using VP8 over RTP (c.f. stands for "compressed frame")

libjitsi already has a VP8 implementation, which we use in the *Jitsi* client. It consists of four parts: an encoder and decoder (wrappers around *libvpx*), a packetizer and a depacketizer. For the purposes of recording, we only need 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.

4.2 Depacketization

The VP8 Payload Descriptor can be thought of as an extension to the RTP header. It is included in the beginning of the RTP payload for every RTP packet with the VP8 format. It has variable length (between 1 and 6 bytes) and 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.

When we do depacketization, we use the above three fields, as well as the following fielsa 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 FMJ codec chain (see fig. 5).

In order to decide whether a received packet is from a new frame or not, we use the RTP timestamp and the *PictureID* fields. If either of them don't match, we assume that the packet is from a new frame).

We use the *PID* and *S* fields from the VP8 Payload Descriptor to detect the first packet of a frame– the first, and only the first packet have both the *S*-bit set and *PID* set to 0.

This observations allow 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;
}
```

4.3 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 not many players support it (for example browsers don't support it), and that it's lack of extensibility.

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 – none of the other tools which we use (like *ffmpeg*) would be able to handle it, and we would need to implement our own decoder.

The *webm*[39] format is a subset of *matroska*[34]. 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, and that it supports many features, so we can later extend our implementation if needed.

We found a small library (written in C++, but with Java mappings already available) with a simple API that would allow us to write *webm* files easily, so we decided to ignore *ivf*, postpone the potential definition of a new format, and use *webm*.

We adapted the library to our needs, and implemented a *WebmDataSink* class which takes as input a stream of VP8 compressed frames and saves them in a *webm* file.

A VP8 stream transported over RTP does not have a strictly defined frame rate. Each VP8 frame has it's own, independent timestamp, generated by the source (usually at the time of capture from a camera, before VP8 encoding has taken place), and this timestamp gets translated into an RTP timestamp and is carried in RTP.

When we save VP8 in *webm* format, we use the RTP timestamp in order to calculate a presentation timestamp. This is simple– for the first recorded frame we use a presentation timestamp of 0 (in milliseconds) and for subsequent frames we calculate the difference from the first frame.

4.4 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. In order to start working, a VP8 decoder needs to first decode a keyframe. Therefore, we want the recorded *webm* files to start with a keyframe.

Because keyframes are rarely sent, and because we want to be able to start recording a conference at any time, we needed a way to trigger the generation of a keyframe. One way to do this, which is supported by *webrtc.org*, is by the use of RTCP Full-Intra Request (FIR) feedback messages (defined in RFC5104[24], section 3.5.1).

We added support for FIR messages in *libjitsi* and made use of them to request keyframes in the beginning of a recording. We faced a difficulty because FIR messages contain an "SSRC of packer sender" field, and *webrtc.org* clients only accept messages from SSRCs that they know about (that is, that have been specifically added via signalling). We had to make *Jitsi Videobridge* generate and use it's own SSRC, which it also announces to the focus via COLIBRI.

4.5 Jitter buffer

A jitter buffer is normally used in a real-time multimedia application to introduce a certain amount of delay in the playback of media, mitigating the effects of the varying delay of packets on the network (the jitter). A buffer which is too small gets emptied quickly when packets are delayed and playback has to be paused. A buffer which is too big adds unnecessary delay to the playback. For this reason adaptive jitter buffers are used, which change their size according to network conditions.

In the case of video with WebRTC, apart from just jitter on the network, there might be packet retransmissions triggered by the receiver (which necessarily arrive at least one RTT later than the originally transmitted packets), and it is beneficial if the buffer is large enough to accept them (as opposed to dropping them because they have arrived too late).

For the purposes of recording, a relatively long delay (on the order of a few seconds) is acceptable, provided that the buffer can be emptied at the end of the recording, without packets being discarded. For this reason we decided to use a fixed-size jitter buffer.

FMJ already includes a jitter buffer in its chain (see figure 5), but for technical reasons it was hard to implement a way to empty it without discarding its contents. We decided to implement our own buffer, in the form of a *PacketTransformer*. For simplicity, we limited its size to a given number of packets, and not to a given length of time. We used a default size of 300 packets, since we observed that this usually corresponds to between 3 and 10 seconds.

4.6 Overview

The components discussed above are pieced together in a *libjitsi* class named *RecorderRtpImpl*[8], which we implemented specifically for recording video[9].

RecorderRtpImpl takes its input in the form of RTP packets. It first demultiplexes the packets by their SSRC, and puts them in a jitter buffer, which is placed as the last element of the *PacketTransformer* chain.

After they exit the jitter buffer, all packets are passed to an instance of FMJ which is configured to transcode from the "VP8/RTP" format, to the "VP8" (i.e. to do depacketization). FMJ does it's own demultiplexing and creates a VP8 depacketizer for each stream. The depacktizer is part of the FMJ codec chain (again, see figure 5), and the internal FMJ jitter buffer is practically not used. FMJ provides its output in the form of multiple *DataSource*s (one for each stream), which represent, in essence, a stream of VP8 compressed frames. We take each of these *DataSource*s and pass it to a *WebmDataSink* instance, which produces the final *webm* file.

As soon as we detect a new VP8 stream, before its packets enter the jitter buffer, we request a keyframe by sending an RTCP FIR message.

When the recording of a stream is stopped (either because of a user request via the API, or because of an RTCP BYE message, or because of a timeout), we empty the jitter buffer for the specific SSRC, processing its contents.

Using this process, we save each stream in a separate *webm* file.

5 Recording audio

The WebRTC standards define two mandatory to implement audio codecs: *G711*[30] and *Opus*[22]. All WebRTC endpoints are required to implement them.

G711 is an audio codec originally developed in the 1970s by the ITU for use in the telephone network. It works on an input PCM signal with a sampling rate of 8000Hz (which already limits the sound quality) and produces an encoded signal with a constant bitrate of 64kbps. It is included in WebRTC for interoperation with legacy devices.

5.1 Opus

Opus is a modern audio codec, whose definition was published as an RFC in 2012. It has a variable bitrate and works with fullband sound (sampled at 48000Hz). The average bitrate is configurable, and at 32kbps (the default in many applications which encode voice) it produces sound with remarkable quality.

In a *Jitsi Meet* conference the sound codec is always *Opus*, because it produces the best quality among the available codecs and because all participants use full-fledged browsers which support it.

[8] Because it implements the *Recorder* interface, using RTP as opposed to a capture device as input

[9] Although we later adapted it to record audio as well.

Opus supports frames of different size, but almost all applications (including *webrtc.org*) use frames of 20ms.

Opus has a build-in FEC mechanism which works differently than the RFC5109 mechanism discussed in section 3.3. If enabled for a packet, it encodes the packet at a lower quality, and attaches this version to the subsequent packet. This allows a single lost packet to be recovered (in lower quality), but if two consecutive packets are missing, only one can be recovered. It also has a Packet Loss Concealment (PLC) function, which can be used in the absence of a packet to produce a low level of noise in such a way as to reduce audible crackling.

libjitsi has support for *Opus* and for it's FEC and PLC features.

5.2 Recording a mix directly ("live" mixing)

When we started work on recording audio, the most straightforward solution, given the existing code, was to create a mix on-the-spot (in the recording application) and record a single file. This was because *libjitsi* has an advanced audio mixer component, which is used in the *Jitsi* client and *Jitsi Videobridge* (if it is so configured) to mix the audio for a conference. The API allows the addition of a basic RTP stream, and handles all the processing (decoding, resampling, if needed) automatically (see figure 9). Also, the audio mixer serves as a capture device, and can thus be recorded with the existing *libjitsi* recorder.

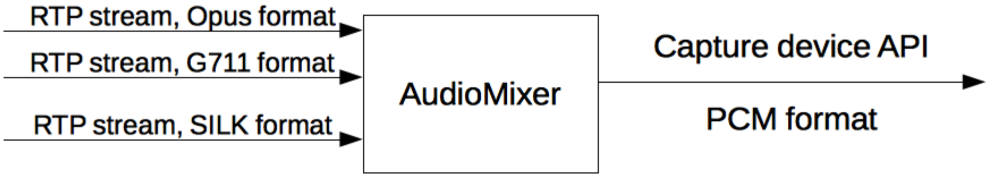


Figure 9: The API of the *libjitsi* audio mixer

Indeed, the implementation proved easy to do, but we faced a problem when we started to think about synchronizing the recorded audio and video. In order to do this, we would at the very least need to know the exact moment where a stream has been added to the mix. This was very hard to do with the audio mixer API.

We would also need a way to guarantee that the spacing between two audio packets in the mix exactly matches the spacing between the packets at the source. We had no way of doing this with the mixer.

We decided that we needed to give up using the *libjitsi* audio mixer and implement code which would allow us finer control over the whole process.

5.3 Recording streams separately

Since we were going to implement the code to handle the decoding and recording of audio streams anew, doing mixing in the application would require additional work, and would introduce substantial complexity. Additionally, there are already many tools which allow the mixing of already recorded files. For these reasons, we decided to record streams individually.

We already had a *RecorderRtpImpl* (see section 4.6) which we used for video, and which performed many of the things needed for audio as well. We adapted *RecorderRtpImpl* so that it could handle streams with different formats, including audio formats.

The existing code in *libjitsi* supports recording only in *mp3* format. We decided to use this, for the time being, because it is relatively easy to transcode it in post-processing to whatever format is needed.

For an audio stream, we use FMJ to decode (to a PCM format), and then encode it and save it in *mp3* format.

5.4 Repairing gaps

In contrast to video in a *webm* file, in which frames have their individual presentation timestamps, an audio stream consists of just sound samples to be played one after the other. This means that if some samples are missing for some reason (e.g. because of a packet lost in the network), the resulting stream will be shorter than the original. When recording a conference, which might last a relatively long time (possibly hours), the missing samples accumulate, and, when audio and video are merged in the end, this leads to audio and video drifting apart. In order to solve this problem, any gaps in the audio streams need to be repaired.

The *libjitsi* implementation of *Opus* already repairs gaps on two levels while decoding. First, it tries to use FEC, which can be used when only a single packet is missing. Then, in case of 3 or less missing packets, it uses *Opus*' PLC. But gaps of more than three packets are not repaired.

Apart from lost packets, there is another cause of gaps in the audio streams. When participants in a conference use the "mute" functionality, their browsers continue to send audio packets. However, these packets are specifically marked as containing silence using RFC6464, and *Jitsi Videobridge* drops them. When a participant unmutes, this results in a gap, which is possibly many minutes long.

Repairing such a gap by adding silence samples would require non-negligible amount of computation, because all these samples will then need to be encoded. In such cases it would be better to re-start the recording for the stream in a new file, giving it its own timestamp in the metadata, so that it can be properly mixed in post-processing.

To solve this problem, we decided to introduce an element in the FMJ codec chain, which would be placed between the *Opus* decoder and the *mp3* encoder, and would detect and handle gaps. We called this the *SilenceEffect*.

It works by monitoring the timestamps of the buffers which pass through it. It handles small gaps (the length of gaps to consider short is configurable, with a default of 3 seconds) by inserting the necessary amount of samples (accurate up to a single sample). For gaps longer than this, it signals to the *RecorderRtpImpl*, which restarts the recording into a new file.

This scheme allows us to know, for each recorded audio file, the RTP timestamp of the first RTP packet, the audio from which is contained in the file. This piece of information allows us to ensure the synchronization of the resulting audio and video files (see section 7).

5.5 Overview

Initially we started with recording a mix of all audio streams. We then changed to recording streams individually, because we discovered limitations with the mixing approach.

In the current implementation each incoming audio stream (which is usually *Opus*, but any of the codecs supported in *libjitsi* can be used as well), is first decoded (and re-sampled to 48000Hz, if necessary), then gaps are repaired by inserting silence. Then the stream is encoded and saved in an *mp3* file. See figure 10 for an illustration. If a gap of more than 3 seconds is detected, we start recording in a new file.

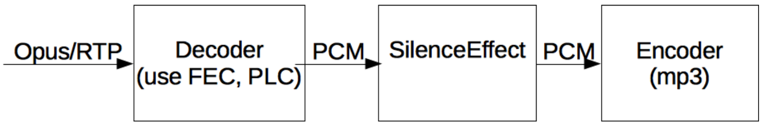


Figure 10: The chain used to transencode audio for the purposes of recording.

This solution involves re-encoding and that makes it suboptimal – it is more computationally expensive than necessary and reduces the sound quality of the recording. A better solution would be, whenever possible, to record in a container format that can accommodate the original format, and thus does not require transcoding or even decoding (as we do for video). For *Opus*, a viable solution would be to use the ogg/opus format and we plan to implement it (see section 10.2).

6 Recording metadata

In order for the post-processing application to correctly process the recorded 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 *RecorderEvent*s have two mandatory fields: an event type (”type”), and an instant (”instant”). The instant is an integer that represents the time at which the event took place, in milliseconds. The instant has no global interpretation, it is only used to provide timing information relative between the events in a single metadata file. There are three types of events:

- **RECORDING_STARTED**
  We use these events to indicate that a recording has begun in a specific file (given by the ”filename” field), at a specific time. These events have an ”ssrc” field, which stores the SSRC associated with the recording. We use this field in the post-processing application to associate different files with a single participant, and it is necessary in case recording of a stream is split among more than one file.

  RECORDING_STARTED 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”. We use this information in post-processing in order to overlay text identifying the participant on top of their video.

- **RECORDING_ENDED**
  We use these events to 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**
  We use these events to indicate 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 of the new dominant speaker. The post-processing application uses SPEAKER_CHANGED events to change the video shown in full size.

A few examples of *RecorderEvent*s follow, and appendix B has 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
}
```

7 Synchronization

As explained in the previous section, for each recorded audio and video file, we save a *RecorderEvent*, and each such event has an "instant" field containing an integer specifying milliseconds. This fields gives the relative time at which the audio or video should be included in the final mix. This section describes how the values of the "instant" fields are calculated, in order to provide proper synchronization between the audio and video of a participant and between different participants in the final result produced by the post-processing application.

7.1 Synchronization between streams from a single source

If we use a simple and naïve method, for example, saving the local time when a recording starts, this would leave the audio and the video stream of a participant not synchronized, because of network jitter and differences in the time used for local processing. Therefore we would like to somehow use timing information, which comes from a place as close as possible to the source of the media.

According to its definition, the timestamp field in an RTP packet "reflects the sampling instant of the first octet in the RTP data packet"[20]. When we record video, we know the RTP timestamp of the first frame written, and similarly for audio we know the RTP timestamp of the first packet included in the recording[10].

We implemented a component, named *Synchronizer*, which is responsible for translating RTP timestamps (coming from different RTP streams) into a time on a common

[10] The initial implementation of the recording didn't expose this information, we had to add it for the purpose of synchronization.

clock. For the common clock we used the local system clock, with standard UNIX time represented in milliseconds. We then use times acquired from the *Synchronizer* directly when writing *RecorderEvent*s.

The RTP timestamp is a 32bit unsigned integer field, which uses a frequency specific to the media format. For example, *Opus* uses a frequency of 48000Hz, and *VP8* uses 90000Hz. This means that two *Opus* packets, generated exactly one second apart will have RTP timestamps differing by 48000. The initial RTP timestamp is chosen at random.

When two RTP streams (for example one audio and one video) come from the same source (the same participant in a conference), their RTP timestamps are not directly related, but they share the same wall clock from which the RTP timestamps are generated.

7.1.1 Saving mappings

RTCP Sender Report packets contain two timestamp fields. One is an RTP timestamp, and the other is a representation of the source's wall clock in NTP timestamp format[9]. Both fields represent the same instant, so we can think of the pair as a mapping. Having this mapping allows us to calculate the relative difference between RTP timestamps from different streams (provided that they come from the same source, of course).

For each SSRC we save one such mapping of an RTP timestamp to an NTP timestamp. On the network the NTP timestamp is a 64-bit fixed-point number (with 32 bits for the whole part and 32 bits for the fraction) in seconds, but we locally save it as a *double* (still in seconds).

Then, for each source, we save a pair mapping an NTP timestamp to a local time. It is important to only use one such mapping, because if we make a second mapping, we are likely to get inaccurate results because of network jitter. To create such a mapping, we use the time of reception of an RTCP Sender Report and the NTP timestamp in said report.

7.1.2 The calculation

After we have these two necessary mappings– from the RTP clock of a specific SSRC to the source's clock, and from the source's clock to the local clock – we can perform translation from the RTP clock to the local clock. The procedure is illustrated on figure 11. The calculation is not complex: using the notation from the figure, to translate the RTP timestamp $rtpX$ coming for SSRC A to its corresponding local time $localX$ we calculate:

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

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: RTP timestamps are integers with an SSRC-specific rate which are 32-bit and wrap, the local timestamp is an integer in milliseconds, and the NTP timestamps are (saved locally as) *doubles* in seconds.

So the final calculation is something like

$$localX = local1 + 1000 * (ntp0 - ntp1) + 1000 * diff(rtpX, rtp0)/rateA$$

where $rateA$ is the RTP clock rate for the SSRC A, and $diff$ calculates the difference taking into account the wrap at 2^{32}.

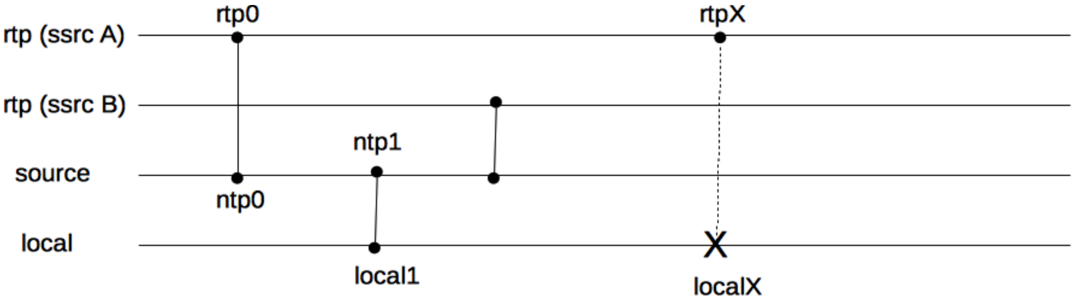


Figure 11: The mappings saved for a single source

7.1.3 Identifying sources

By specification, two SSRCs can be recognized to come from the same source and thus share the same wallclock by looking at the CNAME field in RTCP SDES packets. This would allow for a very simple API for the *Synchronizer* in which it is fed RTCP packets, and recognizes which SSRCs are grouped together automatically.

However, there is a bug in Chrome[35], which makes it use different a CNAME for each SSRC, making it impossible to deduce which SSRCs come from the same source by only looking at RTP and RTCP packets. We implemented a workaround, in which we explicitly specify to the *Synchronizer* identifiers for the source of each SSRC (and we get this information from the signalling path).

We support the CNAME mode of operation and plan to switch to it if the bug in Chrome is fixed.

7.1.4 Overview

The scheme just presented allows for audio and video to be very accurately synchronized, because it takes into account timestamps sampled at the source of the streams. It has the drawback that the *Synchronizer* needs to be initialized (with an RTCP Sender Receive packet, and with an endpoint identifier currently coming from signalling) before it can perform translation for a given SSRC. Without this initialization, the calculation simply cannot be performed. This makes the usage of the *Synchronizer* slightly inconvenient.

7.2 Synchronization between different sources

When streams come from different sources we have no mechanism which allows us to synchronize them with absolute accuracy. As explained in the previous section, we keep a single mapping of a time of the source's wall clock to a local time, for each source. We save these mappings independently. We perform the measurement of the local time as soon as possible (as soon as the packet is received by the recorder), in order to minimize the effects of the varying time used for local processing.

These same limitations apply for clients participating in a conference, trying to render multiple incoming streams together.

8 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 [23].

In a *Jitsi Meet* conference there are two use-cases for DSI: for "live" use 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, following the dominant speaker). In this case DSI is performed on *Jitsi Videobridge*, which uses WebRTC data channels[11] 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 by the recording application[12], and changes of the dominant speaker are saved as metadata.

8.1 Implementing dominant speaker identification

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

[11] These are channels which carry application data (and not audio or video), but are negotiated and established in the same way as the media channels.

[12] Although, if the recording application is running as a participant, it can potentially also open a WebRTC data channel to the *Jitsi Videobridge* and use the events received there, instead of doing DSI itself.

We 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 we consider one stream "active", and while the rest– "competing". With each stream we associate a score. When we measure new audio levels for a stream, we recompute its score, and then examine the scores of the streams in order to determine if one of the competing streams should replace the 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 speech, 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. We chose the values based on a few uncontrolled experiments we performed. We used MIN_ACTIVE = 1000 and $C = 1.15$ (MIN_SCORE is too arbitrary to deserve mention).

We compute scores 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, another *BlueJimp* and *Jitsi* team member has now taken over this task, and has implemented an improved algorithm that more closely resembles the one proposed in [23]. However, his implementation still relies only 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[5]. This allows *Jitsi Videobridge* to do DSI very efficiently, without the need to decode the received audio streams.

9 Conclusion

During the last five-and-a-half months we have worked on the implementation of a system for recording video conferences. We have have successfully brought multiple non-trivial components from the design stage to a final working implementation. We have also learned the shortcomings of some of our initial ideas and thus gathered valuable practical experiences.

The work on the recording system has been, to a large extent, funded by the *Comcast Cable* internet service provider, now a *BlueJimp* customer. The implemented procedures therefore allow *BlueJimp* to assist *Comcast* in delivering rich real-time communication services to its subscribers.

Currently, the project is in a working state– recordings are produced as described in the previous chapters, in a format which allows for their successful post-processing. However there are further improvements and optimizations which we can do, and plan to do in the near future. These are discussed in the next section.

10 Future work

The recording system examined in this document is now fully functional, but there's room for many improvements and optimizations. This section discusses some of the work planned for the near future.

10.1 RTP retransmissions

As discussed in section 3.4, there are two things related to RTP retransmissions which need to be implemented. Handling of packets retransmitted using the format defined in RFC4588 needs to be added, because the next release of Chrome/Chromium is expected to use it. We also need to be able to request retransmission by use of RTCP NACK packets, in order to improve video quality when the recorder is run as a separate application (as opposed to on *Jitsi Videobridge*).

10.2 Recording audio in *ogg/opus* format

Currently, when we record and post-process audio, we perform two transcoding operations on the stream. First, we transcode from *Opus* to *mp3* while recording, and then we transcode to *vorbis* in post-processing. We can avoid one of the transcodings by recording audio directly in a container format which supports *Opus*. This would both make the recording procedure more efficient, and (slightly) improve the quality of the final sound.

We are planning to implement this idea by recording directly in the *ogg/opus* format (without decoding). One difficulty that we face, which we don't yet know how to solve, is the handling of FEC in this case. In the current implementation we control the *Opus* decoder and we repair gaps using FEC, whenever possible. The *ogg/opus* format supports filling gaps, but the specifications[13, 21] do not define a way to handle FEC, and the currently available tools don't support it.

10.3 Support for payload type mappings

The current implementation uses predefined mappings for the RTP payload type numbers. We need to implement support for set them dynamically. One complication arises, because we currently do not support re-writing of the payload type numbers for encapsulated formats such as RED and RTX.

10.4 Improving post-processing

The post-processing application (see appendix A) has limited performance, and our examinations so far show that it would require significant changes to allow it to work sufficiently fast. We are planning a major re-design.

A Jipopro

Jipopro (short for Jitsi Post-Processing) is an application which was created specifically for post-processing recordings for *Jitsi Meet*. It was developed in parallel with the recording implementation discussed in this document, and is closely tied with the formats for media and metadata produced by that system. *Jipopro* is written in Java, but uses external applications for many of its tasks. It was mainly developed by Vladimir Marinov, then in *BlueJimp*.

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

1. Reads a metadata file.

2. Processes video, producing a single file.

3. Processes audio, producing a single file.

4. Merges the audio and video files together.

The audio and video processing are independent.

A.1 Video

For all video handling, *jipopro* uses *ffmpeg*[28].

First every input *webm* file is transcoded to an MJPEG file with a static frame rate (25fps by default). Then, a timeline of events is constructed according to the metadata. The events divide the timeline into sections. Figure 12 shows an example.

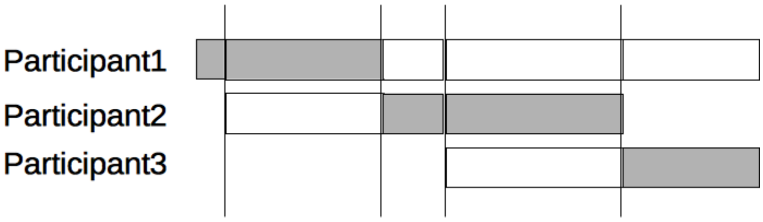


Figure 12: A representation of the process of combining videos. There are 5 sections, separated by vertical lines. The shared regions represent the dominant speaker.

For each section, a list of participants, one of whom is active, is maintained. The sections are then processed: a ”slice” of video is taken (with a specific offset) from the MJPEG file for every participant, and the obtained videos (all with the same length) are decoded, combined together (overlaid) and encoded in an MJPEG file again.

After all sections have been processed, they are concatenated together, resulting in a single MJPEG file.

Then the file is transcoded to a *webm* file.

A.2 Audio

For audio handing we use *sox*[37]. First all *mp3* files are converted to *wav* format and padded, if necessary. All the resulting files are mixed together in a single *wav* file.

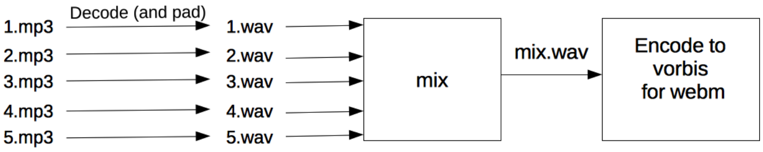


Figure 13: Post-processing for audio.

A.3 Merging

The resulting audio (a single *wav* file) and video (a single *webm* file) are combined using *ffmpeg*. The audio is automatically transcoded to *vorbis*, because it is the codec supported by *webm*[13].

A.4 Performance

The running time of the application depends on a variety of parameters: the number of participants, the chosen frame rate and resolution, the number of SPEAKER_CHANGED events.

The process was optimized by running certain tasks– the initial ”decoding”, and the section processing and concatenation– in parallel (in different operating system processes).

We observed that for a particular recording with 4 participants the running time, after all optimizations, was about 1.3 times longer than the length of the conference.

The process is not very efficient and performs at least 2 redundant transcodings. We found no other way to optimizing it without a major re-design, which we are planning to do in the future.

B The full contents of a metadata file

```
{
  "audio" : [
    { "ssrc" : 2473760775,
      "filename" : "2473760775.mp3",
      "type" : "RECORDING_STARTED",
      "instant" : 1401807483964,
      "mediaType" : "audio" },
```

[13] Although the specification is being extended to allow for Opus

```
  { "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"},
  ]
}
```

References

[1] J. Bankoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, and Y. Xu. VP8 Data Format and Decoding Guide, Internet Engineering Task Force Request for Comments (RFC) 5245, November 2011.

[2] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP), Internet Engineering Task Force Request for Comments (RFC) 3711, March 2004.

[3] J. Fischl, H. Tschofenig, and E. Rescorla. Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS), Internet Engineering Task Force Request for Comments (RFC) 5763, May 2010.

[4] Emil Ivov, Lyubomir Marinov, and Philipp Hancke. COnferences with LIghtweight BRIdging (COLIBRI), XMPP Standards Foundation XEP-0340, January 2014. http://xmpp.org/extensions/xep-0340.html.

[5] J. Lennox, E. Ivov, and E. Marocco. A Real-time Transport Protocol (RTP) Header Extension for Client-to-Mixer Audio Level Indication, Internet Engineering Task Force Request for Comments (RFC) 6464, December 2011.

[6] A. Li. RTP Payload Format for Generic Forward Error Correction, Internet Engineering Task Force Request for Comments (RFC) 5109, December 2007.

[7] Scott Ludwig, Joe Beda, Peter Saint-Andre, Robert McQueen, Sean Egan, and Joe Hildebrand. Jingle, XMPP Standards Foundation XEP-0166, December 2009. http://xmpp.org/extensions/xep-0166.html.

[8] Scott Ludwig, Peter Saint-Andre, Sean Egan, Robert McQueen, and Diana Cionoiu. Jingle RTP Sessions, XMPP Standards Foundation XEP-0167, December 2009. http://xmpp.org/extensions/xep-0167.html.

[9] D. Mills, U. Delaware, J. Martin, J. Burbank, and W. Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification, Internet Engineering Task Force Request for Comments (RFC) 5905, June 2010.

[10] Marcin Nagy, Varun Singh, Jörg Ott, and Lars Eggert. Congestion control using fec for conversational multimedia communication. *CoRR*, abs/1310.1582, 2013.

[11] J. Ott, S. Wenger, N. Sato, C. Burmeister, and J. Rey. Extended RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF), Internet Engineering Task Force Request for Comments (RFC) 4585, July 2006.

[12] C. Perkins, I. Kouvelas, O. Hodson, V. Hardman, M. Handley, J.C. Bolot, A. Vega-Garcia, and S. Fosse-Parisis. RTP Payload for Redundant Audio Data, Internet Engineering Task Force Request for Comments (RFC) 2198, September 1997.

[13] S. Pfeiffer. The Ogg Encapsulation Format Version 0, Internet Engineering Task Force Request for Comments (RFC) 3533, May 2003.

[14] E. Rescorla and N. Modadugu. Datagram Transport Layer Security, Internet Engineering Task Force Request for Comments (RFC) 4347, April 2006.

[15] J. Rey, D. Leon, A. Miyazaki, V. Varsa, and R. Hakenberg. RTP Retransmission Payload Format, Internet Engineering Task Force Request for Comments (RFC) 4588, July 2006.

[16] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols, Internet Engineering Task Force Request for Comments (RFC) 5245, April 2010.

[17] J. Rosenberg, H. Schulzrinne, and O. Levin. A Session Initiation Protocol (SIP) Event Package for Conference State, Internet Engineering Task Force Request for Comments (RFC) 4575, August 2006.

[18] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core, Internet Engineering Task Force Request for Comments (RFC) 6120, March 2011.

[19] P. Saint-Andre, S. Ibarra, and E. Ivov. Interworking between the Session Initiation Protocol (SIP) and the Extensible Messaging and Presence Protocol (XMPP): Media Sessions, Internet Engineering Task Force Internet Draft, March 2014. https://tools.ietf.org/html/draft-ietf-stox-media-04.

[20] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications, Internet Engineering Task Force Request for Comments (RFC) 3550, July 2003.

[21] T. Terriberry, R. Lee, and R. Giles. RTP Payload Format for Opus Speech and Audio Codec, Internet Engineering Task Force Internet Draft, February 2014. http://tools.ietf.org/html/draft-ietf-codec-oggopus-03.

[22] JM. Valin, K. Vos, and T. Terriberry. Definition of the Opus Audio Codec, Internet Engineering Task Force Request for Comments (RFC) 6716, September 2012.

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

[24] S. Wenger, U. Chandra, M. Westerlund, and B. Burman. Codec Control Messages in the RTP Audio-Visual Profile with Feedback (AVPF), Internet Engineering Task Force Request for Comments (RFC) 5104, February 2008.

[25] P. Westin, H. Lundin, M. Glover, J. Uberti, and F. Galligan. RTP Payload Format for VP8 Video, Internet Engineering Task Force Internet Draft, February 2014. http://tools.ietf.org/html/draft-ietf-payload-vp8-11.

[26] P. Zimmermann, A. Johnston, and J. Callas. ZRTP: Media Path Key Agreement for Unicast Secure RTP, Internet Engineering Task Force Request for Comments (RFC) 6189, April 2011.

[27] Additional IP Rights Grant (Patents). http://www.webmproject.org/license/additional/.

[28] FFmpeg. http://ffmpeg.org.

[29] Freedom for Media in Java. http://sourceforge.net/projects/fmj/.

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

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

[32] Jitsi Meet. https://jitsi.org/meet.

[33] Jitsi Recording Container. https://github.com/jitsi/jirecon.

[34] Matroska. http://matroska.org/technical/specs/index.html.

[35] non-bundled createAnswer uses more than one CNAME, WebRTC issue tracker. https://code.google.com/p/webrtc/issues/detail?id=3431.

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

[37] SoX Sound eXchange, the Swiss Army knife of audio manipulation. http://sox.sourceforge.net/sox.html.

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

[39] The webm project. http://www.webmproject.org/about/.

[40] Yuv pixel formats. http://www.fourcc.org/yuv.php#IYUV.