Linköping University | Department of Computer and Information Science
Master thesis, 30 ECTS | Datavetenskap
2018 | LIU-IDA/LITH-EX-A--18/049--SE

WebRTC-based Video Quality of Experience Evaluation of the Janus Streaming Plugin

– Integrating Video Door Systems and WebRTC-Supported Browsers

---

Videoutvärdering av Janus streamingmodul med fokus på WebRTC-baserad Quality of Experience

Raymond Leow

Examiner : Niklas Carlsson

External supervisor : Johan Åberg

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

The number of Information and Communication Technologies (ICTs) used in organizations is increasing. Without careful management, this trend can have sever consequences for organizations as it easily can fragment the business communication. A promising approach to manage these ICTs is to integrate them into a single medium so to ease the communication within and between organizations. The open source project Janus introduces new ways to communicate between legacy protocols and the real-time communication API *Web Real-Time Communication* (WebRTC). This master thesis addresses how this WebRTC-based implementation can communicate with an IP video door system. In particular, a minimal front-end solution has been implemented and evaluated. The implementation allows the user to make a call, get notified in the communication tool (both visually and auditory), have a two-way audio call, and see the video stream from the video door system. An objective WebRTC-based quality assessment, focusing on Quality of Experience (QoE), has been conducted on the implementation. As part of this assessment, we used the *WebRTC-internals* tool to collect receiver-relevant video attributes. A QoE-focused comparison to a light-weight Kurento client implementation was also performed. The results show that our Janus implementation and setting is feasible, aside from the periodic bandwidth drops, and outperforms Kurento in the defined QoE attributes.

Acknowledgments

First, I would like to thank the people at Briteback for giving me the opportunity to carry out my master thesis. Furthermore, thanks goes to Johan Åberg and Niklas Carlsson for helping me move the project into the right direction.

Finally, I would like thank my family for helping me carry out their wish.

Contents

List of Figures

List of Tables

Acronyms

API Application Programming Interface

AR Augmented Reality

HTTP Hypertext Transfer Protocol

ICE Interactive Connectivity Establishment

ICT Information and Communication Technology

IETF Internet Engineering Task Force

IP Internet Protocol

JSEP JavaScript Establishment Protocol

JSON JavaScript Object Notation

MAD Median of all Absolute Deviations

MCU Multipoint Conferencing Unit

MJPEG Motion JPEG

MOS Mean Opinion Score

NAT Network Address Translation

P2P Peer-To-Peer

PERC Privacy Enhanced RTP Conferencing

PLI Picture Loss Indication

PSNR Peak Signal-to-Noise Ratio

PSTN Public Switched Telephone Network

QoE Quality of Experience

QoS Quality of Service

REST Representational State Transfer

RPC Remote Procedure Call

RTC Real-Time Communication

RTMP Real-Time Messaging Protocol

RTP Real-Time Transport Protocol

RTSP Real Time Streaming Protocol

SDK Standard Development Kit

SDP Session Description Protocol

SFU Selective Forwarding Unit

SIP Session Initiation Protocol

SSRC Synchronization Source identifier

STUN Session Traversal Utilities for NAT

TURN Traversal Using Relays around NAT

URI Uniform Resource Identifier

VoIP Voice over Internet Protocol

W3C World Wide Web Consortium

WebRTC Web Real-Time Communication

1 Introduction

Business communication is established through a large spectrum of activities. One way to establish connections is through *Information and Communication Technologies* (ICTs). Examples of such are emails and microblogs [42]. With the introduction of social media, new methods to communicate within and between organizations have been popularized. Usage of such technologies has risen at an incredible speed [21]. It has been shown to boost positive attitude and improve employee engagement [11]. With social media added to the spectrum of activities, organizations have to choose between more ICTs than ever. Multiple ICTs are often chosen to fulfill their intended communication purposes. However, Yuan et al. argue that the combination of ICTs used affects the business communication quality [42]. Furthermore, they mention that when it comes to knowledge sharing in organizations, some choices could be worse than other. An incorrect choice of combination of ICTs could instead create "technological divides" among the users. It is better to use a single tool in which all forms of communication are integrated. Whether email or instant messaging is used, every employee can choose their own communication forms for organizational knowledge sharing as they see effective. Grandstream, an IP (Internet Protocol) communication solution manufacturer, extended their portfolio with a new way to communicate over IP. The video door system GDS3710 offers integration over Hypertext Transfer Protocol (HTTP) and Session Initiation Protocol (SIP) for integration to any IP-based solution [26]. This opens up even more ways to communicate from a business context.

In a normal scenario, video door systems are either used to call phones as any other phone. The video is in most cases streamed through a third-party solution or the manufacturer's own software. This adds additional ICTs which the business needs to use. To simplify to the users, the video door system could be integrated into the same tool for seamless communication between the door and other ICTs.

1.1 Aim

This thesis aims to demonstrate how an IP video door system can be integrated to another ICT. We also investigate the acceptance of the different cross-protocol communication used by the Janus gateway, in terms of pre-defined *Quality of Experience (QoE)* related factors, which tries to capture "the degree of delight or annoyance of the user of an application or service"

[7]. By comparing to another WebRTC server solution, this thesis will address the benefits and limitations of the integration from a user experience perspective.

1.2 Research Questions

Motivated by our high-level aims, the thesis will address the following questions:

1. How can the video and audio of the Grandstream video door system be integrated in to the Briteback communication tool with the help of the Janus gateway and API?
2. Is it feasible for such integration to provide acceptable QoE as defined by the QoE performances presented by Ammar et al. [4]?
3. How does the implementation compare to the Kurento media server [23] from a QoE standpoint?

1.3 Contributions

The main contribution of this thesis is to integrate a Grandstream IP video door system with a WebRTC-supported browser. Motivated by WebRTC's flexibility to support future-proof tools, in this thesis the *Janus WebRTC gateway* is used to provide such service. It has the capability to bridge together video streaming, SIP, WebRTC, and more. With the gateway, we integrate the two ICTs and enable cross-platform communication between the two. To investigate the acceptance of the integration, an evaluation on the Janus gateway API is performed. More specifically, we provide with measurements of the API based on certain QoE factors. The QoE factors in WebRTC-based video communication set practical constraints to the integration. Bad user experiences between the integrated organization ICTs can be avoided by fulfilling specific QoE factor criterias.

1.4 Related Work

Evaluations on QoE in real-time video is conducted in a multitude of papers.

Skowronek and Raake brings a conceptual framework regarding quality in humans and their processing of quality in a multiparty conferencing context [35]. Three categories (levels) of *influence factors* affecting video-mediated multi-party conversations are discussed.

The first is the system level, e.g. application-specific issues and network-related conditions. De Cicco et al. goes into the extent of which Skype Video throttles its sending rate in order to find a solution which preserves a stable network traffic [9]. In a study conducted by Schimitt et al., a QoE testbed for group video communication is implemented [34]. The aim with the component is to monitor QoE in real-time and to dynamically adjust network and communication parameters, targeting multi-party video-mediated communication. Others have shown that network imposed rate caps can be beneficial to the QoE of competing video on demand streaming players [19].

The second is the human level which for example involves a person's role in the conversation and previous experiences to video conversations.

The third level is the context, e.g. environmental settings and location of the conversation. An example of such study is the investigation by Vucic and Skorin-Kapov of QoE for video conferencing in mobile context [39]. Specifically, subjective studies were conducted with WebRTC-based applications over Wi-Fi under different conditions.

Alternatively to the framework produced by Skowronek and Raake [35], Perkis et al. provide a model which instead composes of a *technology* category and an *user* category [29]. Technology contains all *system*-related attributes which are measurable, whereas the user category contains *human*- and *context*-related attributes, e.g. understanding, impressions and expectations.

1.5 Limitations and System Requirements

A server implemented with *Asterisk* [12] is provided by the customer and is not under our control. How this server is set up and what configurations it has is not public information and cannot be specified in this thesis. However, at a high level, the Asterisk server acts as an ordinary SIP server and the gateway is deployed according to the descriptions of its deployment documentation.

A complete QoE-based comparison between Janus and Kurento is beyond the scope of this thesis. A considerable amount of scenarios, e.g. different network settings, implementations of different design patterns, comparisons in environmental situations, are needed to fully cover the comparison. Therefore, we have defined a list of system requirements in which the implementation must fulfill, as shown below. Both the implementation with the Janus library and the *comparable* implementation of the Kurento library must be able to fulfill the functional system requirements. Additionally, we have reduced the scope of test scenarios to a single one. Data is gathered from the test scenario and a comparison between Janus and Kurento is made. We further describe the test scenario assessment in the Method chapter. In addition, testing the acceptance of the integration is only determined for this specific scenario.

Due to a specific behaviour in the Janus gateway, as we refer to the *periodic bandwidth drops*, we have decided to use the *location-scale model* (more details in the Method chapter). The selection of the removal has some limitations. Applying the model is not done without making biased assumptions. One example is that without the periodic bandwidth drops the data gathered from Janus solution is assumed to always have non-zero values. However, we further discuss and motivate the reason of this method selection in the Discussion chapter.

We next list the system requirements outlining how the integration must work.

Non-functional system requirements

- **Requirement: 1.1**
  Description: The video door system must communicate with the Briteback communication tool through SIP.
- **Requirement: 1.2**
  Description: The implementation must use a Janus gateway.
- **Requirement: 1.3**
  Description: The Briteback communication tool must use the Janus API to access the Janus gateway and the video door system server.
- **Requirement: 1.4**
  Description: The comparable implementation must use a Kurento media server.
- **Requirement: 1.5**
  Description: The comparable version of the Briteback communication tool must use the Kurento API to access the Kurento media server and the video door system server.

Functional system requirements

The Grandstream video door system also goes under the model name *GDS3710*. The *caller* refers to the user of the GDS3710 video door system and *callee* refers to the user of the Briteback communication tool.

- **Requirement: 2.1**
  Description: When a user makes a call from the GDS3710 to the Briteback communication tool, the communication tool must be notified auditorily and visually.
  Rationale: The callee needs to able to know of the caller's presence.

- **Requirement: 2.2**
  Description: When the communication tool notifies of an incoming call, a video feed is shown. The video stream comes from the video door system server.
  Rationale: The user needs to be able to see the video feed without switching ICT.

- **Requirement: 2.3**
  Description: When the communication tool notifies of an incoming call, the video stream must be shown in a *toast* (i.e. a smaller popup showing any temporary information to the user). It must also be able to provide with time-based activities. An example is to give the callee the opportunity to answer a call. The toast must disappear after a short period.
  Rationale: To fully integrate the GDS3710 as a part of the communication tool, it needs to behave as any other peer calling the communication tool. Toasts are key components in the functionalities of the communication tool.

- **Requirement: 2.4**
  Description: When the communication tool notifies of an incoming call, the callee must be able to accept and decline the call.
  Rationale: The user needs to be able to select how they want to interact with the caller.

- **Requirement: 2.5**
  Description: When an incoming call is accepted, declined or ignored any visual or auditory notifications stop.
  Rationale: This is one of the functionalities which gives the user the opportunity to return to any previous system states.

- **Requirement: 2.6**
  Description: When an incoming call is accepted, the callee must be able to see an enlargened version of the video stream.
  Rationale: The callee must be able to interact with the video door system as if it is a video call with any other video-supported device.

- **Requirement: 2.7**
  Description: When the video door system is communicating with the Briteback communication tool, the audio goes both ways whereas the video only goes from the video door system to the Briteback communication tool.
  Rationale: The video door system and communication tool need to be able to auditorily communicate with each other, despite the one-way video channel.

1.6 Project Context

This thesis is conducted in conjunction with an assignment given by Briteback. The company is currently developing a communication tool with the same name. This tool gives companies the opportunity to communicate and collaborate across multiple communication platforms. Currently the tool uses a videoroom plugin and a SIP-based plugin to the Janus gateway. An extension, using Grandstream devices, is needed to integrate every product directly into the communication tool.

According to the developers of Briteback, they have a vision that the communication tool shall be able to connect multiple end-users across multiple platforms and protocols. Grandstream is one of the brands which offers IP systems from surveillance to conference video systems over IP. An addition with this brand further widens the possible variety of ICTs which can be integrated together into a single tool.

An implementation with the usage of Janus is preferred. A Janus gateway with it is deployed, which current version of the application is using.

1.7 Thesis Outline

In this section, we show a thesis overview and brief descriptions of the upcoming chapters.

- **Chapter 2: Background**
  In chapter two, key concept and metric definitions are identified. Furthermore, methods and tools used in similar studies are presented.
- **Chapter 3: Method**
  In chapter three, we identify the method and tools used to address the questions of this thesis.
- **Chapter 4: System Design**
  In chapter four, system descriptions and architectures are provided.
- **Chapter 5: Performance Results**
  Relevant data as a result of the evaluations are presented and observations of such.
- **Chapter 6: Discussion**
  In chapter five, discussions of the evaluations, comparisons and the system from the results are made. We also reconsider the choice of method and the thesis work in a wider context.
- **Chapter 7: Conclusion**
  In chapter six, we conclude the works of results, discussions of this thesis, and address possible future work.

2 Background

More information is needed to fully specify the contexts of the problem and methods used to address the questions. For example, the GDS3710 has its advantages and limitations. It has support for functions such as IP surveillance camera and IP intercom [26]. The system also supports SIP/VoIP, offering 2-way audio and video streaming to SIP endpoints. However, it lacks support for WebRTC. Therefore, we need to address some bridging solutions to WebRTC in order to establish, manage, and end communications to it. Furthermore, a means of addressing acceptance in a video QoE context is needed.

2.1 SIP

SIP is a similar protocol to HTTP in the way sessions are set up and torn down [17]. *Session Description Protocol* (SDP) is used to describe the call (audio, video, size of packets, etc.). Instead of locating an IP address, a *URI* is identified. This can be an email address, phone number or any other type of nickname. SIP may also be used to send notifications (for example to notify others that you are online or that someone has disconnected) and instant text messaging functionality.

As earlier mentioned, the client-server model SIP is similar to the HTTP in the request-respond exchange. The clients send the requests and the servers accept them. Then, the servers execute the requested methods and send responds. The most important request methods are defined as

- **REGISTER** is used to register information with a server.
- **INVITE** initiates a SIP dialog, indicating to others that a call is active.
- **ACK** is used to acknowledge INVITE requests.
- **CANCEL** is used to cancel a request.
- **BYE** is used to tear down a session.
- **OPTIONS** queries the capabilities of an endpoint.

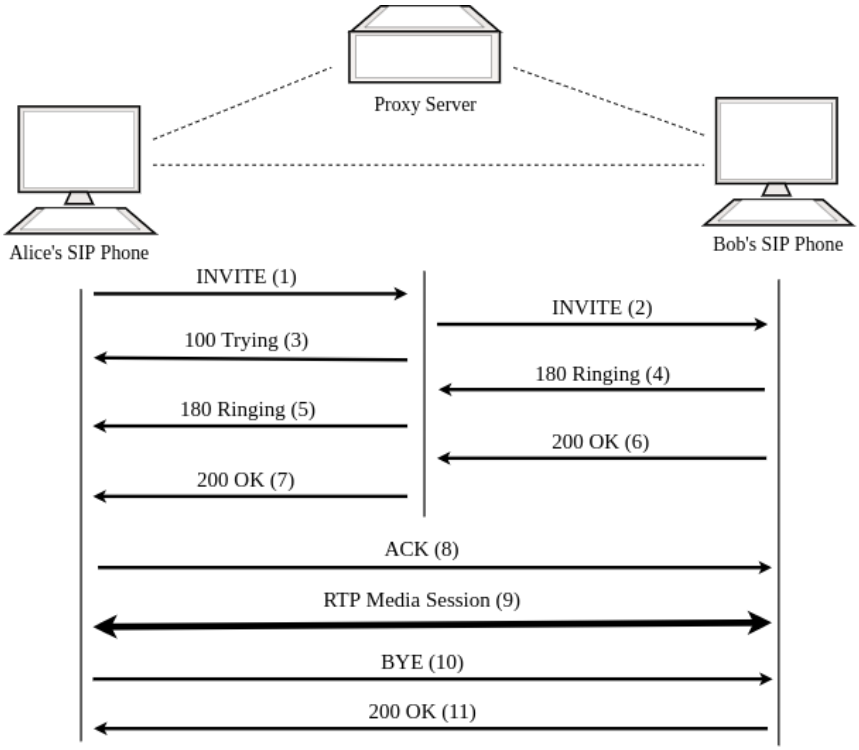


Figure 2.1: SIP call setup with a single proxy server

To easier explain the SIP operations, Goode brings an example [17]. Consider the situation where Alice calls Bob. In this simple scenario, they are in the same building and therefore have the same SIP proxy server. Figure 2.1 shows how the request headers could look like this example. First, Alice has to locate Bob's SIP phone. This is first handled by going through the SIP proxy server. Alice dials Bob's phone number (111-22222). Her SIP phone converts this number into a SIP URI (e.g. sip:111-22222@ab.com) and sends an INVITE to the SIP proxy server. After that Alice has sent the INVITE request to the proxy server, it sends *100 Trying* response back to Alice's SIP phone. This means that the proxy will attempt to route the INVITE request to Bob's SIP phone. Then the proxy server adds its own IP address to the INVITE message and forwards it to Bob. Once Bob receives the INVITE, he gets a notification and is able to respond by sending *180 Ringing* back to Alice through the proxy server. Once the response goes through the proxy server, it can see the destination of the response by looking at the *Via* header. It removes its own IP address and forwards it to Alice's SIP phone. Alice receives a *180 Ringing* response. When Bob activates the speakerphone, the SIP sends an *200 OK* message with the purpose of showing that Bob has answered the phone call. The *200 OK* message contains descriptions the media and session type, informing the other endpoint of its capabilities in the contexts of SIP. Once Alice receives the message, her SIP phone sends an *ACK* directly to Bob's endpoint and a *RTP media session* is established between the two. The two phones are exchanging actual voice packets directly. No information about the proxy server or SIP request methods are needed during this exchange. When Alice wants to end the call, her SIP phone sends a *BYE* directly to Bob. His SIP phone then responds with *200 OK*. This message terminates the call and the *RTP media session* between them.

2.2 WebRTC

The Internet Engineering Task Force and the World Wide Web Consortium had a goal of standardizing cross-platform and cross-device communication in the web browsing model [2].

Web Real-Time Communication (WebRTC) is the resulting standard made for exchanges of real-time media across different types of browsers, applications, or devices.

It is widely used in applications, such as WhatsApp and Facebook Messenger, as described by Dutton [13]. Furthermore, WebRTC uses in general three APIs. First, the `MediaStream` (`getUserMedia()`) API is used to synchronize media (RTP) streams. The audio and video tracks gathered from devices such as microphones and cameras are examples of such synchronized media streams. A `MediaStream` can be passed on to a video element (e.g. a HTML tag) or `RTCPeerConnection`. The `RTCPeerConnection` is used as channels to exchange data between peers, but also for handling control messages (i.e.. *signaling*). The signaling part itself is not a part of WebRTC nor `RTCPeerConnection`. Signaling is used to:

- Initialize and close WebRTC sessions and report possible errors
- Tell the world what your IP address is
- Exchange the capabilities of my browser, e.g. what codecs and resolutions it can handle

In addition to audio and video, WebRTC has the capability to communicate with other types of media with the help of `RTCDataChannel`. This API offers packet exchanges of arbitrary data. Examples of such are file transfers and real-time text chats.

Amirante et al. describe the conceptual architecture of WebRTC [2]. They outline the similarities between the WebRTC architecture and SIP Trapezoid, as illustrated in Figure 2.2. In their example, the browsers may run on different web applications and downloaded from different web servers. Like SIP, messages are sent back and forth to initiate and tear down sessions. The signaling between the client and the servers are implementation-specific for that web application. A `PeerConnection` is used to transport media between the peers, representing an unique connection between the two. The exchange of communication is handled by the client, with the help of for example `XMLHttpRequest` or `WebSocket`. Any standard signaling protocol may be used for the communication between the two web servers.

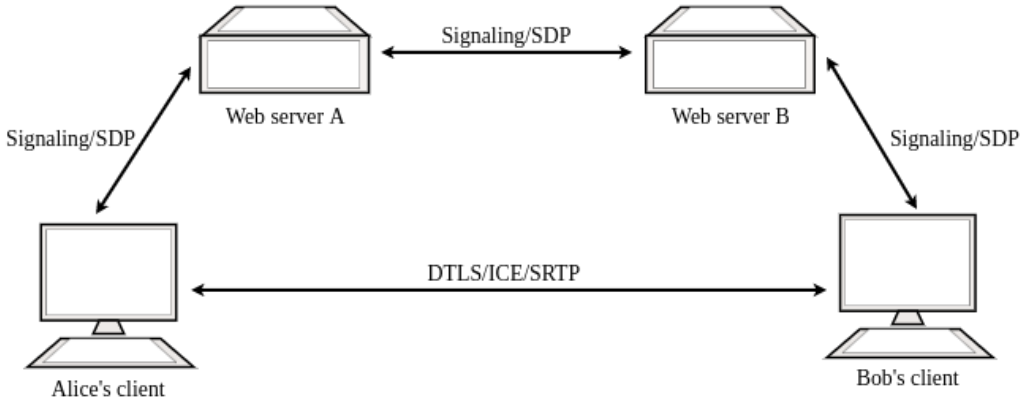


Figure 2.2: WebRTC trapezoid

Janus

Some WebRTC-based applications offer more than simple P2P connections. PSTN gateways and conferencing servers are examples of such. However, the standard WebRTC library does not suffice with this kind of support. By extending the current library, we may implement bridges in which WebRTC is able to communicate with other streaming protocols (e.g. SIP

and RTMP). Amirante et al. brings the design and implementation of *Janus* into the light [2]. This solution is an open source WebRTC gateway that enables communication between WebRTC endpoints and other platforms not supported by WebRTC. In general, it is a complex task to connect WebRTC endpoints with different *legacy* infrastructures (i.e. not based on the IP protocol). Therefore, a WebRTC gateway is motivated.

The Janus gateway is described as a *general purpose* component considering that its architecture is modular. This gives the possibility to design and attach modules with specific features to the gateway. Amirante et al. motivate the usage of general protocols, using *control packages* to establish the general communication and some extra packages to provide the more specific, modular communication. Specifically, the Janus architecture consists of two parts; the core which handles high level communication between the users and the server-side plugins which handles the modular functionality.

The authors further describe that the core has three main tasks. First, with the help of a REST-ful API, it has the control of the management of the application sessions. Second, it takes care of the whole WebRTC process (i.e. establishing, managing and tears down connections). Lastly, it attaches users to the specific plugins, allowing the users to exchange data and media. This means that the plugins are the bridge between the legacy architecture and WebRTC. An overview of the Janus gateway can be seen in Figure 2.3. The PeerConnections and usage of plugins may vary from implementation to implementation. An example is the video call plugin which relays packets, transporting them from one user to others. Some of the existing plugins compatible with Janus are the streaming plugin (relaying video and audio to external WebRTC users), a SIP plugin (bridging the SIP infrastructure with WebRTC), and video multipoint control unit (MCU) plugin (enabling for example video conferences and screen sharing).

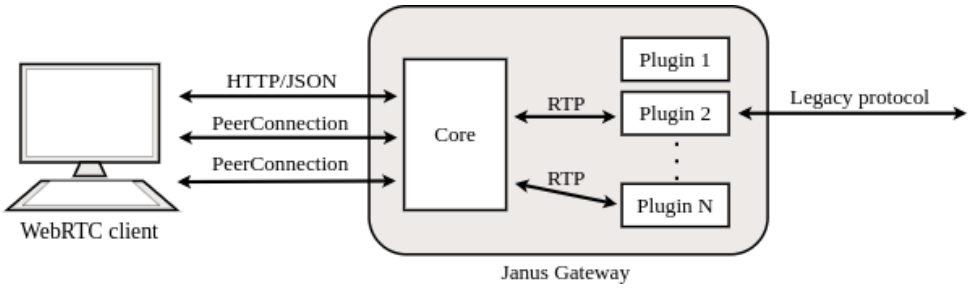


Figure 2.3: Janus gateway overview

In addition to the plugin-specific functionality Janus provides, the Janus core has support for JavaScript Session Establishment Protocol (JSEP) and SDP. This gives the opportunity to setup and manage multimedia sessions with WebRTC users. Also, the support for Datagram Transport Layer Security (DTLS) extensions for SRTP and Trickle ICE allows for more safe respectively effective multimedia channel establishments. Most of the time, the gateway uses and manipulates Real-time Transport Protocol (RTP) and Real-time Transport Control Protocol (RTCP) flows.

From a WebRTC user's perspective, the Janus gateway can be reached through a JSON-based ad-hoc protocol with their REST-ful API. This means that the sessions between the WebRTC user and the gateway and plugin *handles* (connections to the specific plugin) to be general. Whenever a WebRTC user is interested in a feature of Janus, a connection to the gateway is established and a session is initialized. The gateway web server creates an endpoint which awaits for HTTP-based events. Through the API, one or multiple handles can be created and attached to the specific plugin. An interaction with the plugin is possible, typically in the form of PeerConnection. When the WebRTC user is finished with the Janus feature, they can close the handle and its associated PeerConnection and destroy the session. The general concept can be described as illustrated in Figure 2.4.

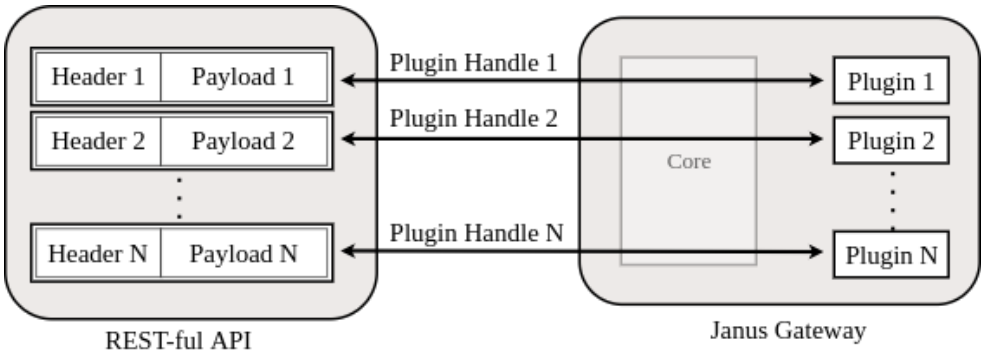


Figure 2.4: The REST-ful API creating plugin handles to the associated Janus gateway plugin

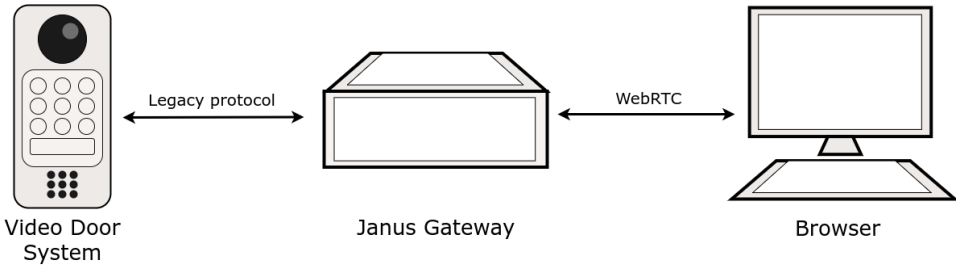


Figure 2.5: Janus gateway can be used so that IP cameras can use their protocol and the browser can use WebRTC

More practically, if we want to communicate with a video door system with a legacy protocol we first configure and start the gateway. It is possible to deploy the gateway in the same network as the the browser. Since the video door system does not have a screen we only care for one-way video streaming. The videoroom plugin is deemed excessive, because it expects a two-way video communication. Instead the *streaming* plugin is used. The door system only supports video via HTTP or RTSP. With the API, JavaScript can be used to perform operations on the gateway to handle the signaling and transcoding.

In Miniero's Janus streaming reference, the *streaming* API offers different types of requests which are both synchronous and asynchronous [25]. The request types `list`, `create`, `destroy`, `recording`, `edit`, `enable` and `disable` are synchronous. When these requests are used, the response will also be in Janus API object notation. `list` gives a list of available streams. `create` and `destroy` creates respectively destroys a mountpoint. `recording` handles whether the RTP stream is live or not. `enable` and `disable` enables respectively disables a mountpoint. This is similar to `create` and `destroy`, but they assume a mountpoint exists and it will not be destroyed. `edit` configures mountpoint settings, such as PIN code. In JavaScript, we tell the plugin handle to send messages. The general syntax is very similar to any HTTP request, where we send the request to the server and make some operations in the callback function with the response.

```
pluginHandle.send(request, callbackFunction(response));
```

To exemplify, if we want to utilize `list` to receive the available streaming mountpoints from the Janus API, a message is sent with the `list` request:

```
function getList(pluginHandle) {
  const listRequest = {"request": "list"};
  pluginHandle.send(listRequest, function(response) { /* Handle response */});
}
```

If the request was successful, we get an array of available mountpoints. Each mountpoint has a unique type. `rtp` means that the stream is sent via RTP; `live` is when a local file is streamed live; `ondemand` is when a local file is streamed on-demand; `rtsp` is when the stream is coming from and external RTSP feed. The syntax for the result goes in the form of:

```
{
  "streaming": "list",
  "list": [
    {
      "id": "<ID of the first mountpoint>",
      "description": "<Description of the first mountpoint>",
      "type": "<Type of the first mountpoint>",
      ...
    },
    {
      "id": "<ID of the second mountpoint>",
      "description": "<Description of the second mountpoint>",
      "type": "<Type of the second mountpoint>",
      ...
    }
    ...
  ]
}
```

`watch`, `start`, `pause`, `switch` and `stop` are asynchronous.

- `watch` prepares the playout of the available stream.
- `start` starts the playout.
- `pause` pauses the playout without tearing down the session.
- `switch` switches to a different mountpoint.
- `stop` stops the PeerConnection and tears it down.

The request being sent and response received for `start`, `pause` and `stop` are straight forward. We send a simple request and receive a simple response. Watching and switching the mountpoint has additional parameters:

```
{
  "request": "start"
}

{
  "response": "starting"
}

{
  "request": "switch",
  "id": "<ID of the mountpoint to switch to>"
}

{
  "request": "watch",
  "id": "<ID to subscribe to>",
  "pin": "<PIN code needed to access the mountpoint, if configured>",
  "offer_audio": "<Set whether or not audio should be available through the
      mountpoint>",
  "offer_video": "<Set whether or not video should be available through the
      mountpoint>",
  "offer_data": "<Set whether or not DataChannels should be available through the
      mountpoint>"
}
```

Because the requests are asynchronous, the Janus API uses callback functions which acts as event handlers for receiving these messages from the gateway. All messages sent from the gateway is handled by `onmessage`.

```
{
  ...
  onmessage: function(message, jsep) {
    const result = message["result"];
    if (result) {
      const status = message["status"];
      switch (status) {
        case 'starting': // Handle when stream is starting
          break;
        case 'started': // Handle when stream has started
          break;
        case 'stopped': // Handle when stream has stopped
          break;
        ...
      }
    }
  }
  ...
}
```

Kurento Media Server

An alternative to the Janus gateway is the Kurento Media Server. López et al. state that this WebRTC media middleware is a more extensive and more modular server than Janus [23]. Furthermore, the authors claim that the modules can be interconnected as opposed to the modular solution of Janus. In addition, it offers more modules, primarily in the field of media processing. For example, the media may be filtered for augmented reality or computer vision purposes. Kinect-like games or incident reports (by face or crowd detection) are examples which are possible with WebRTC.

As with the Janus, Kurento have its own APIs for Java and JavaScript application developers. It also offers communication using Kurento Protocol, which is based on WebSocket and JSON-RPC. In total, Kurento Client API offers two solutions as shown in Figure 2.6.

The modular architecture of Kurento is based on *Media Elements*, the building blocks of Kurento Client APIs. A Media Element represents either an endpoint, a filter, or a hub. The endpoints are the components which communicate the media with others. For example, `WebRtcEndpoint` sends and receives WebRTC media streams. A filter processes the media, e.g. applying AR capability to the video streams. Hubs connect the communication and enable group communication. An example of such build is illustrated in Figure 2.7. Based on topology in the figure, we see that the application takes WebRTC media from the browser, records it to the file system, and uses that media to play through the GUI. Kurento uses `WebRtcPeer` to handle client-side WebRTC streams. However, standard WebRTC API elements (e.g. `getUserMedia` and `RTCPeerConnection`) may be used for communication with the `WebRtcEndpoint`.

In the creation of a Kurento application, the application developer can conceptually take Media Elements and connect them to any arbitrary topology as they see fit. All Media Elements have the `connect` primitive in which the connectivity between the elements is accomplished through. This primitive is invoked on the source element and takes the sink as argument. The syntax would be:

```
sourceMediaElement.connect(sinkMediaElement)
```

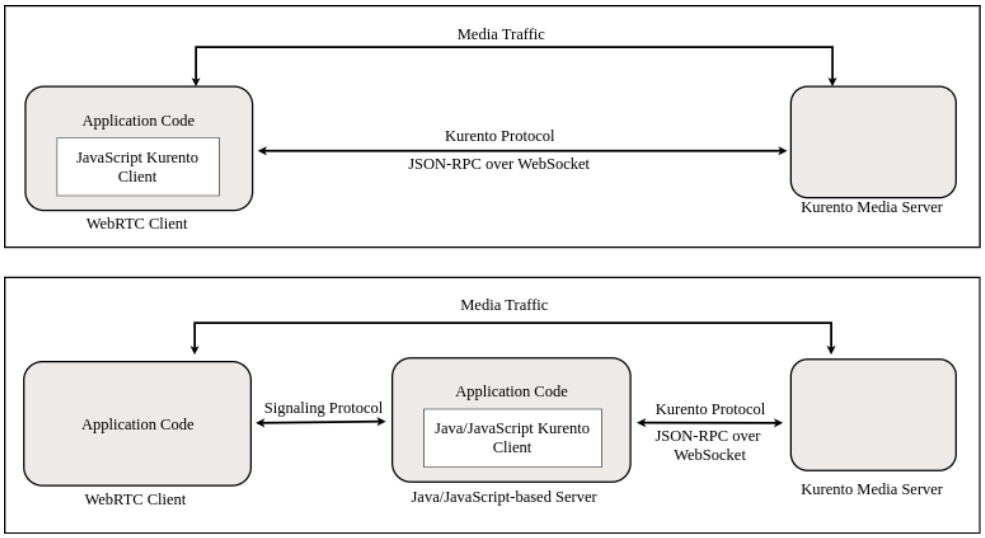


Figure 2.6: Kurento Client architectures. With the upper alternative, the user directly communicates with the Kurento Media Server with the usage of Kurento Protocol. The lower alternative is the usage of the Kurento Client SDK and API, which either can help the application developer to implement a Java client or a JavaScript client.

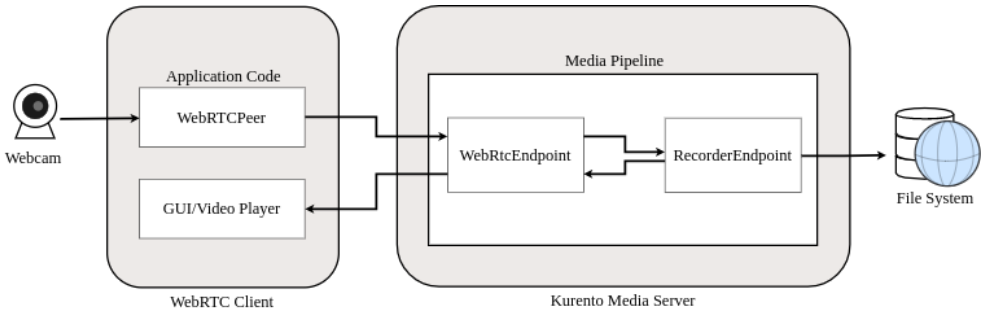


Figure 2.7: Illustration of a pipeline of Media Elements in a Kurento application

Jitsi and Jitsi Meet

Jitsi is another open source multi-platform solution, which offers video, audio and instant messaging. It is built on various Java libraries and is not web-oriented. This means that the user has to download specific software in order to run it. A *Selective Forwarding Unit (SFU)*, which forwards streams to others in a video conference call, is needed to support WebRTC [6]. Paščinski et al. further describes the Jitsi videobridge component, which enables the application used to exchange real-time multimedia streams using WebRTC. The WebRTC-specific application, Jitsi Meet, uses a centralized architecture [28]. This means that the media processing component does not process the payload (e.g. transcoding video and audio). The main focus of Jitsi Meet is video conferencing. López et al. state that the SFU lacks support for extension and combinations with other types of media processing modules [23]. It is also not modular, meaning that application development in other purposes than video conferences are discouraged.

Janus Stress Test

Amirante et al. also developed an automated stressing tool for analyzing WebRTC-based servers, in terms of performance and scalability [3]. The tool *Jattack* focuses on stress testing

server-side components by quickly generating *PeerConnections*. The implementation is a modification on the Janus gateway. No signaling is made on the tool. It only gives and reallocates commands to the corresponding Janus component. For example, if a client wants to register, then the client connects to the controller first and forward the registration handling to the controller. This means that signaling is up to the user, allowing the establishment of a more realistic test environment. The architecture of the tool together with the Janus gateway is shown in Figure 2.8. First, the controller handles the registration of the Jattack testing tool. Media sources are generated on request from the user. Janus is contacted and create sessions (PeerConnections). If the Jattack instances request for media, then the media is attached to the PeerConnection.

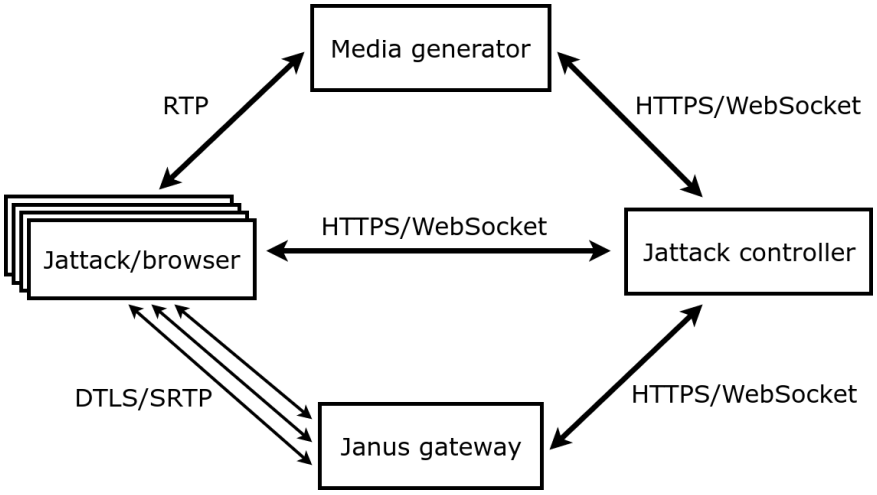


Figure 2.8: The Jattack architecture

A stress test on the VideoRoom plugin was then performed by Amirante et al. [3] The testbed was a 8 Intel Core i7-4770S CPUs @ 3.10GHz and 16 GB of RAM with Ubuntu 16.04 OS and was set up with the *Docker* containers. In the stress tests, 1000 PeerConnections were maintained by the Jattack test tool and Janus gateway. The results showed that a decrease in quality started to show at around 800 PeerConnections. Amirante et al. speculate that the cause of this is that the media router is unable to enqueue packets for delivery in time. Another speculation is that Janus creates a new thread to transmit for each receiver.

Alternatives to the Jattack test tool are mentioned by Amirante et al. Those are the Kurento Media Server, Jitsi Hammer, and Selenium Framework. The Kurento Media Server is deemed to require an extensive amount of resources. The browsers are expected to be implemented on server machines to able to test the server-side components. The Jitsi Hammer only works on the Jitsi Videobridge. However, the Selenium framework is a valid alternative as it makes it possible to fully simulate and control a browser's behaviour. It also has the capability to create WebRTC PeerConnections in an automated manner.

2.3 GDS3710

The video door system has several solutions for the user to access its functionalities. The four primary solutions are through SIP, RTSP, with the usage of the HTTP API, and Grandstream's internal HTTP client software.

The *SIP solution* offers communication with both audio and video. SIP servers (e.g. proxies and registrars) has to be set up by the user in order to handle SIP requests. This means that the capabilities of this solution lies in the capabilities of the servers. For example, some are able to transport audio but not video.

To get audio and video through *RTSP*, the application developer only needs to access the feed from the RTSP server hosted by the video door system. In comparison to SIP, we assume that it is a one-way video communication channel. No two-way video request exchange is needed.

With the *GDS3710 HTTP API*, the user has the capability to change and get properties of the video door system. Typical HTTP requests are sent to the system and HTTP responses are received from it.

Grandstream offers its own *internal software* where the user can see the video stream on the video door system's allocated IP address. This IP address is local, meaning that the user of the Grandstream's internal software has to be in the local network to access it. Grandstream's own solution is meant for those who do not want to implement their own solution.

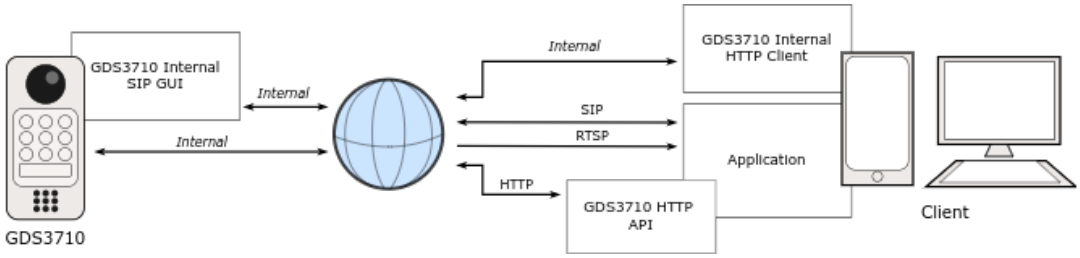


Figure 2.9: Four ways to access and/or change the properties of the GDS3710, either with SIP, HTTP, RTSP and/or the internal software. The channels of the figure marked as *Internal* are handled solely by GDS3710 and cannot be modified by external operations.

Figure 2.9 provides with an overview of the solutions. Observe that we have no control of the internal software. We are not able to access or modify the resources as the GDS3710 client and server is isolated. Furthermore, no official documentation of the behaviours of its client nor server is available at the moment.

GDS3710 Internal Software

As the resources are accessible and manageable through a browser, the JavaScript code can be analyzed with the help of the browser developer tool. The GDS3710 software is Grandstream's own JavaScript client implementation. It also extends and makes use of its own HTTP API. Normally, the HTTP API does not explicitly offer support for receiving video streaming. However, the internal software gives the user a video stream by repeatedly requesting for snapshots from the GDS3710 server. It then replaces an image element with the new snapshot. Same as with the usage of HTTP API, the internal software begins with authorization to the GDS3710 server. It retrieves, handles and sends back the challenge code. Then, cookies are received as well as an approved response. With the cookies, the internal software calls the `stream()` function. As shown in the simplified example below, it sets the name of the HTML element of a camera picture. It then repeatedly calls `freshPic()` with some delay, setting a new picture to this HTML element. To summarize, the internal software "streams" by sending requests for a snapshot from the camera every 500 ms and replaces the image HTML element.

```
var isStart = 1;
var page_flag = 0;

function holdUp()
{
  setTimeout("freshPic()", 500);
}

function freshPic()
{
  var trimmedPath = new Array();
  trimmedPath = currentPath.split("?");
  document.campic.src = trimmedPath[0] + "?" + Math.random();

  if (isStart) {
    holdUp();
  }
}

function stream()
{
  url_protocol = location.protocol;
  url_host = location.hostname;
  url_port = location.port;
  var path = location.pathname;
  url_path = path.split("/")[1];
  currentPath = url_protocol + "//" + url_host + "/" + url_path + "/snapshot/view.
      jpg";

  document.campic.src = currentPath + "?"  + Math.random();

  if (0 == page_flag)
  {
    holdUp();
  }
}
```

2.4 Quality of Experience for Real-Time Video

Quality of Experience is described as a rating of the performance of a service and how well it performs from a user's perspective [37]. Venkataraman and Chatterjee further claim that there is a gap between expected and delivered quality in Internet-based QoE services. Before the QoE was brought to the light, Internet QoS was a main focus in streaming services. However, QoS-based quality assessment have shown to be inaccurate when it comes to evaluate from a user's perspective. It is deemed to not be applicable in video quality evaluation [31, 38].

To understand why QoS-based mechanisms are as not representative for quality control, Venkataraman and Chatterjee argue the following example scenario. Consider Figure 2.10 as the expected playout at the destination. However, at times the playout is instead shown as either 2.11 or 2.12. If any human was asked which option of the two they would prefer, the former choice would be made. However, if PSNR (which is the most common QoS-based quality control) would be used, both options would be equally rated.

QoE has been a "driving factor" in customer satisfaction evaluation in domains other than video quality. Retail, food service and customer support are some of the examples.

An user's QoE depend on a variety of factors. Some of the *measurable* factors Venkataraman and Chatterjee classify are:

- **Media quality**: This is described as the most important metric as it describes how good the quality of the video is to the user. The user ideally wants high-quality audio and video synchronized with each other. Metrics relevant for this parameter are delays,

packet losses and delay jitter. These metrics affect blockiness (block artifacts [18]), blurriness and blackouts (as illustrated in Figure 2.12).

- **Network loss**: A miss of information which affects the media playout. The effect on the user may vary depending on the loss. Some losses may be negligible from an user's perspective.
- **Network jitter**: Jitter is described as "the variance in packet arrival times at the destination". To prevent packets to arrive out of order, a *jitter playout buffer* is implemented on the receiver side. However, if the jitter is too high it may go beyond the buffer's threshold. Consequently, the receiver could increase the size of the jitter playout buffer. However, this affects if the user would change the channel. Changing a channel means bigger rebuffer delays if this buffer would be larger.

Figure 2.10: Video under ideal circumstances

Figure 2.11: Video with block artifacts

Figure 2.12: Video during blackout

Quality of Experience Assessment

García et al. state that the Quality of Experience is gaining popularity in terms of media transmissions and services [16]. In contrast to the term *Quality of Service (QoS)*, QoE highlights the user's experiences and sets constraints on system and service designs from the user's perspective. The authors claim that QoE metrics is classified using two methods.

The first quality measurement method is considered subjective and not as applicable in production. A *Mean Opinion Score (MOS)* is estimated. The testers judge the quality of a system and set a grade on that quality from a score of 1 (bad quality) to 5 (excellent quality). This score is in most cases estimated in audio listening test environments.

The second, more objective method can be classified into five additional categories:

- **Media-layer models**: The QoE is calculated from the speech and video signal.
- **Parametric packet-layer models**: The QoE is predicted by reading the packet header information. Media signals are ignored.
- **Parametric planning models**: The QoE is predicted based on *quality planning parameters* for the networks.
- **Bit-stream-layer models**: The QoE is measured by the usage of encoded bit-stream and packet-layer information.
- **Hybrid models**: The QoE is measured using a combination of earlier mentioned methods.

Quality of Experience Factors in WebRTC-based Communication

The Quality of Experience is a fragile aspect in real-time communication. A bad experience from one user could possibly give a negative effect to others, despite the others having an otherwise ideal QoE. Ammar et al. identify and measure performance metrics which may affect a user's QoE [4].

Recall that a `RTCPeerConnection` is used for transmissions of audio, video, and data packets. A `RTCDataChannel` is established to exchange packets between two browsers. Track is described as either audio, video, or screen sharing inside a `RTCDataChannel`. Ammar et al. make comparisons of the WebRTC statistics provided by W3C [1]. The statistics are based on the following terms (which are referred to as *stats*):

- **RTP Stream Stats:** Statistics of the RTP stream. This includes the codec used, inbound and outbound RTP (packets received/sent, bytes received and resent, packets lost, jitter, round trip time, and target bit rate)
- **Media Stream Stats:** Statistics of the stream property, track identity, and media stream track. This includes audio-stream performance (such as audio level and echo return loss) and video-stream performance and statistics (including frame width and height, frames per second, frames received and sent, frames decoded, dropped, and corrupted)
- **Data Channel Stats:** Statistics used for data channels (including bytes received and sent, messages received and sent over the API, and the protocol used)

Despite not explicitly reflect on the packet transmission performance, ICE Candidate statistics may provide information which can affect the QoE factors. One example is the information from TURN or STUN server which can tell about the end-to-end delay.

One way to gather statistics is with the WebRTC-based functionality tool *WebRTC-internals* in the Google Chrome browser. With this tool, observations can be made on the local WebRTC connections. An end user may receive video frame rate, packet losses and bit rates. These statistics can also be read in real-time or be downloaded to a file. Since the data is collected per browser, the user has to manually collect and synchronize the data from all browsers in order to analyze multi-party sessions.

In a video call with two peers, each end point contains at least four tracks. Two are for audio (one for receiving audio and one for receiving) and the two others are for video, as seen in Figure 2.13. Each track is identified by a *SSRC* ID. If the conversation only has two peers, the SSRC ID is shared between the two. The sender would label its stats for the SSRC ID as *sent (bitsSentPerSecond)* whereas the receiver would instead label its stats for the same SSRC ID as *received (bitsReceivedPerSecond)*. Some of the statistics which can be used for both *sent* and *received* are *bitsPerSecond, bytes, packets, packetsPerSecond,* and video *frame rate*.

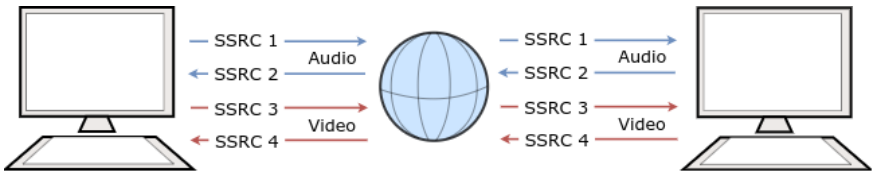


Figure 2.13: A two-party call where each peer uses four tracks

The statistics which Ammar et al. present are of the forms accumulated numbers, rate and ratio [4]. The *accumulated numbers* n_t at time point t is defined as $x_t - x_{t-\tau}$, where τ is the length of the observed interval and x_t is the sample at t. The *rate* r_t is defined as $n_t \,/\, \tau$. The *ratio* ρ_t is defined as $n_{a_1 t} \,/\, n_{a_2 t}$, where a is an attribute. An example of an attribute is 'bitsReceivedPerSecond'.

The data are split into four categories and the following attributes are observed to be relevant for each category:

- **Throughput and bandwidth** The available bandwidth r_t with a ='availableSendBandwidth' and bits received r_t with a ='bitsReceivedPerSecond'.
- **Packet loss** Packet loss ratio with ρ_t with a_1 ='packetsLost' and a_2 ='packetsSent'.
- **Picture indication loss (PLI)** PLIs received n_t with a ='plisReceived' and PLIs sent n_t with a = 'plisSent'.
- **Bucket delay** n_t with a ='bucketDelay'.

Among all the stats, the Ammar et al. observed that the following parameters as *"QoE killers"* (i.e. causes of severe video freezes) for a two-way communication channel:

- **Data rate drops** at the receiver side.
- **Packet loss ratio** which are non-zero.
- The **Picture Loss Indication** (PLI) is a WebRTC mechanism, where a value is sent to the receiver if a full video frame or more are lost. If this value is non-zero at either side of the media, the QoE is deteriorated.
- The **bucket delay** is "the time since the oldest queued packet was enqueued". Freezes may occur if the delay is non-zero.

The statistics focuses solely on the WebRTC-based video communication aspect. The audio aspect of the statistics is not analysed in the study.

Acceptable Attribute Thresholds for Video QoE

The effects of delay, packet loss and jitter impact the selection of buffer sizes and techniques used to repair packets.

Regarding delay, it is deemed to be an impactful factor for acceptable QoE [40, 33, 30]. Schmitt et al. investigated the threshold in video-mediated group discussions [33]. The results indicated that the participants in a group communication would be aware if the delay were between 650ms and 1150ms. When listening to *active* participants in a group conversation, the threshold would instead be between 100ms and 600ms. It also showed that if only one participant became aware of the delay, it would greatly degrades the whole group experience, as suggested by Ammar et al. [4].

Berndtsson et al. also report the effects of delays in video conversations [5]. In a subjective test in which a two-party audiovisual conversation occurred, synchronization between audio and video was an important factor. As long as the delay goes below 600 ms, the user would instead prefer to delay the audio to match the video. As for multi-part telemeetings, a delay of 800ms and higher was deemed unacceptable.

These threshold values can be compared to the measurement study conducted by Xu et al. [41] Three popular video chat systems were presented in the study: Google+, iChat, and Skype. In an ideal setting, the one-way delay was between 140ms to 270ms on average for the systems. However, in a video conversation between New York to Hong Kong showed higher values. The average one-way video delay using Google+ was 374ms and 788ms using Skype.

As for delay in a mobile video service context, De Pessemier et al. explored the threshold of when the delay of such service is unacceptable to an user [30]. 57 participants were asked to watch 14 videos under different bandwidth settings and video source qualities. They were then asked to answer subjectively of the quality of each video. Additionally, they were asked whether the video was deemed acceptable or not. Based on the evaluation, a model was

produced which indicated that video sessions which have a waiting time below 20 seconds have a relatively high probability (75%) to be accepted by the user, whereas waiting times longer than 60 seconds (over 75%) tend to be unacceptable to the user.

In terms of jitter and packet loss, Claypool and Tanner found in their study that jitter degradations were just as impactful to the video quality as packet loss [10]. In the experiment, 40 users were asked to each view 25 video clips at different jitter and packet loss settings. They were then to give a subjective score for each movie. In their findings, they found that even small indications of jitter or packet loss greatly degrade the quality of the video. However, any further jitter or packet loss beyond that was not as impactful.

3 Method

To address the research questions, the method of the thesis is separated into five phases.

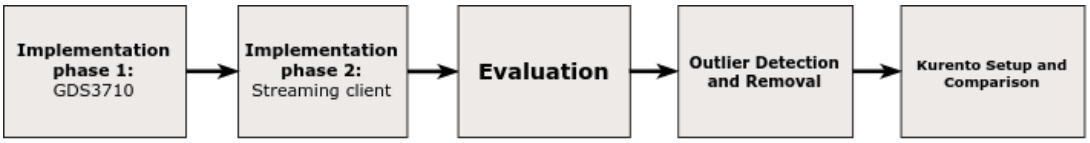


Figure 3.1: The five phases being performed in this thesis

Implementation Phase 1

To fulfill the requirements (outlined in section 1.5), we first establish audio transportation between the video door system and the communication tool. The implementation is illustrated in Figure 3.2. Both ends need to be connected to the Asterisk.

First, the GDS3710 is registered to the Asterisk server. The video door system resides on its own server. Accessing this server with the help of a browser provides the user with a GUI for configuration of the video door system. With the right credentials, the GDS3710 has the possibility to call any SIP phone.

The communication tool is also connected to the Asterisk. Cross-protocol communication between SIP and WebRTC is needed in order to provide audio to WebRTC-supported clients.

On the other side, an integration to the WebRTC-supported browser is needed. A challenge which rises in the integration is the capability for cross-protocol communication. To integrate the two ICTs across different protocols and architectures, a general and modular component is needed. Furthermore, we need a gateway capable of enabling communication to any ICT using a single communication tool.

A WebRTC gateway architecture should be relatively general, flexible and modular. On the other hand, the solution might not be as scalable as other more specific implementations, such as *webrtc2sip* (WebRTC-to-SIP gateway) [36] or *Medooze* (media server and MCU (Multipoint Conferencing Unit) with support to WebRTC) [24]. One could also argue the choice of Jitsi.

However, the Jitsi library lacks the support for RTSP streaming and is therefore excluded in this thesis.

Instead, a Janus WebRTC gateway (v0.4.1) is used. A client API built on the Janus REST API is used to connect to the gateway. With the correct credentials, we use the client's interface to establish connections with the Asterisk server implementation. The communication tool makes use of *React* JS library for front-end component building.

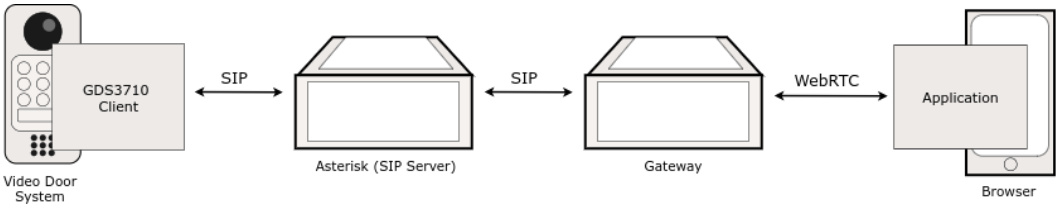


Figure 3.2: Overview of audio transport through SIP

Implementation Phase 2

As opposed to the audio transport implementation, no registration to the Asterisk is needed as long as the video door system server is reachable. For this thesis, the GDS3710 is located in a local network. Again, using a client API provided by a gateway is used to connect to the gateway. The gateway itself relays to the specified server (in this case the GDS3710 RTSP server). See Figure 3.3.

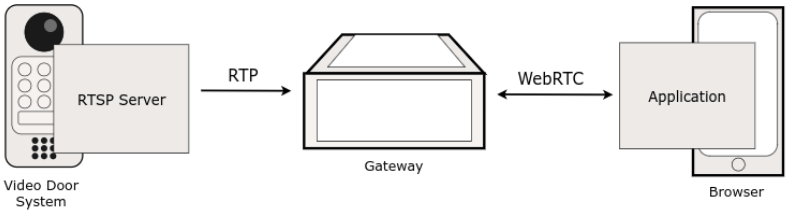


Figure 3.3: Overview of video streaming through RTP

Another solution to the implementation would be to manually transcode and relay the streams of the RTSP server directly to the communication tool application. More effective streaming would be possible. However, an implementation for effective support for video conferences would then be needed.

Evaluation

The *WebRTC-internals* (*chrome://webrtc-internals/* in Google Chrome) tool is used on a browser to gather WebRTC information which goes to and from the browser locally. The tool used in this thesis is embedded in Google Chrome. However, browsers with WebRTC support have support for such tracking. End users may view the data live represented as graphs and download them for further observation. These are gathered as JSON files and contain information such as bit rates, packet losses, and delays.

We run a series of tests, with a focus in a two-party telemeeting. The first party consists of the GDS3710. On the other party, our implemented extensions to the Briteback communication tool is used. These Janus and Kurento implementations are both run in a web application on a 4 Intel Core i7-7700 CPUs @ 3.6GHz and 16 GB of RAM with Ubuntu 16.04 OS. Both parties, including the gateway and media server, resides on the same local network. In the

Setting	Low	Medium	High
Bit rate	1024	2048	4096
Frame rate	10	20	30
Image quality	Very low	Normal	Very high
I-frame interval	20	40	60

Table 3.1: Video settings for the tests

telemeeting, we film a YouTube video (*Ink In Motion*[1] by *Macro Room*) through the video door system's camera to the communication tool. This is repeatedly done under different *video settings*: *low*, *medium* and *high*, for the Janus implementation. All of the settings are described in the Table 3.1. The *image quality* is an option defined by Grandstream. Refer to Figure 3.4 for an example configuration. The video door system films the same YouTube video for the equally long span of 200 seconds.

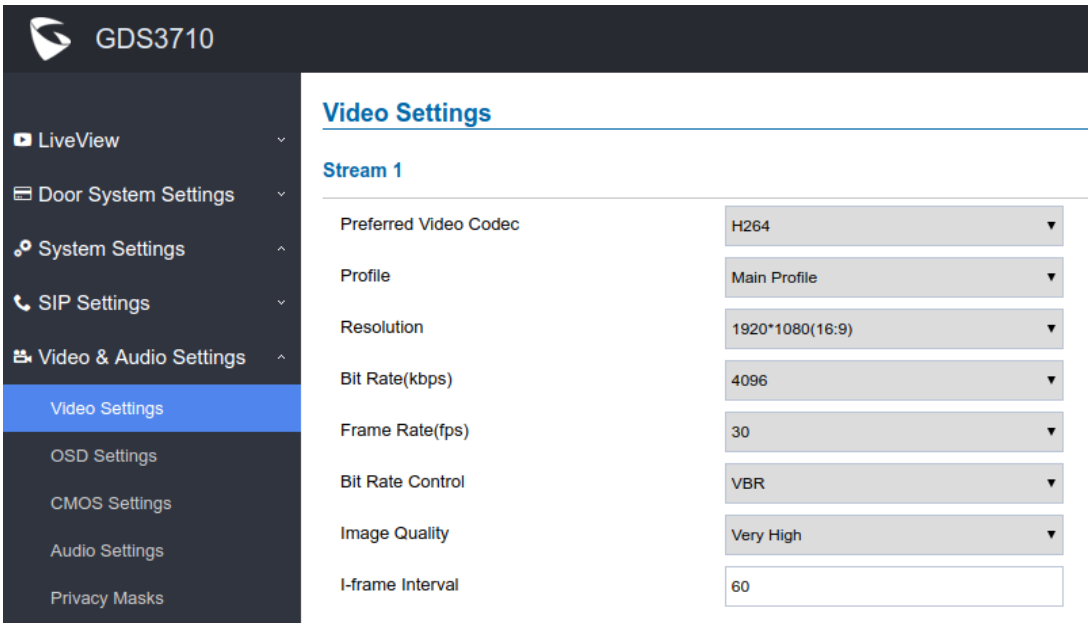


Figure 3.4: Video settings set to 'High' in the Grandstream GUI

As the two parties have their telemeeting, the WebRTC-internals tool simultaneously collects and downloads the data. Plots are then produced as a result of the downloaded data. They are set for a span of $t = 0, ...200$ with $\tau = 1$ between each data point. We also only evaluate video attributes. Evaluations of attributes audio sources are not included in this thesis. In addition, we only observe the data on the receiver side. This can be compared to the two-way video channel study conducted by Ammar et al. [4] Therefore, the following set of attributes are observed:

- The **available receive bandwidth** r_t ("the bandwidth that is available for receiving video data" [22]) with the attribute a = 'googAvailableReceiveBandwidth'
- The number of **bits received per second (bit rate)** r_t with a = 'bitsReceivedPerSecond'
- The **frame rate received** r_t with a = 'googFrameRateReceived'

[1] https://www.youtube.com/watch?v=BmBh0NNEm00

- The **jitter buffer delay** n_t (the sum of the time "each frame takes from the time it is received and to the time it exits the jitter buffer" [1]) with a = 'googJitterBufferMs'
- The number of **packets received** n_t with a = 'packetsReceived'
- The number of **packets lost** n_t with a = 'packetsLost'
- The number of **PLIs sent** n_t with a = 'googPlisSent'
- The **bucket delay** n_t with a = 'googBucketDelay'

One could argue to use the MOS to bring a more subjective study. However, Cien et al. claim several drawbacks with this method [8]. One example is that the user's perception may change over the course of a survey. The score of a question at the beginning of a test may not be as impactful as a score later of the test. Another is the *recency effect* in which humans tend to believe that the most recent event is the most impactful for setting the score. Users have different perspectives of the ratings. A "poor" rating may be interpreted in another way. Lastly, not much information can be taken from only getting a score. This means a larger sample size is needed to draw conclusions.

However, Fiedler et al. state that MOS can be also be "predicted from objective measurements of properties of delivered goods such as audio, video, or files" [15]. One such property is the jitter buffer delay, which we have included as an attribute we are observing with the WebRTC-internals tool. Furthermore, the authors expect that "excessive jitter to degrade the QoE in a similar way as real data loss". A threshold of around 100 ms is when the the user feels that the system reacts instantaneously. Holding below this threshold is ideal for qualitative QoE.

Outlier Detection and Removal

Due to a bug, as we refer to the *periodic bandwidth drops*, a periodic sequence of in which the bandwidth that is available when receiving video data goes to zero. As a result, we can make the observation of the Janus stop receiving bits and packets for a specific amount of time periodically. Consequently, this causes freezes on the client. As this is a deterministic behaviour, we have decided to cut off the *outlying* data points where these freezes occur. The location-scale model presented by Rousseeuw and Hubert [32] is used. It is a method of identifying values which we claim to not belong to the observation. We assume that every univariate observation is independent and identically distributed. It consists of a location parameter and a scale parameter. First, we want to find a *robust* method for obtaining the *location*, i.e. finding a location where the outliers do not exist. The most common estimator for such is the mean $\bar{x}$. However, the authors claim that the mean is not a robust method for estimating the location as it is very sensitive to few deviating values. The median is instead used as the location estimator. For the observation x_i, the median is defined as $x_{((n+1)/2)}$ if n is odd, $(x_{(n/2)} + x_{(n/2+1)})/2$ if n is even. The *breakdown value* is about 50%, meaning that the median method can resist up to 50% of outliers. The most common estimator for the *scale* parameter is the standard deviation $s = \sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2/(n-1)}$. However, since the mean is used in the standard deviation, it may become arbitrary large. Instead, another scale parameter based on the median is used. It is called the **median of all absolute deviations** (MAD) and is defined as:

$$MAD = c \times \underset{\text{i=1,...,n}}{median} \left| x_i - \underset{\text{j=1,...,n}}{median}(x_j) \right|,$$

where c is a *correction factor* which is 1.483 for normal distributions. This method is conducted on data which consist of rates, e.g. the available bandwidth or bit rates. To detect an outlier we use the rule as Rousseeuw and Hubert describe [32]. They base the rule on z-scores is used.

If $|z_i|$ exceeds the boundary $T_{z_i} = 2.5$, it is deemed as an outlier. It is motivated that a robust score can be calculated as follows:

$$z_i = \left(x_i - \underset{\text{j=1,...,n}}{median}(x_j)\right)/MAD.$$

Because we can make the assumption that the outlier is always at the lower bound, we instead mark any z_i below $T_{z_i} = -2.5$ as an outlier.

As for data of attributes which consist of accumulated numbers, e.g. the amount of packets lost and PLIs sent. We detect and remove the outliers if the derivative of the function is zero at some point, unless the accumulated number is zero or that more than 50% of the derivate between every interval is zero.

Kurento Setup and Comparison

We believe it is motivated to bring forward Kurento to the thesis, as it constructs a baseline and something to compare the Janus setup to. Kurento has one of the few (if not only) modular architectures which is similar to Janus. It comes with its own API, gateway, and plugins which an API developer has the capability to dispose of. A reference is in general motivated, especially when outlier removal is performed on the Janus QoE evaluation. Therefore, we will also implement a communication tool extension with the Kurento library.

The Kurento media server (v6.7.1) and an implementation of the Kurento client (based on the Kurento RTSP-to-WebRTC interoperability implementation [20]) is used. These replace the Janus gateway (illustrated as *Gateway* in Figures 3.2-3.3) and the implementation built on the Janus library (illustrated as *Application* in Figures 3.2-3.3) respectively. We repeat the process of using WebRTC-internals tool on this setup and perform a QoE evaluation as earlier described. Specifically, we perform the same test as in the Evaluation phase but only in low video settings. As no known periodic bandwidth drops is recorded on the Kurento solution, we instead directly make a QoE-based comparison of the data collected from the test using the Kurento media server and client to our Janus-based implementation. In order to fairly compare the two implementations, only the basic functions have been used. Only the standard JS library and libraries from either Janus or Kurento, depending on implementation, have been used in both implementations. Neither the Janus gateway library nor the Kurento media server library have been modified.

4 System Design

4.1 Scenario

The workflow of the communication tool can be seen in Figure 4.1. Visually, the communication tool GUI has three states. One of each is shown as in Figures 4.2-4.4. At the first state, no interaction with the video door system can be made. When a caller presses the doorbell button, the video door system makes a call through SIP to any device with SIP support. If the GDS3710 is calling the communication tool, the communication tool enters the second state. In this state, the communication tool receives a toast with a notification and a video stream from the camera is shown. At this state the callee has three options. The first two options are to decline the call or not performing any action. If either of those two options are chosen, the communication tool returns to the first state. The third option is to answer the call, entering the third state. At the third state, the screen which shows the video feed is significantly larger. Only then may the caller and callee have a two-way conversation through audio. However, the video is one-way, meaning that only the callee can see video from the video door system. The caller has no possibility to see the callee. If either of the parties end the call, the communication tool returns to the first state.

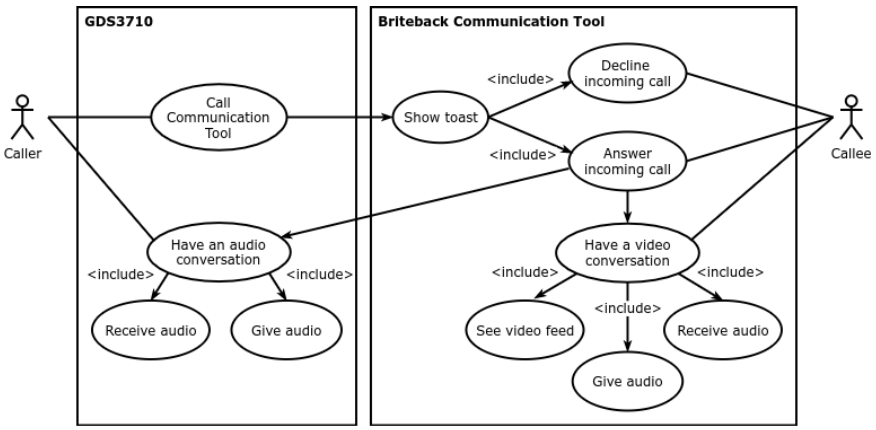


Figure 4.1: The use case diagram between the GDS3710 and Briteback communication tool

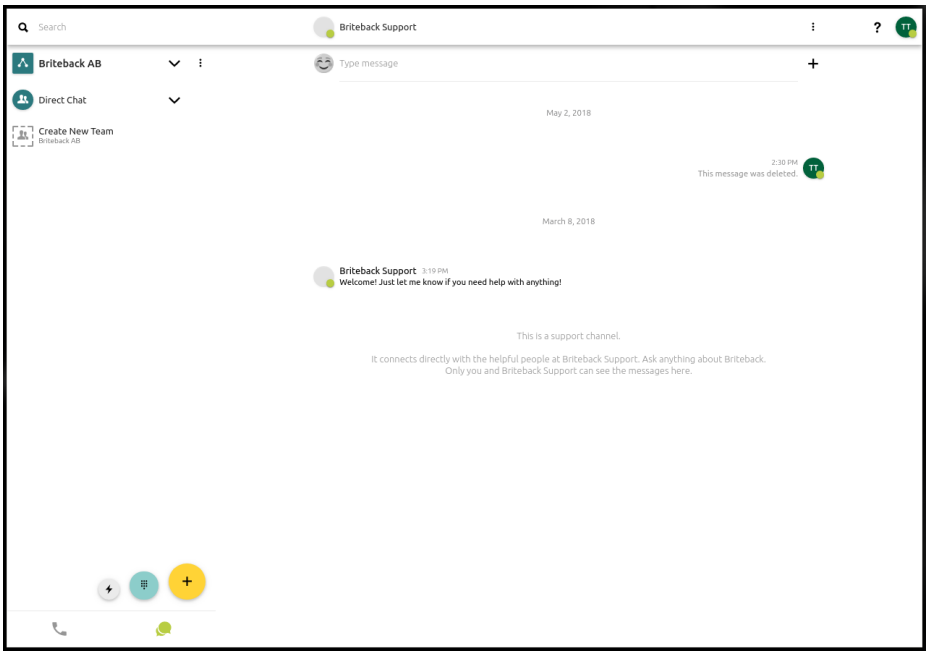


Figure 4.2: First state: The initial state of Briteback communication tool

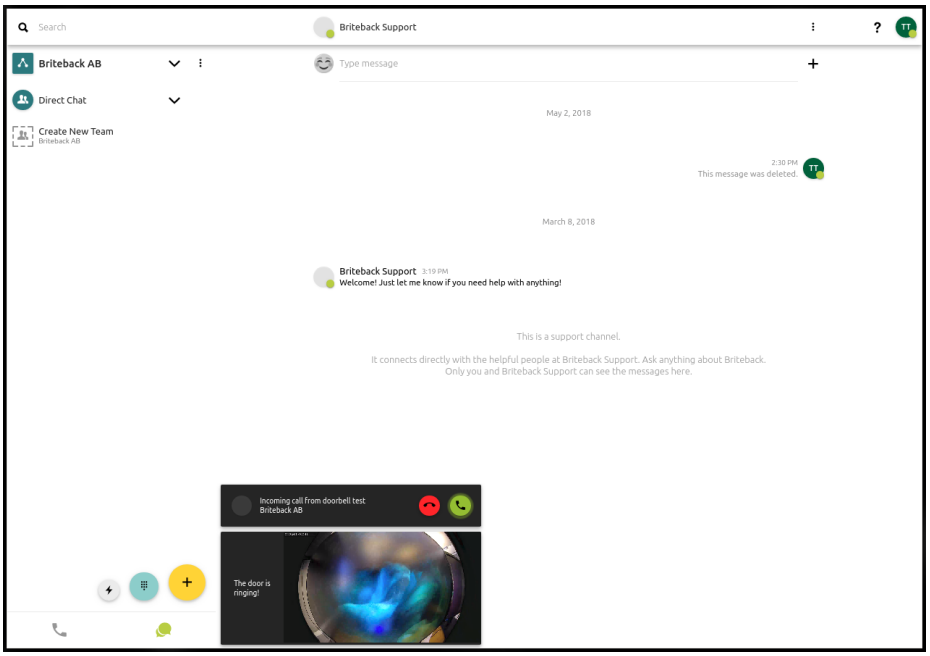


Figure 4.3: Second state: The communication tool has an incoming call from the GDS3710

4.2 System Overview

To understand the contexts of the implementation, we provide with an overview of the communication tool's architecture. The system is built using a basic setup of React and a *Webpack*-hosted development server. Figure 4.5 illustrates that the communication tool is composed of two parts. Those are the *Model* and *View* modules. Figures 4.6 and 4.7 illustrate class diagrams for the Model respectively the View. Observe that some methods have arbitrary parameters

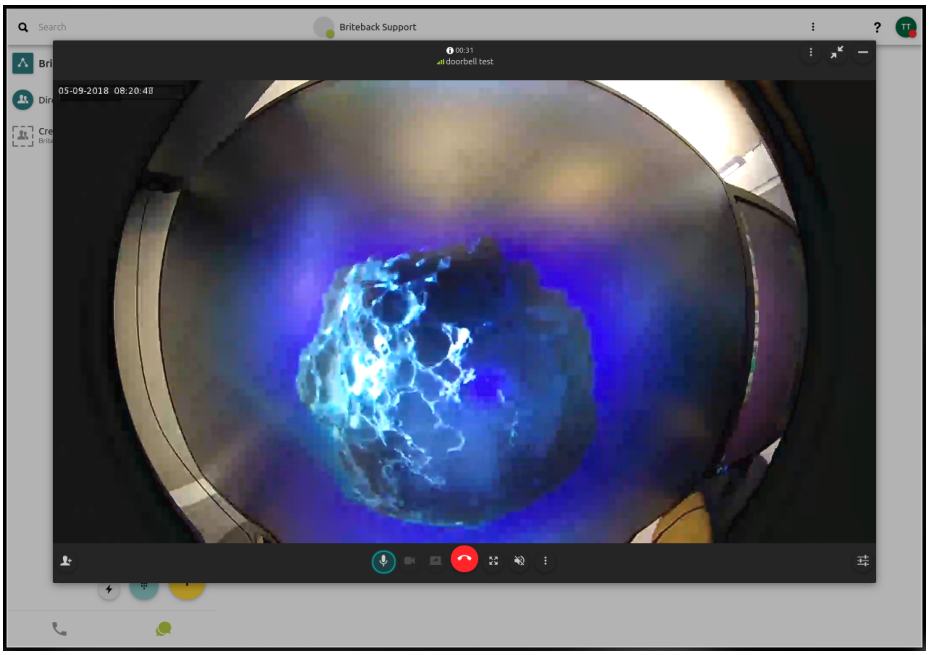


Figure 4.4: Third state: The communication tool having a video call with the GDS3710. To ensure that the call was not interrupted by inactivity, a YouTube video was played in front of the camera. Note that the screen and the movies on the screen (shown within the fish-eye lens) is rounded due to the fish-eye lens.

and we refer to them as the *arg* type. Additionally, `Button` and `Icon` are custom-made React components made for buttons respectively icons on the front-end.

The architecture of the communication tool is built based on the *Backbone.js*[1] structure. Backbone provides with `Model` (i.e. structures with interactive data and associated logic operations) and `Collection` (i.e. sets of models and contains events). Additionally, Backbone.js also has a `View` structure. However, we use the React View in the system. The connection between the two modules is handled by the `bus`. The `bus` in both class diagrams refers to the same class. Furthermore, the `bus` derives from the Backbone.js module **Events** which bind and triggers custom events. For example, if we receive an inbound call the appropriate toast is triggered.

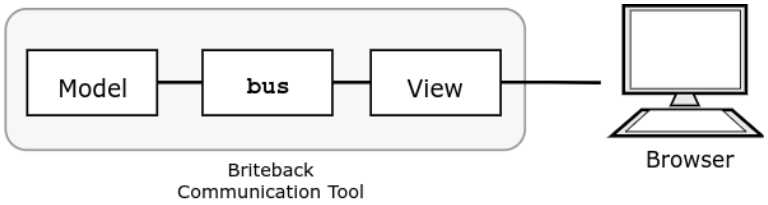


Figure 4.5: Overview of the communication tool architecture

Model Module

The Model module handles logic involving setting up and handling data structures for video conversation, handle WebRTC and Janus operations, and manage logic for events.

[1] http://backbonejs.org/

The main class of the Model module is the `voiceVideoRoomModel` and it handles everything which is associated with the video room data structures, video conversation logic and events.

Specifically, `voiceVideoRoomModel` handles the logic for video conversation setup and teardowns. It manages classical operations, e.g. parking, ringing and logic for inbound and outbound calls. The class conducts setups and teardowns of Janus connections and handles Janus and clients sessions. In addition, audio and video streams are set, renegotiated and stopped by this class. In terms of the data structure, it handles addition and removal of users and video rooms. Furthermore, `voiceVideoRoomModel` has logic for showing and hiding notifications e.g. toasts.

In a scenario where the user is called, we receive an event by the `bus`. If the call is accepted, the class sets up a Janus client with the Janus extension. A connection to Janus is established and a Janus session is created. Credentials to the Asterisk are used to create a SIP handle for the Janus session. Then, the instance registers the SIP handle. This is done with the help of the Janus SIP plugin. All media streams are gathered and connected. We tell the bus to look for existing `voiceVideoRoomModels` for the specific call. If none exist, a new is created. Otherwise, the existing room is joined.

The class extends the Backbone `Model` object. When its constructor is called, its attributes are filled with information of:

- Identifiers of video room, host, and the user of the component
- Collection of participants and WebRTC-based feeds to the peers
- Session information, e.g. whether it is active and identifiers
- `MediaStream` of WebRTC audio and video
- Janus SIP handles and Janus Videoroom handles
- Flags indicating if the current user of the model is host
- Additional information such as dial tones, connection delays, and voice mail information

A `voiceVideoRoomModel` is constructed for each peer. We then use an extended class of the Backbone `Collection` object, i.e. `voiceVideoRoomCollection`, to gather all peers within the same conversation into the data structure.

View Module

All the visuals are handled by the View Module. We use React `PureComponent` for the components. In most cases, a component contains another component. For example, the toast component `DoorbellToast` contains the video element component `Doorbell`. In the implementation, there are two occasions where the video feed is shown. The first is in a video conversation (i.e. the video element in Figure 4.4) and the other is on a toast (i.e. the video element in Figure 4.3).

For the video conversation, we use a single component `PeerVideo`. This contains most of the functionalities which exist in a video conversation, e.g. mute, play video and handle fullscreen.

For the toasts, the `ToastMaster` is used as the key class for toasts. It handles logic depending on the triggered events from `bus`, i.e. the component is initialized whenever an event to turn on or off is made. In addition, it takes arguments for any type of toasts. In this case, it uses the properties from the `DoorbellToast` component. `DoorbellToast` derives from two classes. The first, `Toast`, is a generic HTML element which all toasts consist of. It contains a `ToastButton` button which is used to cancel a toast, hence the invocation of `trigger` to `bus`. The second, `Doorbell`, is similar to `PeerVideo` but lacks any video conversation functionalities.

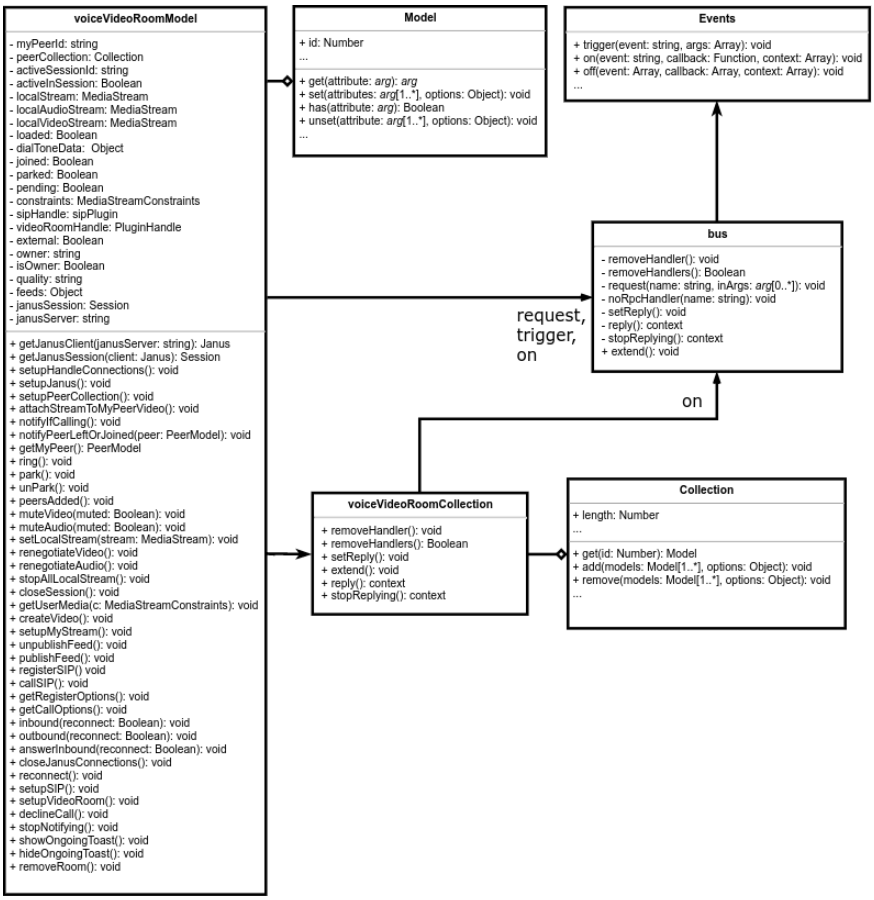


Figure 4.6: Class diagram of the Model

Janus

The streaming functionalities lie in the `doorbellStream` class. Conceptually, the *streaming* API is used as the code snippet below. Firstly, we include and initialize the `Janus` JS library. If it is done successfully, we attempt connect to the gateway and create a session. Thereafter, we attach a plugin. The *streaming* plugin takes information (such as identifiers) and gateway callbacks. The first gateway callback is triggered on successful attachment, i.e. `success`. When it occurs, we gain a plugin handle. Then, requests are called to *watch* and *play* the stream. If the error gateway callback is called or the destructor is called, some error handling respectively destruction handling is performed. The callbacks also include some event handlers `onmessage`, `onremotestream,` and `oncleanup`. The `onmessage` event is triggered when the Janus gateway receives WebRTC negotiation messages. Therefore, it is recommended to send any necessary WebRTC responses when entering this callback function. It is also here we receive response messages from asynchronous requests made to the gateway. `onremotestream` is triggered whenever the gateway receives a stream from the remote peer. The stream is a `MediaStream`, meaning that it can be applied to any `srcObject` if plain HTML video elements are used. `oncleanup` is called whenever the `PeerConnection` to the *streaming* plugin is closed. Whenever this callback function is entered, we want to remove any attachments on the front-end (for example any `MediaStream`).

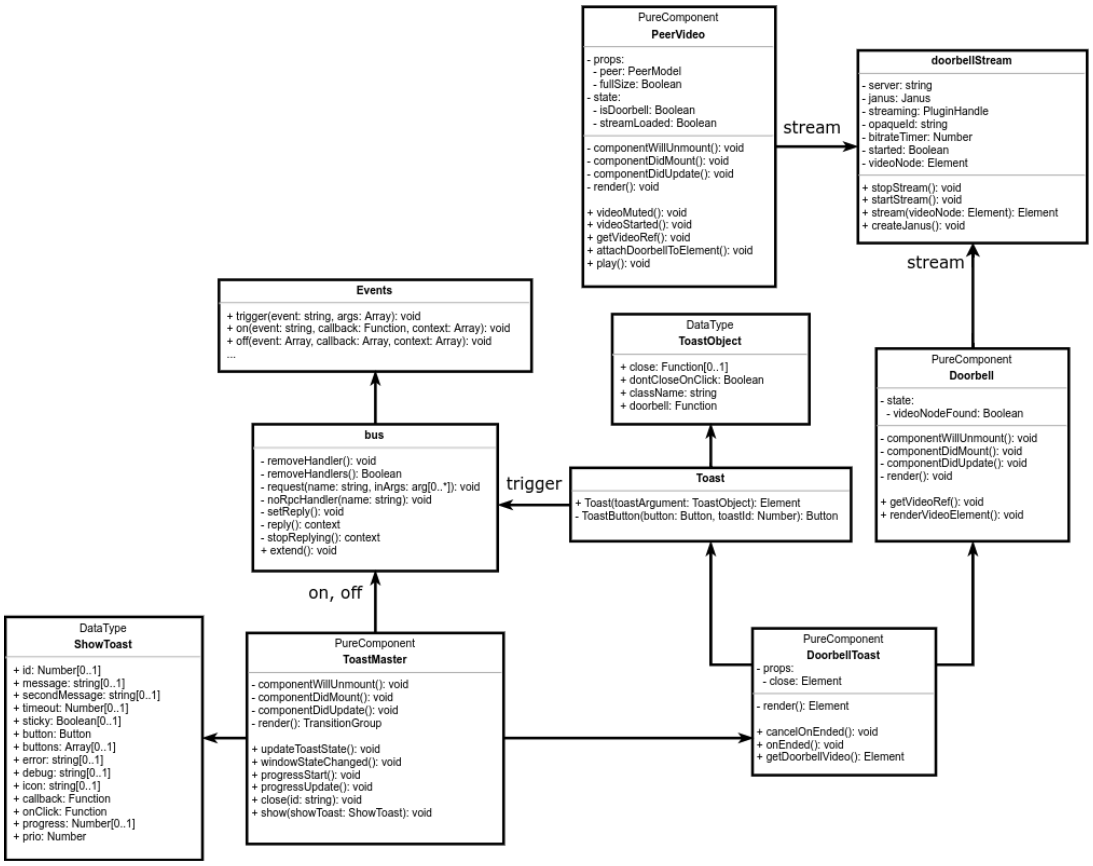


Figure 4.7: Class diagram of the View

```
var myPluginHandle = null;
var server = "wss://videodoorsystem.briteback.com/video_stream"

Janus.init({callback: function() {
  janus = new Janus(
    {
      server: server, // Server location of the Janus gateway
      success: function(){
        janus.attach(
          plugin: "janus.plugin.streaming",
          opaqueId: Janus.randomstring(12), // Some unique ID for the Janus
                                            // gateway session, which is
                                            // useful for mapping to the same
                                            // pluginHandle
          success: function(pluginHandle){
            myPluginHandle = pluginHandle;
            startStream(myPluginHandle);
          },
          error: function(error){ ... },
          onmessage: function(msg, jsep){
            // Messages/events from the plugin in form of WebRTC
            // negotiations with the help of JS Expression Parser
            const result = msg["result"];
            const status = result["status"];

            if (status === 'starting') {
              ...
            } else if (status === 'started') {
              ...
```

```
            } else if (status === 'stopped') {
              stopStream(myPluginHandle);
            }

            // Create and WebRTC answer, because we got an offer
            myPluginHandle.createAnswer({
                jsep: jsep,
                // Since we only want to receive the stream, not send it
                media: { mediaSend: false, videoSend: false },
                success: function(jsep) {
                  myPluginHandle.send({"message": { "request": "start" },
                                       "jsep": jsep});
                },
                error: function(error) { ... }
            });
          },
          onremotestream: function(stream){
            // When we finally have a connection, we receive a stream
            // from the remote peer. The stream is attached it to a video element
            attachStream(stream, videoElement);
          },
          oncleanup: function(stream){
            // Miscellaneous clean up operations
          }
        );
      },
      error: function(error) { ... },
      destroyed: function() { ... }
    }
  );
}});

startStream(pluginHandle) {
  pluginHandle.send({"message": {"request": "list"},
    function(result) { // We get a list of streams available
      const streamId = result["list"][0].id; // Pick the first stream
      pluginHandle.send({"message": {"request": "watch", id: parseInt(streamId)}});
    }
  });
}

stopStream(pluginHandle) {
  pluginHandle.send({"message": {"request": "stop"}});
  pluginHandle.hangup();
}
```

Except for the deployment of the Janus gateway with *streaming* plugin support, a configuration file for the streaming plugin needs to also be set. The example below shows the configuration of a RTSP stream.

```
[rtsp-test]
type        = rtsp
id          = 1
description = RTSP Stream Test
audio       = no
video       = yes
url         = rtsp://janus.testserver.se
rtsp_user   = admin
rtsp_pwd    = password
```

GDS3710

The software, provided by Grandstream, automatically handles connection operations to the server. All it needs are servers of where to connect and credentials to register the phone, as illustrated in 4.8. The important fields of the Grandstream phone is the SIP server (registrar)

and the outbound proxy. Both are configured in the Asterisk. Since the proxy server is reached by anyone, we need some credentials which we use to register to the server. An ID and password is needed to authenticate the phone. A SIP User ID is also used to give the phone an identity for other SIP peers to call.

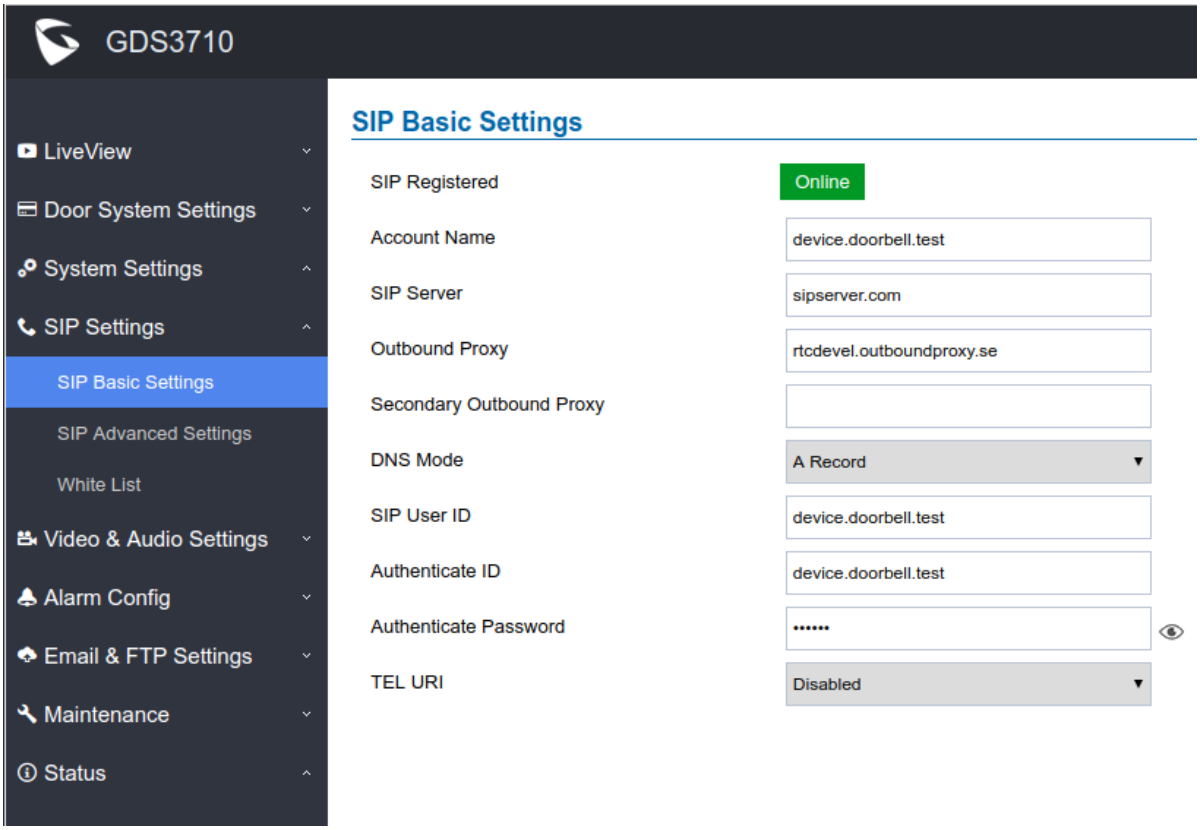


Figure 4.8: GUI for specifying your SIP settings

As earlier mentioned, the Asterisk needs to register SIP peers to prevent unauthorized peers to call. This means that any SIP User Agent, whether it is a phone or a computer, can call anyone through the Asterisk as long as they have the appropriate credentials.

5 Performance Results

5.1 Quality of Experience Evaluation

The tests and data gathering to follow commence when we answer the call (i.e. the third state as illustrated in Figure 4.4). By removing the outliers from the gathered data, the Figures 5.1-5.6 are produced. The acquired WebRTC data is shown starting with rates followed by accumulated numbers. The figures illustrate available receive bandwidth, bits received per second, frame rate received, jitter buffer delay, packets received and PLIs as a function of time over the span of $t = 0, ..., 200$ with $\tau = 1$ (i.e. approximately one second between every t) for three different video settings. The three settings, *low*, *medium* and *high*, are earlier defined in Table 3.1.

Figure 5.1 illustrates available receive bandwidth. We can observe similar behaviour for all settings. It starts off with a short initial rise and stagnates. Thereafter, it stays at the same level and periodically gains heavy increases. This behaviour repeats indefinitely with a mean rise of 9kb/s, 25kb/s and 50 kb/s for low, medium, and high settings respectively.

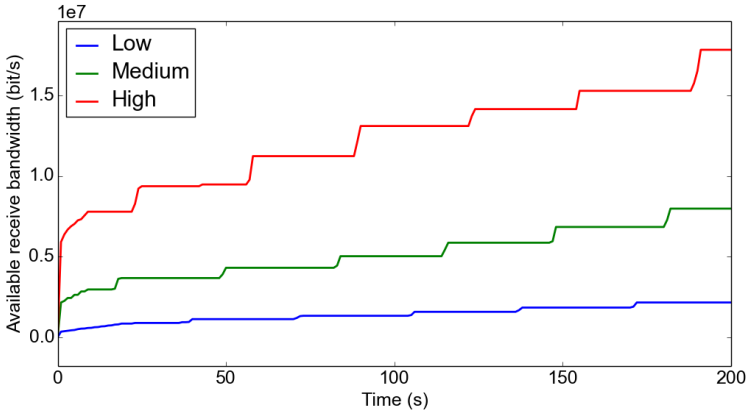


Figure 5.1: Available receive bandwidth as a function of time, using Janus gateway in different video settings

Attribute	Setting	Mean	Median	SD	MAD	T_{z_i}
Bits received per second $[kb/s]$	Low	3.7×10^2	3.6×10^2	1.6×10^2	2.0×10^2	−1.1
	Medium	1.5×10^3	1.5×10^3	3.4×10^2	3.3×10^2	−2.5
	High	4.4×10^3	4.5×10^3	8.8×10^2	6.6×10^2	−2.6
Frame rate received $[fps]$	Low	10	10	1.0	1.5	−1.3
	Medium	20	20	1.7	1.5	−2.5
	High	30	30	2.3	1.5	−2.5
Jitter buffer delay $[ms]$	Low	32	33	4.3	0.7	−2.5
	Medium	28	28	5.2	1.5	−2.5
	High	28	28	3.3	3.0	−3.4

Table 5.1: Values for different rate attributes and their settings

In Figure 5.2 throughput for the implementation can be seen. The bit rate stays at around same values for each setting, though the video settings with higher throughputs lead to more dispersion. This behaviour is similar with the frame rate, as illustrated in Figure 5.3. For all the settings, we can also observe that these value also stay the same and that the rate at which the amount of packets is received is almost constant. Janus tries to fixate both the bit rate and frame rate to certain values as specified in the video settings. In this case, the bandwidth is sufficient leading to both attributes to be considered stable. Even at high video settings, no values of zero have been reached.

Figure 5.4 shows the jitter buffer delay. It can also be observed that after a brief period the values are about the same for all video settings. We believe that a constant jitter buffer delay presumably enables the client to receive pictures between same time gaps. This means that the packets are most likely sent in the correct sequence. This is nevertheless an expected behaviour as no packets were lost nor were insufficient bit rate found.

In general, the plots for the rate attributes can be seen in Figures 5.1-5.4. The relevant values for the rates r_t can be seen in Table 5.1 including standard deviation, MAD, and threshold for z-score T_{z_i}. The threshold does not take into account the first data point as we can assume that it is always zero whenever the WebRTC-internals tool starts collecting data. Other thresholds have been applied to some of the figures instead of the lower *standard* threshold $T_{z_i} = -2.5$ (as referring to Rousseeuw and Hubert [32]) in order to remove additional important outliers associated with the periodic bandwidth drops.

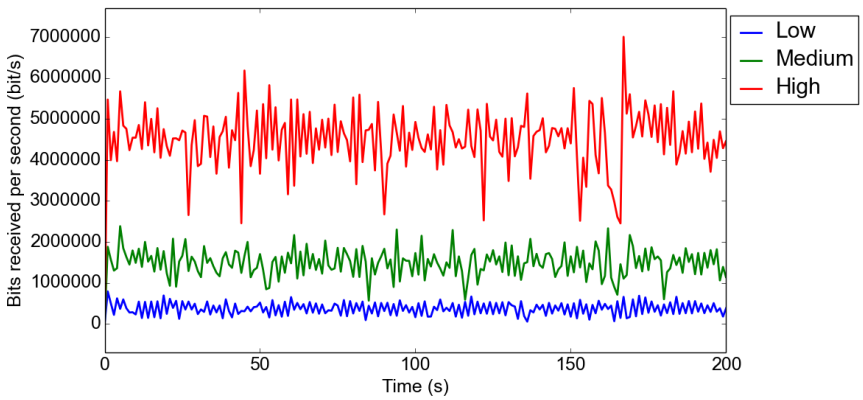


Figure 5.2: Bits received per second as a function of time, using Janus gateway in different video settings

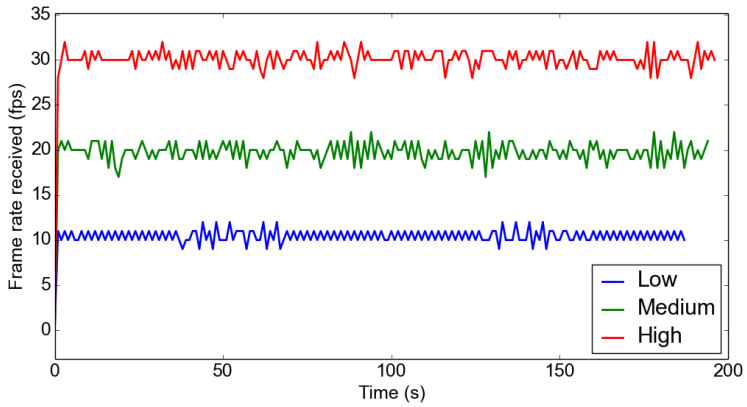


Figure 5.3: Frame rate received as a function of time, using Janus gateway in different video settings

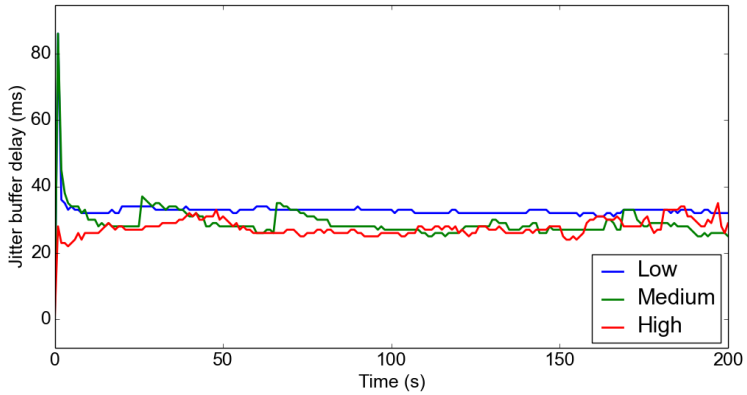


Figure 5.4: Jitter buffer delay as a function of time, using Janus gateway in different video settings

Figures 5.5 and 5.6 are for attributes defined as accumulated numbers n_t. First, Figure 5.5 illustrate the packets received over time. The amount of packets received are at a constant pace of 36, 133 and 382 packets/s for low, medium, and high settings respectively. There are some bumps in which the packets received switch between a steady rise to a gradual rise. The bumps are linked to the receive bandwidth. The periodic bumps on the available bandwidth leads to an increased amount of available packets a client may receive. Second, Figure 5.6 shows the Picture Loss Indication packets sent over time. These send at a constant pace of around 0.5 PLIs per second for all settings. The non-zero PLI rate be due to the CPU being unable to produce the full frame or insufficient receive bandwidth. Which one it is in this case is further explained in the PLI comparison between Janus and Kurento in section 5.2.

The Janus gateway, in the current setup, has zero bucket delays. We also have a maximum amount of packet losses at 12 for high settings over t. Despite increasing the video settings to high, no values of zero nor deterioration of QoE is found.

5.2 Kurento Comparison

The implementation of the Kurento client remains the same as for the implementation with the Janus library with the exception of `doorbellStream`. The functions `doorbellStream` and

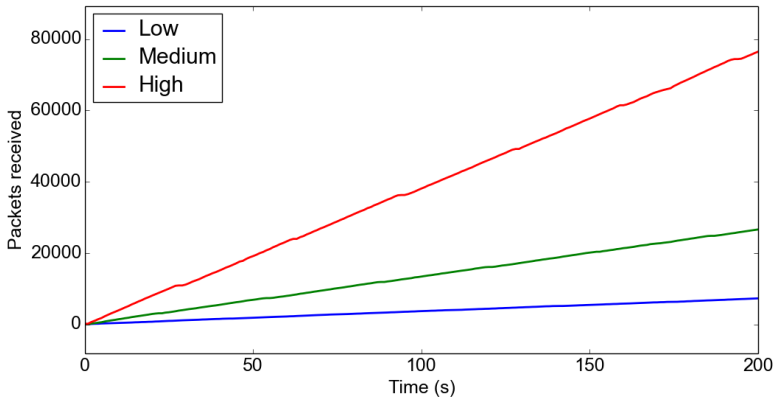


Figure 5.5: Packets received as a function of time, using Janus gateway in different video settings

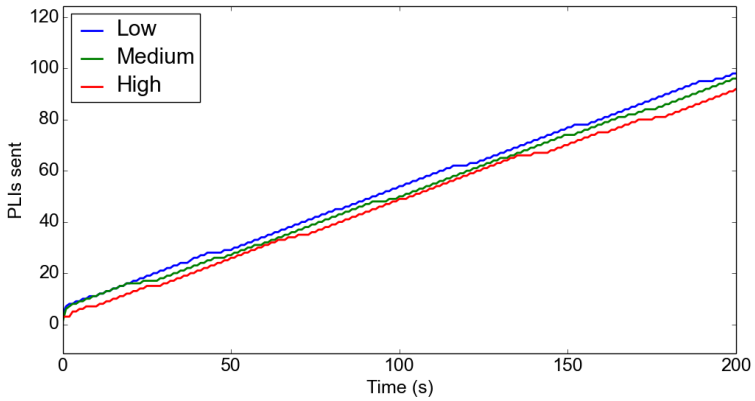


Figure 5.6: PLIs sent as a function of time, using Janus gateway in different video settings

the calls to it are replaced with methods from the Kurento library. The way the methods are invoked of the Kurento-implemented client are more similar to how the standard WebRTC library is used. We have event handlers for different states of ICE connections as well as handling for offers and answers with other peers. Recall from section 2.2 that Media Elements `WebRtcPeer` take care of the WebRTC streams from the client side and the `WebRtcEndpoint` is used for calling standard WebRTC API methods. In this case, the `player` is making the connection by taking the `WebRtcEndpoint` as sink media element.

```
window.addEventListener('load', function(){
  console = new Console('console', console);
  var address = document.getElementById('address');
  address.value = 'wss://videodoorsystem.briteback.com/video_stream';
  var pipeline, webRtcPeer;

  function startStream() {
    address.disabled = true;
    var options = { remoteVideo : videoElement };
    webRtcPeer = kurentoUtils.WebRtcPeer.WebRtcPeerRecvonly(options,
      function(error){
        if(error){
          return console.error(error);
```

```
        }
        webRtcPeer.generateOffer(onOffer);
        webRtcPeer.peerConnection.addEventListener('iceconnectionstatechange',
            function(event){
          if(webRtcPeer && webRtcPeer.peerConnection){
            // Make operations depending on IceConnectionState changes or
                IceGathering states
          }
        });
    });
  }

  function onOffer(error, sdpOffer){
    if(error) return onError(error);
    const ws_uri = server; // Media server location
    kurentoClient(ws_uri, function(error, kurentoClient) {
      if(error) return onError(error);

      kurentoClient.create("MediaPipeline", function(error, p) {
        if(error) return onError(error);
        pipeline = p;

        pipeline.create("PlayerEndpoint",
                        {uri: address.value, useEncodedMedia: false},
                        function(error, player){
          if(error) return onError(error);

          pipeline.create("WebRtcEndpoint", function(error, webRtcEndpoint){
          if(error) return onError(error);

                setIceCandidateCallbacks(webRtcEndpoint, webRtcPeer, onError);

                webRtcEndpoint.processOffer(sdpOffer, function(error, sdpAnswer){
                  if(error) return onError(error);

                  webRtcEndpoint.gatherCandidates(onError);
                  webRtcPeer.processAnswer(sdpAnswer);
                });

                player.connect(webRtcEndpoint, function(error){ // Connection of
                    PlayerEndPoint and WebRtcEndpoint
                  if(error) return onError(error);

                  player.play(function(error){
                    if(error) return onError(error);
                  });
                });
              });
        });
      });
    });
  }

  function stopStream() {
    address.disabled = false;
    if (webRtcPeer) {
      webRtcPeer.dispose();
      webRtcPeer = null;
    }
    if(pipeline){
      pipeline.release();
      pipeline = null;
    }
  }
});
```

```
function setIceCandidateCallbacks(webRtcEndpoint, webRtcPeer, onError){
  webRtcPeer.on('icecandidate', function(candidate){ // Handle local IceCandidate
    candidate = kurentoClient.register.complexTypes.IceCandidate(candidate);
    webRtcEndpoint.addIceCandidate(candidate, onError);

  });
  webRtcEndpoint.on('OnIceCandidate', function(event){
    var candidate = event.candidate; // Handle remote IceCandidate
    webRtcPeer.addIceCandidate(candidate, onError);
  });
}
```

As a result of the tests performed on the Kurento client implementation and Kurento media server deployment, the Figures 5.7-5.12 are produced. Similar to the figures produced from the Janus implementation tests, they span over $t = 0, ..., 200$ with $\tau = 1$. No outliers are removed and only data from Kurento streaming in low video setting is gathered for this comparison.

Similar to the low video setting used in the Janus implementation, we have zero bucket delays and total packet loss below 20. In terms of available bandwidth, as illustrated in Figure 5.7, we also have a short initial rise which later stagnates. Then we may observe periodic sharp rises, similar to Janus.

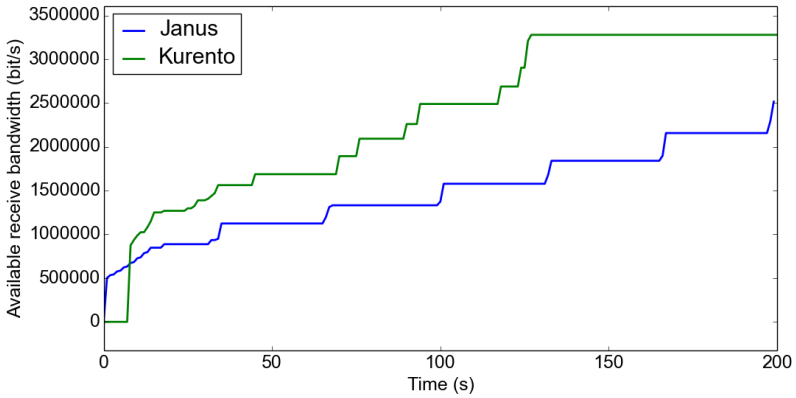


Figure 5.7: Available receive bandwidth as a function of time, using Janus gateway and Kurento media server in low video setting

As for the bits received per second, as shown in figure 5.8, Kurento occasionally has higher throughput than Janus. At times, it reaches rates of up to 1.2 Mbit/s. However, we can also see that the rate at indeterminate occasions goes to zero, despite having a larger receive bandwidth. It is likely that the Kurento media server handles the signaling is different. We believe that the Kurento solution instead strives for a constant high throughput, but falls off when not enough receive bandwidth is available. This affects the video performance as reaching zero in rate attributes greatly deteriorates the video QoE.

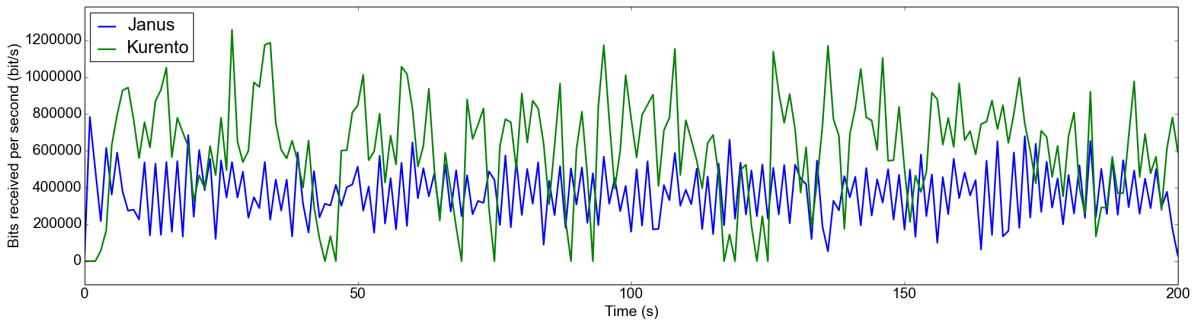


Figure 5.8: Bits received per second as a function of time, using Janus gateway and Kurento media server in low video setting

Comparisons of frame rate and the jitter buffer can be seen in Figure 5.9 respectively 5.10. In both comparisons, some sudden drops to zero occurs for the Kurento implementation. Also in this situation, a zero-value frame rate significantly affects the QoE. In the comparison, these drops are properties which do not exist in the Janus implementation. This is also presumably due to Kurento occasionally having insufficient receive bandwidth. On the other hand, the mean values of both attributes are higher than Janus. We believe that the mean is more relevant in this comparison, as in this case the median does not consider the drops.

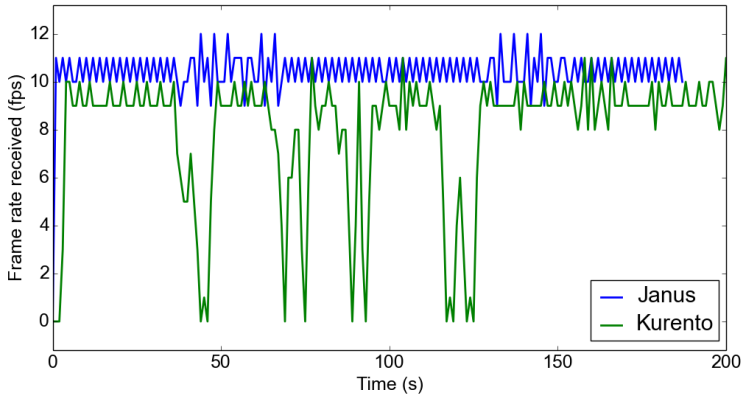


Figure 5.9: Frame rate received as a function of time, using Janus gateway and Kurento media server in low video setting

Figure 5.11 shows a comparison in the amount of packets received over time. Observations of a steady rise on the packets received can be made. Some bumps exist in both implementations. Despite reaching zero in some attributes, the general consensus is that Kurento utilizes more bandwidth, gathers more packets, and has a higher average bit rate. However, the performance difference between the two is marginal. In Figure 5.12, a comparison in the PLIs sent can be seen. We note that the Kurento implementation has zero PLIs sent as opposed to the Janus implementation. This can be explained by Kurento having an overall higher receive bandwidth, meaning it is more likely to receive full pictures.

Once again, we observed that the Kurento implementation in Table 5.2 has a higher bandwidth and bit rate in average. However, the mean frame rate is lower on Kurento. Additionally, Kurento has a higher dispersion in all attributes. The frame rate and jitter buffer delay is no-

ticeably more disperse in the Kurento implementation comparing to the Janus implementation. Intuitively, higher value dispersion tends to higher variation.

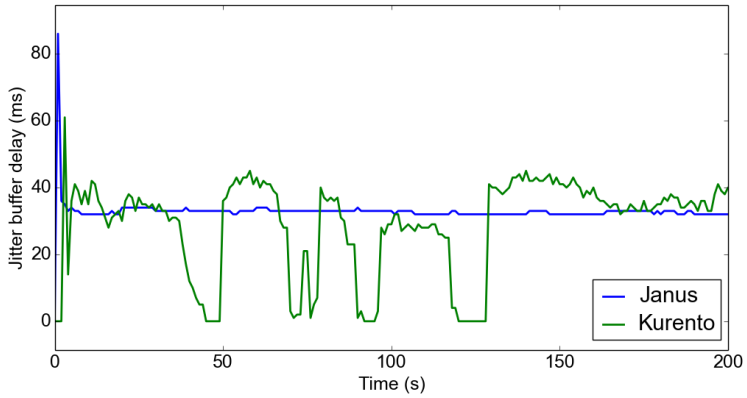


Figure 5.10: Jitter buffer delay as a function of time, using Janus gateway and Kurento media server in low video setting

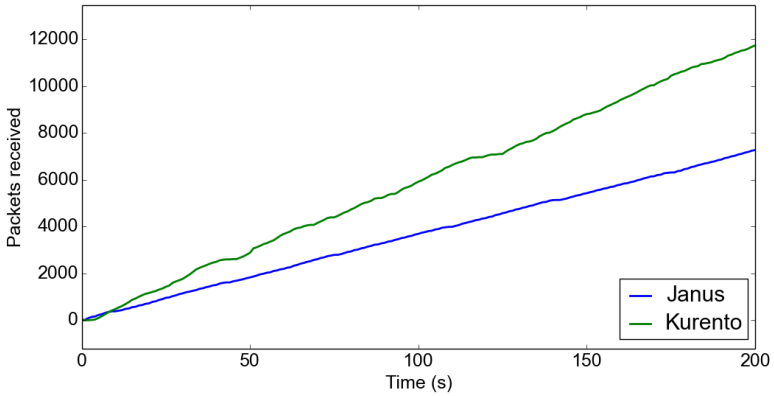


Figure 5.11: Packets received as a function of time, using Janus gateway and Kurento media server in low video setting

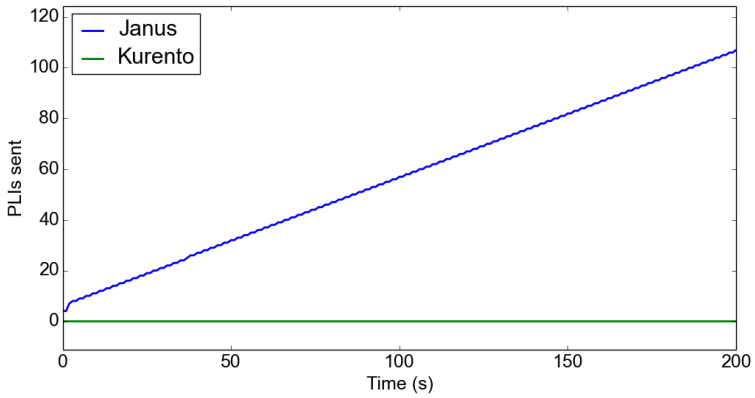


Figure 5.12: PLIs sent as a function of time, using Janus gateway and Kurento media server in low video setting

Attribute	Implementation	Mean	SD
Available receive bandwidth	Kurento	2.3×10^6	9.1×10^5
	Janus	1.4×10^6	4.9×10^5
Bits received per second	Kurento	6.1×10^5	2.8×10^5
	Janus	3.7×10^5	1.5×10^5
Frame rate received	Kurento	8.0	2.7
	Janus	10.4	1.0
Jitter buffer delay	Kurento	29.4	14.1
	Janus	32.8	4.3

Table 5.2: Values for rate attributes of Janus and Kurento implementation in low video settings

6 Discussion

6.1 Results

System and Implementation

Currently, the existing communication tool system has support for additional functionalities, e.g. call parking, mute functionality and storage of video conferences. However, the streaming plugin implementation is light-weight and fully built on front-end. Therefore, no back-end functionality is needed to fulfill the defined requirements.

On the other hand, additional features on the implementation would further integrate the implementation to the communication tool system. If a call park functionality would be introduced to the communication tool integration, an extension supporting back-end operations involving the streaming plugin would be needed. Saving information such as session identifiers and callback functions for the streaming plugin are necessary to track the video communication state. A more general streaming implementation, with MJPEG video streaming support or any other video using RTP, is possible. Also, a feature of adjusting bandwidth, bitrate and resolution depending on QoE drops is possible. At the moment, the video door system acts as a SIP phone. The implementation lacks the capability to open the door nor any associated door system functionality.

The Grandstream interface provides with many complex configurations to the video door system. However, its HTTP API lacks the possibility to retrieve video stream.

Quality of Experience Evaluation

In the results, we recall that the PLIs are being sent at a constant rate. However, we believe that a PLI rate of 0.5 is negligible. No correlation between the stats presented in Figure 5.6 and unacceptable QoE occurring can be found. This contradicts the non-zero PLI value QoE killer claims by Ammar et al. [4].

Despite the different video settings, the frame rates (see Figure 5.3) were still considered stable. They deviated with a maximum of 2 fps from their assigned frame rates for every video setting. As for jitter buffer, as seen in Figure 5.4, the difference between the maximum and minimum (except for the initial spike for medium settings) is at most 10 ms. This is almost negligible to the human eye, meaning that delay is considered stable.

Based on the presented performance stats, we argue that the current implementation and integration is *acceptable*.

Kurento Comparison

In a comparison between Kurento and Janus, we observe some similarities of the attributes shown in both gateways' Plots. Firstly, the bandwidth which is available for receiving video data increases significantly in periods rather than gradually or constantly. In addition, Kurento nor Janus loses any packets or has any non-zero bucket delays.

On the other hand, we observe some significant differences between the two. As earlier observed, Kurento acquires higher average values at the rate attributes. However, the consistency of the rate must be considered. The bit rate and frame rate that the client receives is more consistent using Janus and never reaches zero as opposed to Kurento. Having a value of zero in either bit rate or frame rate is a QoE killer. Fortunately, Janus did not reach zero in any of the video settings.

Despite having the same median jitter buffer delay in both implementations, it is also much more consistent on the Janus. We believe that a more consistent jitter buffer delay has a higher video QoE score as we want the frames to arrive to the user as distributed as possible. Keeping it constant prevents changes in the jitter playout buffer size. In addition, a jitter buffer delay below 40 ms is considered to be almost instantaneous to the humans according to Fiedler et al. [15] A MAD of maximum 3 ms is in this case negligible.

6.2 Method

In the current Janus setup, no indications of jitter, freezes or blackouts were shown. However, the Janus gateway, Kurento media server and clients implemented on Janus and Kurento JS libraries are run in a test environment with a local network. This does not reflect a realistic environment where the available bandwidth may be more limited. However, assessing the study in a controlled environment gives the possibility of performing well-defined simulations.

Recall that we have applied outlier detection and removal by the location-scale model. One could argue that the choice of removing these outliers is not perfect. We made the assumption that each data point is independent from each other. This may not be true since the cause of the periodic bandwidth drops are unknown. If we instead made the assumption that they are mutually independent, the replicability of the evaluation would differ if the periodic bandwidth drops would not occur. Intuitively, it is easy to observe at what timestamps the bandwidth drops occur. Despite different resolution, frame rate or bit rate adjustments, the behaviour of the periodic bandwidth drops remains the same. A method of removing any value below a constant value boundary around zero is sufficient for removing the most impactful outliers. On the other hand, some artifacts, caused by the location-scale method, can be observed. Figures of the available receive bandwidth (see Figures A.13-A.15) and the frame rate received (Figure A.23) show that the method removes data points at the start of the figures. Whether they are related to the bandwidth drop or are only gateway startup periods is unknown. Therefore, the location-scale method is more reliable the later the streaming goes.

Consider the comparison of the bandwidth available that we receive video data on Janus (see Figures A.13-A.18). Initially, we believed that the bandwidth drops being the cause of the heavy rises as a mechanism by Janus and/or WebRTC to increase if a drop occurs. However, if comparisons of the overall behaviour of the Plots are made with the Kurento implementation (A.1), we can see that they are somewhat similar. All the Plots are constant with periodic heavy rises.

In the majority of cases, $T_{z_i} = -2.5$ is a valid threshold for outlier detection. However, there are some cases where the drop is not outlying enough to be detected. In Table 5.1, we can see that modification of the threshold for a = 'bitsReceivedPerSecond' and a = 'frameRateReceived' most likely threshold lies where MAD is too large in relation to the median. However, for a =

'googJitterBufferMs' we have to observe Figure A.27. The initial dip at $t = 4$ is an example of the multiple inconsistencies with the jitter buffer delay. A threshold adjustment is needed despite the noticeable dips caused by the bandwidth drops. This is also considered to be a drawback with MAD. In this case, it would be preferable to use standard deviation for the location-scale model as the initial dip is an important part in the calculation. As for the available receive bandwidth, the threshold's magnitude needs to be larger the larger t is. Therefore, it is not recommended to use location-scale model here.

In the removal of outliers for attributes of accumulated numbers, we find that no association exists between PLIs sent and the bandwidth drop. No periodic behaviour in the Plot was found.

The QoE evaluation is based on the study conducted by Ammar et al. [4]. The defined attributes which are claimed to be QoE killers could be questionable. Throughput, bandwidth and bucket delay are always related to video freezes. However, consistent correlation between PLI, packet losses and video freezes is missing. The attributes miss motivation for the claims to be causes of video freezes. We believe that non-zero PLI and packet losses to not necessarily be QoE killers but if the rate of when these occur reaches some unacceptable threshold.

6.3 Sources

In the fields of QoE in video, there are many well-known sources, e.g. the studies conducted by Pinson et al [28]. Currently, there lacks any multimodal QoE assessment and thereby lacks a clear overlooking perspective of a user's QoE [8]. We believe that the concept of QoE is broad and complex as it lacks concrete guidelines. For example, the definition of QoE is different from different studies [8]. Also, methods to conduct a QoE-based study is differently motivated. We conducted the study due to the drawbacks of MOS [8], i.e. change of perception over the course of a survey, recency effects, different perspectives of a score, and the need for a larger sample size. However, MOS is "a typical user-related measure" [15] and has been used in a multitude of QoE-based studies over the years [15, 38, 30]. The acceptance in video quality does not solely lie in the metrics, but in the user. Therefore, we believe it is motivated to assess a subjective method on the Janus implementation based on MOS.

The selection of sources which are WebRTC-based are not based on research articles as many WebRTC-related concepts are not mentioned. Fortunately, most of the concepts are specifed in W3C working draft involving WebRTC statistics API [1]. In addition, many of the concepts (e.g. PLI) already have well-known definitions in the contexts of video and are arguably similar (e.g. IETF's definition of PLI [27]).

6.4 The Work in a Wider Context

A big issue is that the leading real time communication services have caused an "fragmentation of technologies creating interoperability barriers among these services" [14]. This relates to the selection of ICTs (in section 1) causing "technological divides" amongst users. WebRTC is an important initiative aimed to standardize end the fragmentation by using an unified and standardized technology [14]. The technology is open in standards and provides the user with open source solutions. This gives the application developer the opportunity to create cross-browser web-based communication across any major browsers, e.g. Google Chrome, Mozilla Firefox, Apple Safari, Microsoft Edge and Opera. This paper shows one of many cases of how communication can be un-fragmented and cross-platform. Janus is one of many open source solutions. Even legacy protocols, e.g. SIP used by the GDS3710 video door system, can be integrated with an WebRTC-supported browser.

Recall the effect of QoE-based quality assessment in section 2.4. Venkataraman and Chatterjee claim that the QoE in video is where service providers focus their investment on [37]. The term is closely related to customer satisfaction and experience-related businesses. Furthermore, the "QoE has been the most significant measure of human satisfaction in these

domains". It is deemed important to understand and further improve the concept of QoE. In our work, we have looked into the relevant QoE performance factors in an one-way video communication context.

7 Conclusion

In this paper, we have covered how an additional ICT, namely an IP video door system, can communicate cross-platform with a WebRTC-supported client. A minimal front-end solution has been implemented, demonstrating that the video door system can be integrated into the Briteback communication tool. Objective QoE quality assessment was then performed on the implementation. As a result of the evaluation, we claim that the integration is acceptable for the specified scenario. We also state, in terms of the QoE-based quality on the receiver-side video, that the Janus gateway and API is more consistent and stable than the Kurento media server and API.

For future work, we aim to further implement the integration, adding a back-end integration and storage support of important streaming plugin information. Possibilities to stream from other sources than RTSP, via HTTP or other video sources using RTP, would further help the integration with other types of products than the GDS3710.

We aim to address the acceptance of the implementation in mobile applications with WebRTC and the video door system in a public domain. Evaluations would be conducted under different network conditions. Alternatively, simulations of different realistic bandwidth scenarios could be more appropriate as the scenarios are more well-defined. It would also be possible to simulate errors, e.g. simulate packet losses and send PLIs. This would further correlate the video freezes and the QoE performance factors. We would also bring out the drawbacks of Janus in terms of QoE.

Also, we motivate to assess a subjective QoE evaluation using MOS on the implementation as it would further confirm whether the implementation is acceptable or not. In the end, the acceptance of the video implementation ultimately lie in the users of the system.

Additionally, we will also re-address the possible issues caused by the periodic bandwidth drops on the Janus after a solution to it has been found. Whenever the issue has been resolved, we will try to confirm that no QoE-based performance metrics measured were affected by it.

A Appendix A

In this chapter, we show additional results from the tests in the QoE evaluation. Every figure in the appendix is shown as a function of time $t = 0, ..., 200$, $\tau = 1$, for different video settings. Low video settings are shown as blue lines, medium settings are shown as green lines and high video settings are shown as red lines.

A.1 Attributes using Kurento Media Server (Low Video Setting)

QoE-related attributes for the Kurento media server on low settings are shown in the figures. The statistics were gathered with the WebRTC-internals tool. No outliers were removed.

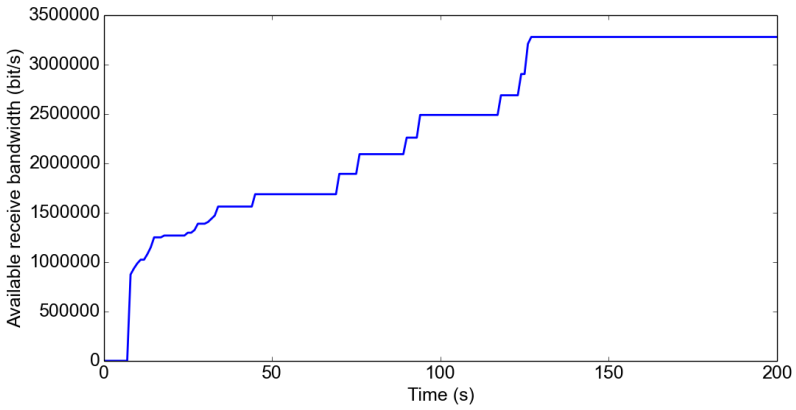


Figure A.1: Available receive bandwidth as function of time, using Kurento media server in low video setting

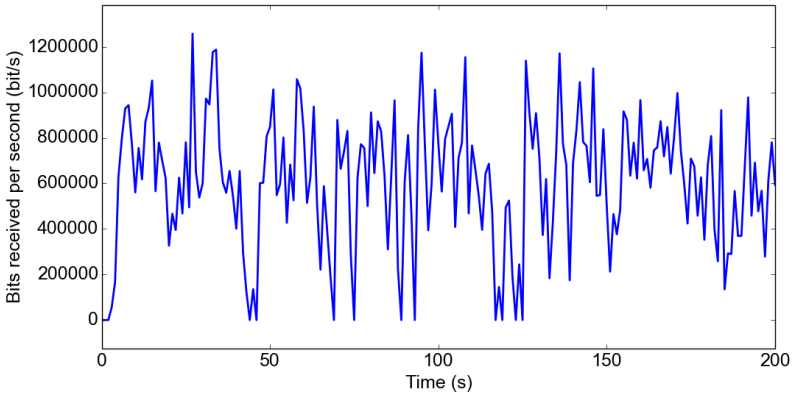


Figure A.2: Bits received per second as function of time, using Kurento media server in low video setting

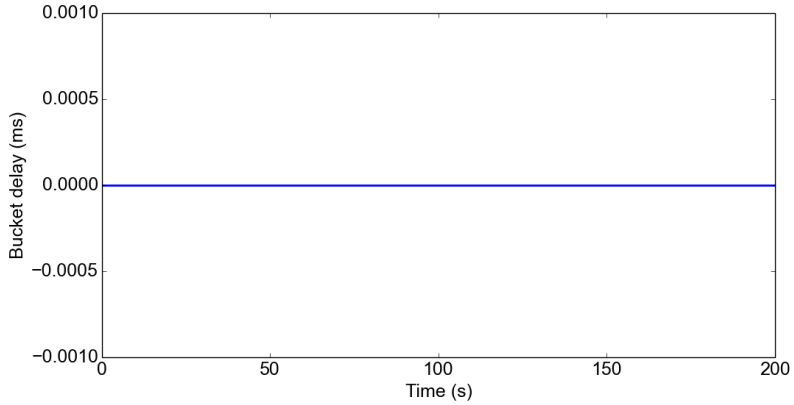


Figure A.3: Bucket delay as function of time, using Kurento media server in low video setting

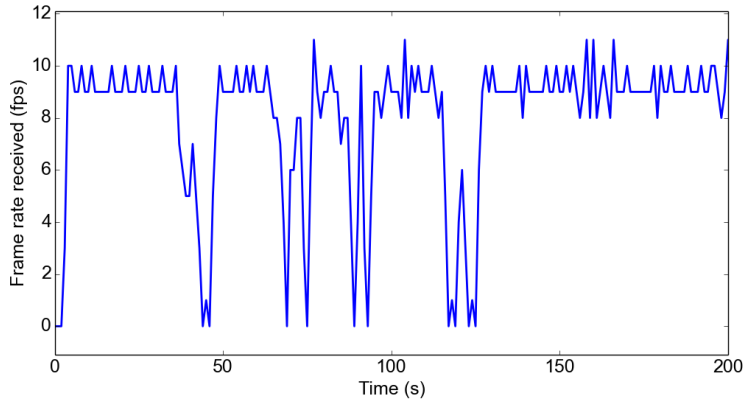


Figure A.4: Frame rate received as function of time, using Kurento media server in low video setting

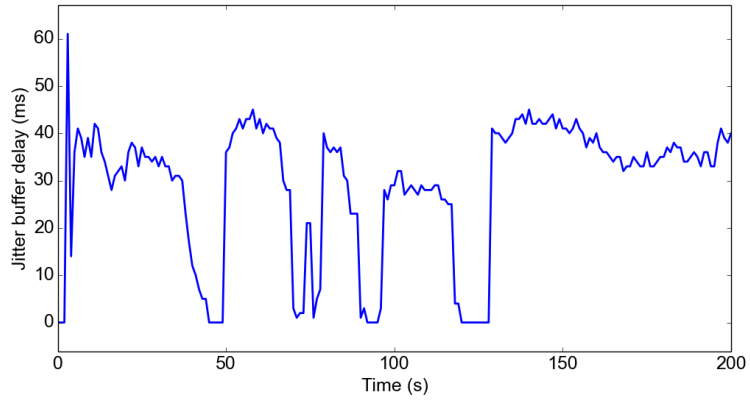


Figure A.5: Jitter buffer delay as function of time, using Kurento media server in low video setting

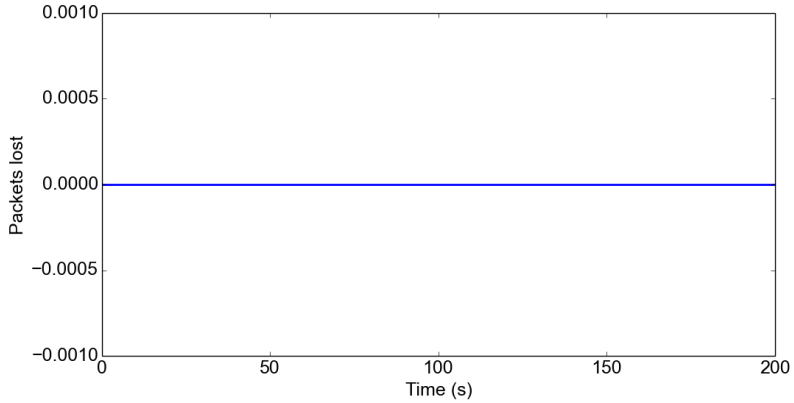


Figure A.6: Packets lost as function of time, using Kurento media server in low video setting

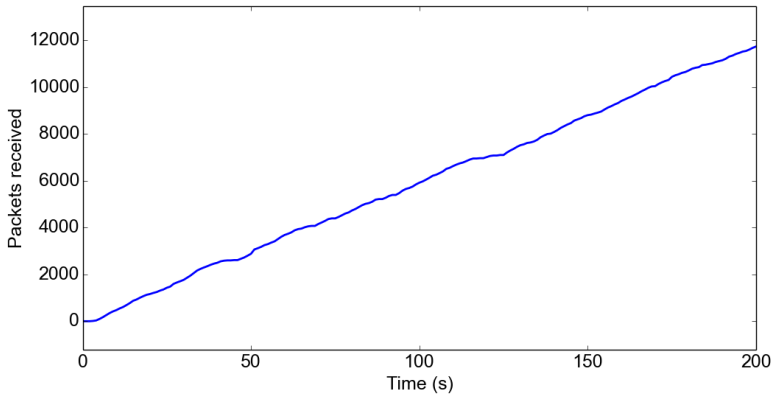


Figure A.7: Packets received as function of time, using Kurento media server in low video setting

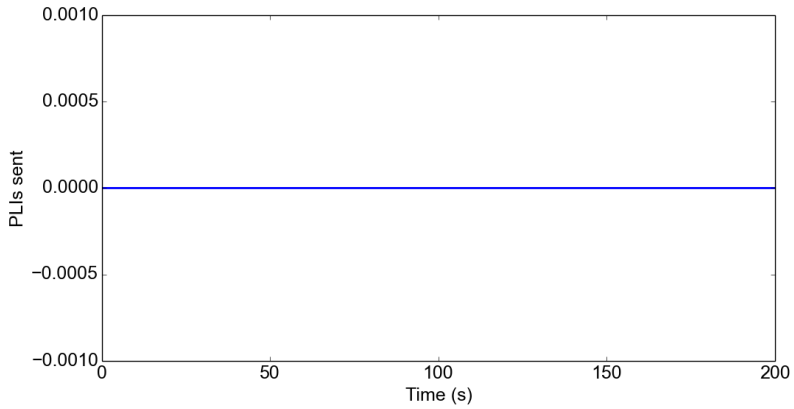


Figure A.8: PLIs sent as function of time, using Kurento media server in low video setting

A.2 Additional Attribute Comparisons Between Kurento and Janus

The attribute comparisons provided are always zero and therefore not included in the results.

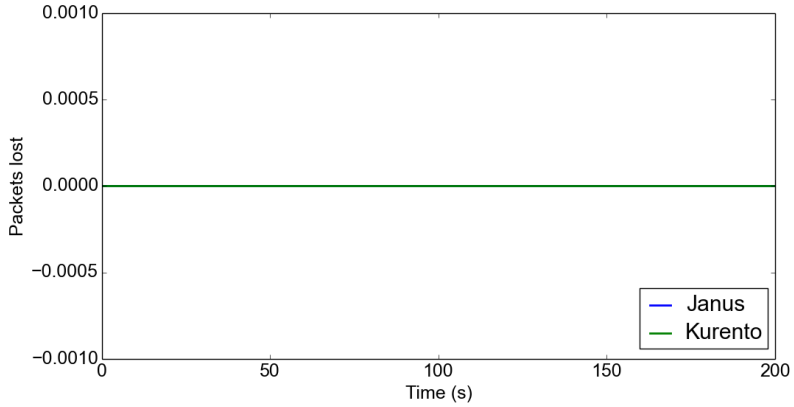


Figure A.9: Packets lost as function of time, using Janus gateway and Kurento media server in low video setting

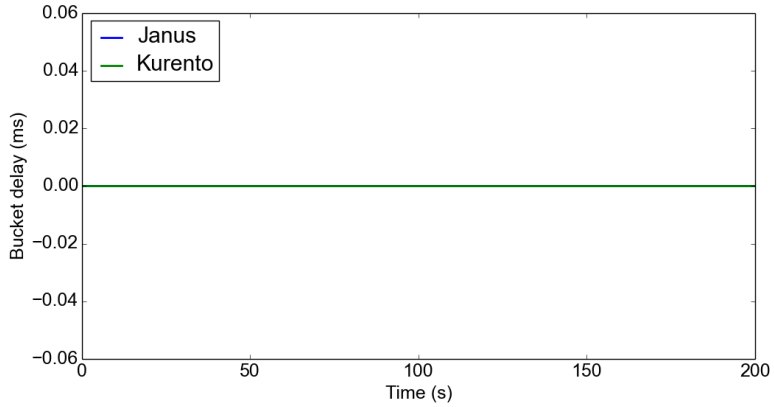


Figure A.10: PLIs sent as function of time, using Janus gateway and Kurento media server in low video setting

A.3 Additional Attributes with Outliers Removed

In the plots below are additional QoE-relevant attributes as function of time over different video settings. However, in all the tests the values was always zero (or close to zero) for the whole span.

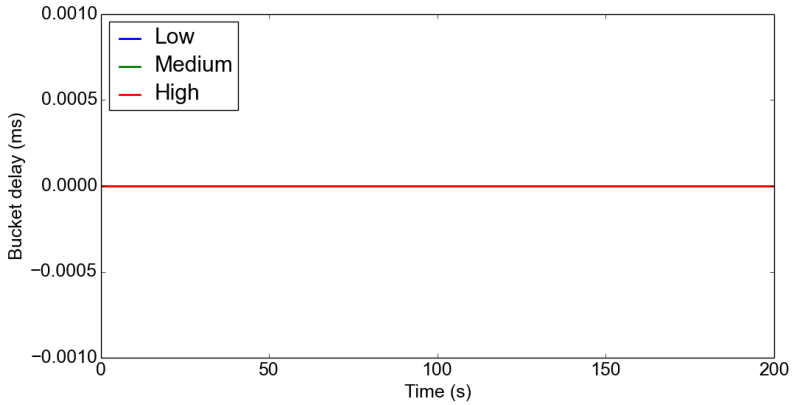


Figure A.11: Bucket delay as a function of time, using Janus gateway in different video settings

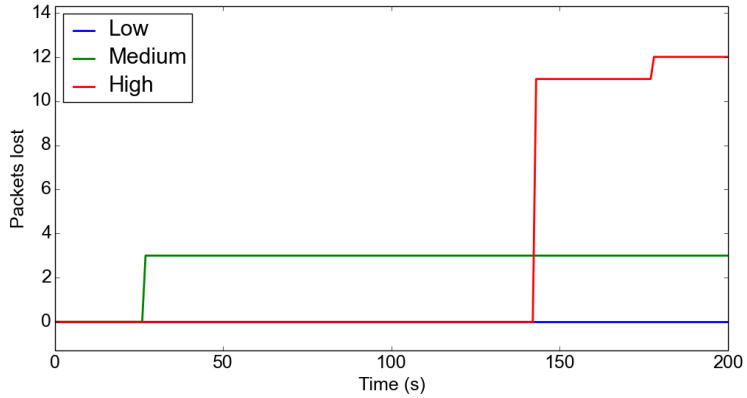


Figure A.12: Packets lost as a function of time, using Janus gateway in different video settings

A.4 Attributes with Outlier Highlights

The graphs show how the data was shown before the outliers were removed. The grey highlights show at what t we removed the data point.

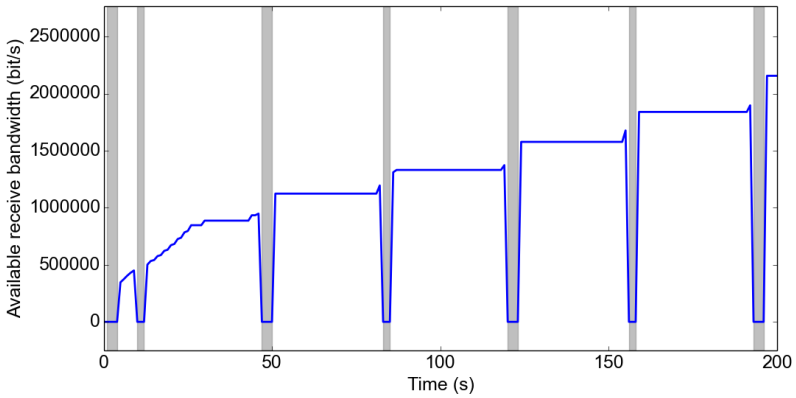


Figure A.13: Available receive bandwidth as a function of time, using Janus gateway in low video setting

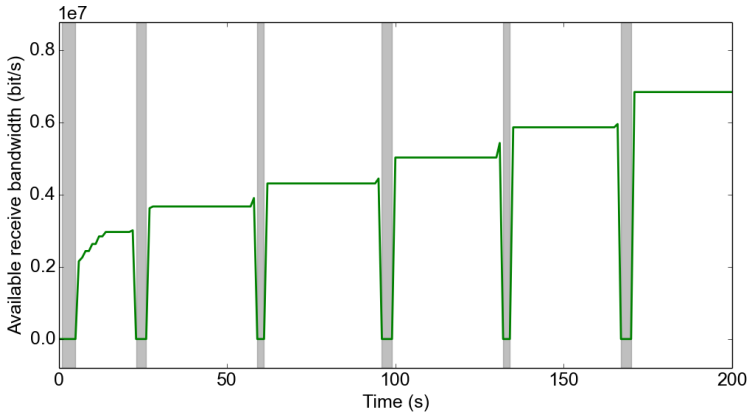


Figure A.14: Available receive bandwidth as a function of time, using Janus gateway in medium video setting

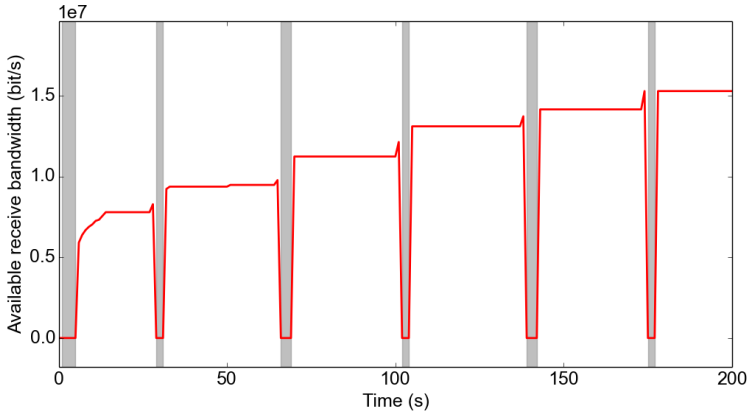


Figure A.15: Available receive bandwidth as a function of time, using Janus gateway in high video setting

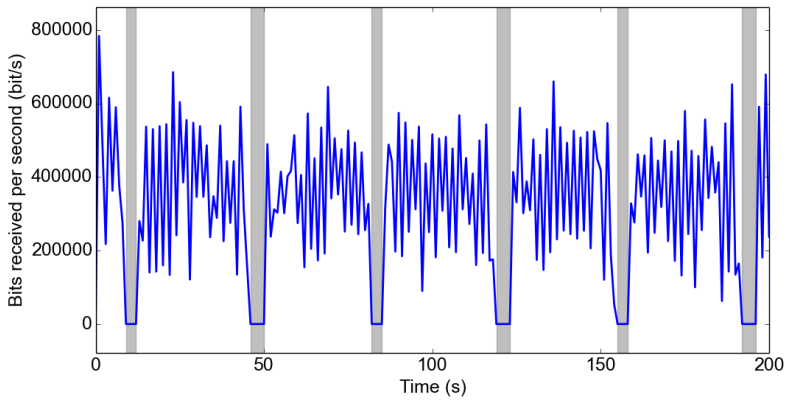


Figure A.16: Bits received per second as a function of time, using Janus gateway in low video setting

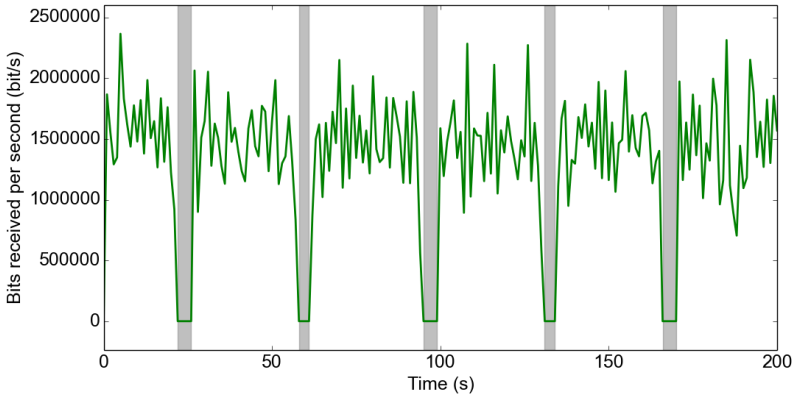


Figure A.17: Bits received per second as a function of time, using Janus gateway in medium video setting

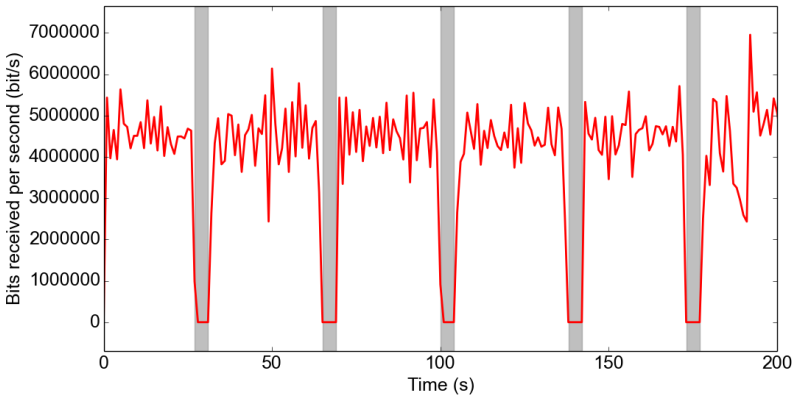


Figure A.18: Bits received per second as a function of time, using Janus gateway in high video setting

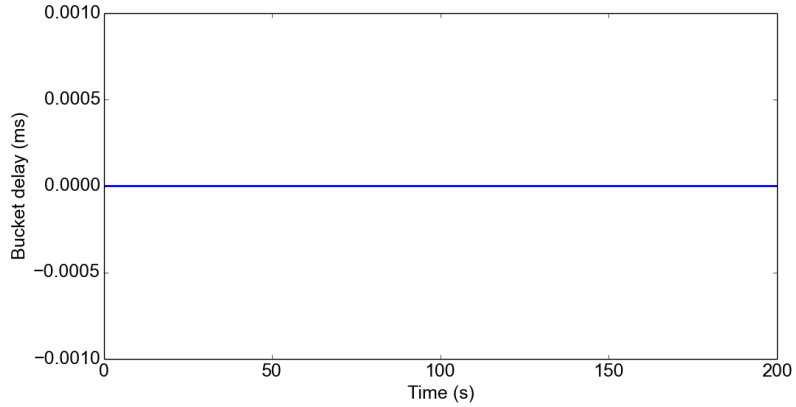


Figure A.19: Bucket delay as function of time, using Janus gateway in low video setting

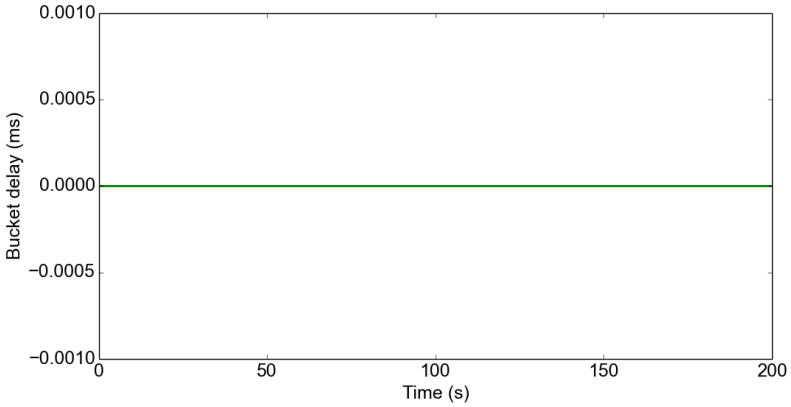


Figure A.20: Bucket delay as function of time, using Janus gateway in medium video setting

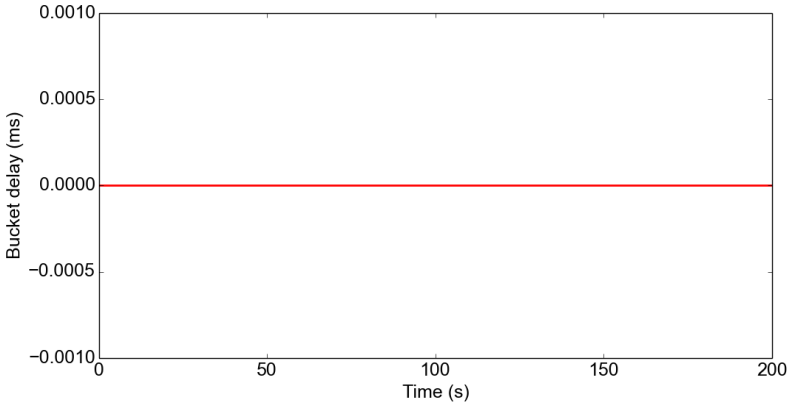


Figure A.21: Bucket delay as function of time, using Janus gateway in high video setting

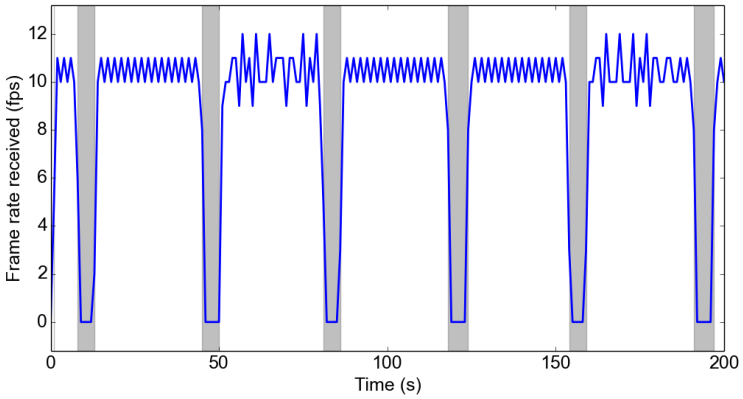


Figure A.22: Frame rate received as function of time, using Janus gateway in low video setting

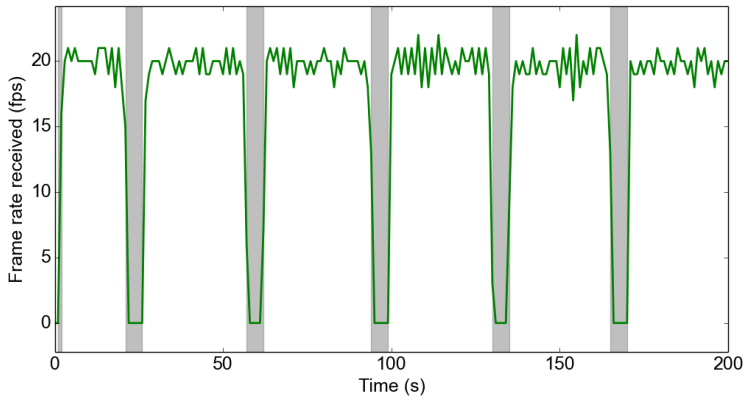


Figure A.23: Frame rate received as function of time, using Janus gateway in medium video setting

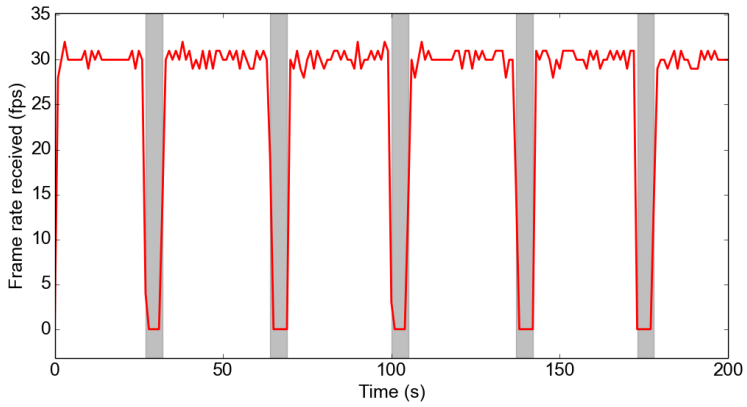


Figure A.24: Frame rate received as function of time, using Janus gateway in high video setting

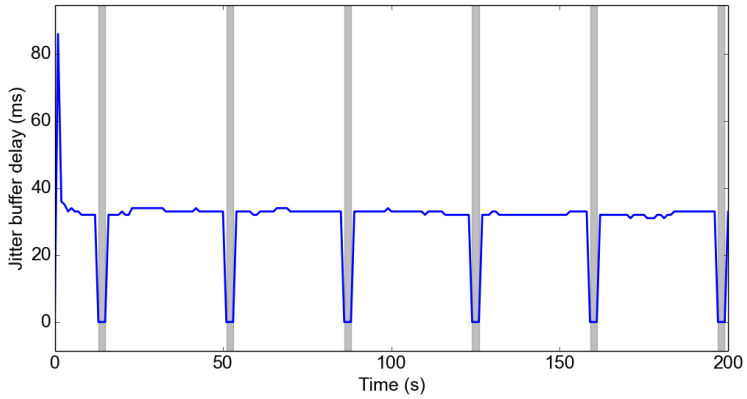


Figure A.25: Jitter buffer delay as function of time, using Janus gateway in low video setting

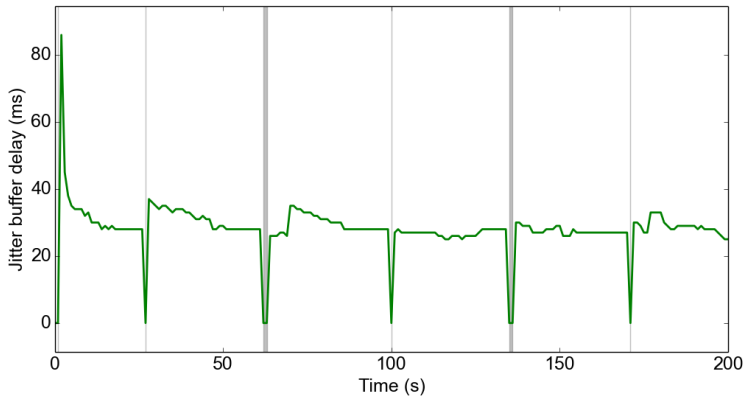


Figure A.26: Jitter buffer delay as function of time, using Janus gateway in medium video setting

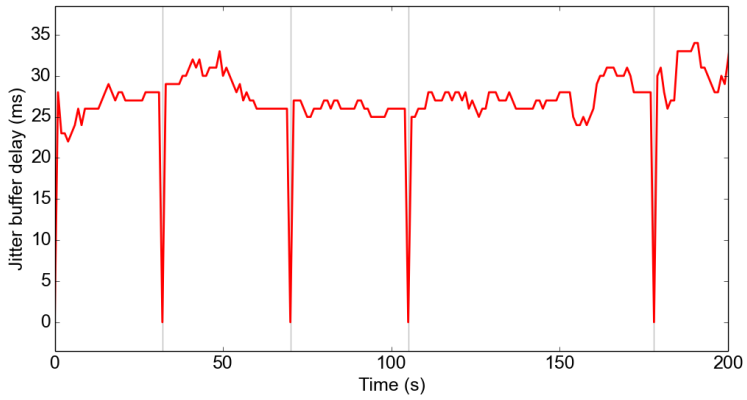


Figure A.27: Jitter buffer delay as function of time, using Janus gateway in high video setting

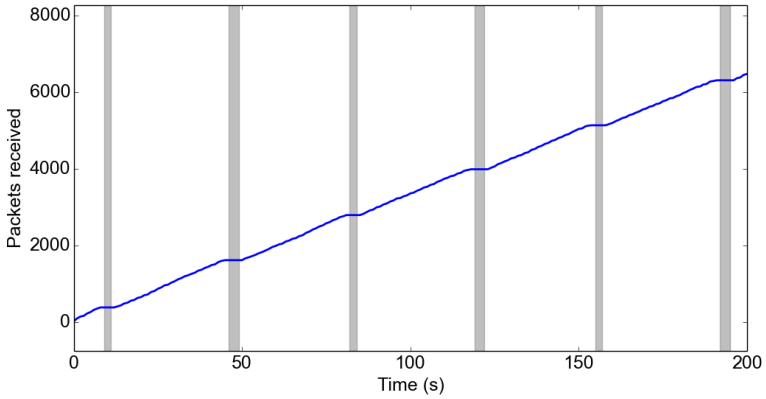


Figure A.28: Packets received as function of time, using Janus gateway in low video setting

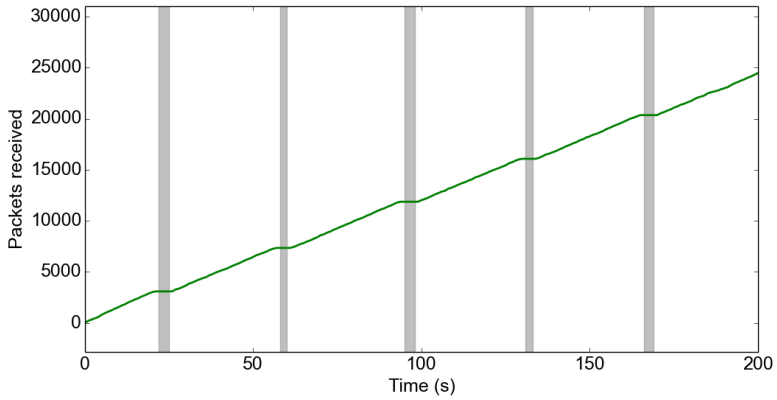


Figure A.29: Packets received as function of time, using Janus gateway in medium setting

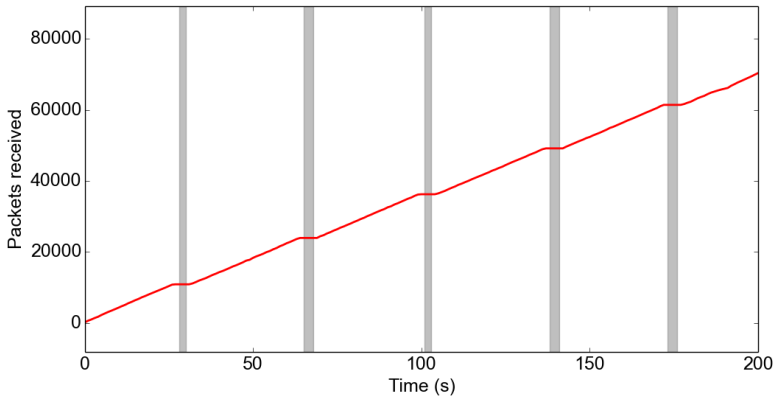


Figure A.30: Packets received as function of time, using Janus gateway in high video setting

A.5 z-Score Threshold Calculation on Janus Gateway (Low Video Setting) with Location-Scale Model

In this section, we demonstrate where T_{z_i} is located for the rate attributes r_t. The plots are results of the tests done on the Janus gateway (low settings). The red line is T_{z_i}; the green line is the median.

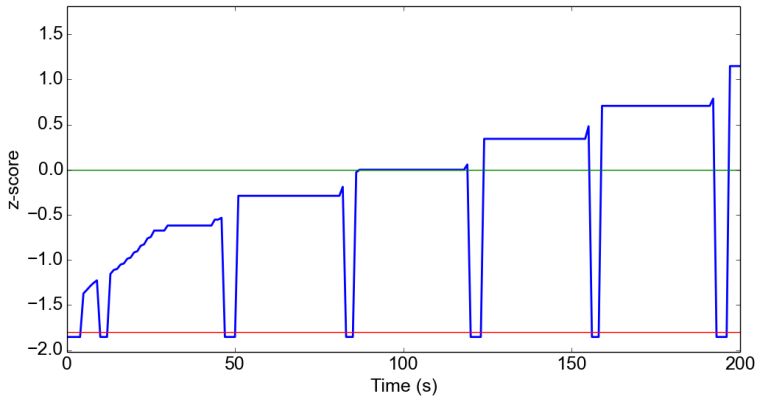


Figure A.31: z-score for available receive bandwidth as function of time, using Janus gateway in low video setting. Includes function median and lower z-score threshold

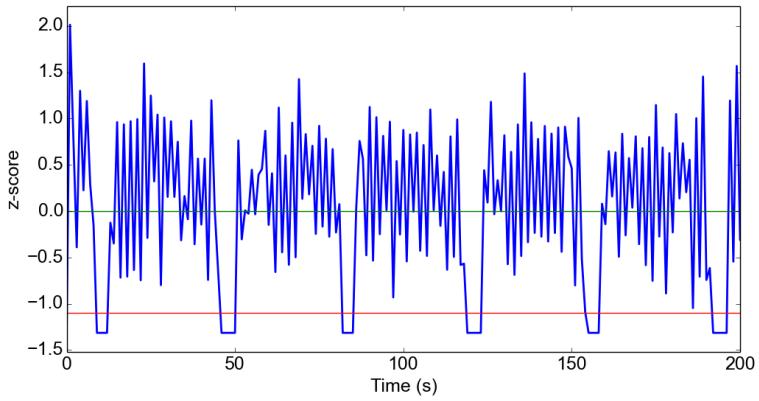


Figure A.32: z-score for bits received per second as function of time, using Janus gateway in low video setting. Includes function median and lower z-score threshold

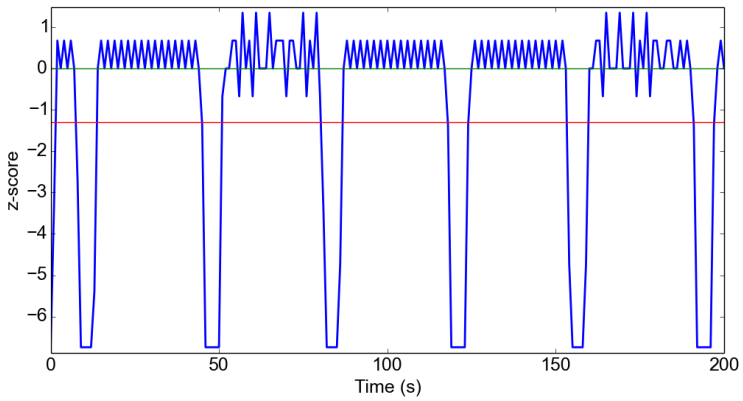


Figure A.33: z-score for frame rate received as function of time, using Janus gateway in low video setting. Includes function median and lower z-score threshold

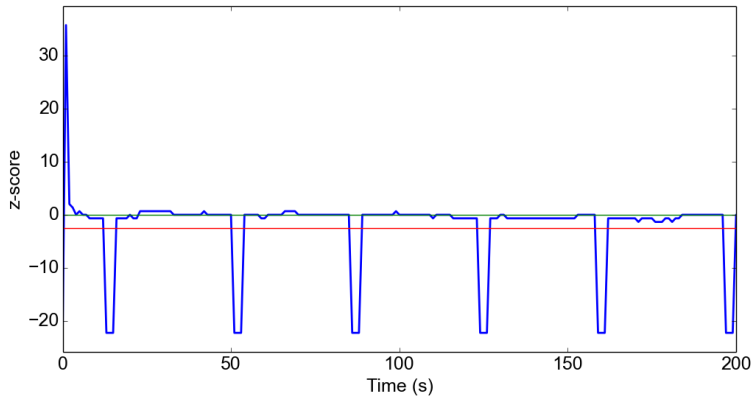
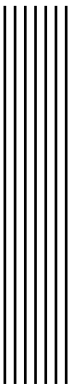


Figure A.34: z-score for jitter buffer delay as function of time, using Janus gateway in low video setting. Includes function median and lower z-score threshold



Bibliography

[1] H. Alvestrand and V. Singh. "Identifiers for WebRTC's Statistics API". In: *W3C Working Draft* (Feb. 2015). (Visited on 03/27/2018).

[2] A. Amirante, T. Castaldi, L. Miniero, and S.P. Romano. "Janus: A General Purpose WebRTC Gateway". In: *Proceedings of the Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm)*. Chicago, Illinois, 2014, 7:1–7:8.

[3] A. Amirante, T. Castaldi, L. Miniero, and S.P. Romano. "Jattack: A WebRTC Load Testing Tool". In: *Proceedings of the Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm)*. Oct. 2016, pp. 1–6.

[4] D. Ammar, K. De Moor, M. Xie, M. Fiedler, and P. Heegaard. "Video QoE Killer and Performance Statistics in WebRTC-Based Video Communication". In: *Proceedings of the IEEE Sixth International Conference on Communications and Electronics (ICCE)*. Ha Long, Vietnam, July 2016, pp. 429–436.

[5] G. Berndtsson, M. Folkesson, and V. Kulyk. "Subjective Quality Assessment of Video Conferences and Telemeetings". In: *Proceedings of the International Packet Video Workshop (PV)*. Munich, Germany, May 2012, pp. 25–30.

[6] R. Bestak and J. Hlavacek. "A Videoconferencing System Based on WebRTC Technology". In: *Computer Networks*. Vol. 718. China: Springer, Cham, 2017, pp. 245–255.

[7] K. Brunnström, S.A. Beker, K. De Moor, A. Dooms, S. Egger, M-N Garcia, T. Hossfeld, S. Jumisko-Pyykkö, C. Keimel, and M-C Larabi et al. "Qualinet White Paper on Definitions of Quality of Experience". In: *Qualinet White Paper on Definitions of Quality of Experience Output from the fifth Qualinet meeting*. 2013.

[8] K. T. Chen, C. C. Tu, and W. C. Xiao. "OneClick: A Framework for Measuring Network Quality of Experience". In: *Proceedings of IEEE INFOCOM*. Apr. 2009, pp. 702–710.

[9] L. De Cicco, S. Mascolo, and V. Palmisano. "Skype Video Responsiveness to Bandwidth Variations". In: *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*. Jan. 2008, pp. 81–86.

[10] M. Claypool and J. Tanner. "The Effects of Jitter on the Peceptual Quality of Video". In: *Proceedings of the ACM International Conference on Multimedia*. Orlando, Florida, USA, 1999, pp. 115–118.

[11] S. Counts and M. De Choudhury. "Understanding Affect in the Workplace via Social Media". In: *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*. ACM, Feb. 2013, pp. 303–316.

[12] Digium. *Asterisk*. 2018. URL: https://www.asterisk.org/ (visited on 05/31/2018).

[13] S. Dutton. *Getting Started with WebRTC*. 2014. URL: https://www.html5rocks.com/en/tutorials/webrtc/basics/#toc-rtcdatachannel (visited on 03/12/2018).

[14] L.L. Fernández, M.P. Díaz, R. Benítez Mejías, F.J. López, and J.A. Santos Naevatec. "Catalysing the Success of WebRTC for the Provision of Advanced Multimedia Real-Time Communication Services". In: *Proceedings of the International Conference on Intelligence in Next Generation Networks (ICIN)*. Oct. 2013, pp. 23–30.

[15] M. Fiedler, T. Hossfeld, and P. Tran-Gia. "A Generic Quantitative Relationship between Quality of Experience and Quality of Service". In: *IEEE Network* 24.2 (Mar. 2010), pp. 36–41.

[16] B. García, L. López-Fernández, M. Gallego, and F. Gortázar. "Testing Framework for WebRTC Services". In: *Proceedings of the EAI International Conference on Mobile Multimedia Communications (ICST)*. Brussels, Belgium, 2016, pp. 40–47.

[17] B. Goode. "Voice over Internet protocol (VoIP)". In: *Proceedings of the IEEE* 90.9 (Sept. 2002), pp. 1495–1517.

[18] Peisong H., Xinghao J., Tanfeng S., and Shilin W. "Detection of Double Compression in MPEG-4 Videos Based on Block Artifact Measurement". In: *Neurocomputing* 228 (2017), pp. 84–96.

[19] V. Krishnamoorthi, N. Carlsson, and E. Halepovic. "Slow but Steady: Cap-based Client-Network Interaction for Improved Streaming Experience". In: *Proceedings of the IEEE/ACM International Symposium on Quality of Service (IWQoS)*. Banff, Canada, June 2018.

[20] Kurento. *Interoperating WebRTC and IP cameras*. Sept. 2015. URL: https://www.kurento.org/blog/interoperating-webrtc-and-ip-cameras (visited on 05/22/2018).

[21] I. Leftheriotis and M.N. Giannakos. "Using Social Media for Work: Losing Your Time or Improving Your Work?" In: *Computers in Human Behavior* 31 (Feb. 2014), pp. 134–142.

[22] T. Levent-Levi. *What Do the Parameters in WebRTC-Internals Really Mean?* Dec. 2016. URL: https://testrtc.com/webrtc-internals-parameters/ (visited on 05/16/2018).

[23] L. López, M. París, S. Carot, B. García, M. Gallego, F. Gortázar, R. Benítez, J.A. Santos, D. Fernández, R.T. Vlad, I. Gracia, and F.J. López. "Kurento: The WebRTC Modular Media Server". In: *Proceedings of the ACM Multimedia Conference*. Amsterdam, The Netherlands, 2016.

[24] Medooze. *MCU*. 2018. URL: http://www.medooze.com/products/mcu.aspx (visited on 02/14/2018).

[25] L. Miniero. *janus_streaming.c File Reference*. 2018. URL: https://janus.conf.meetecho.com/docs/janus__streaming_8c.html (visited on 05/09/2018).

[26] Grandstream Networks. *GDS3710, Video Door System*. 2018. URL: http://www.grandstream.com/products/physical-security/video-door-systems/product/gds3710 (visited on 02/13/2018).

[27] 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)*. July 2006. URL: https://tools.ietf.org/html/rfc4585#section-6.3.1 (visited on 05/23/2018).

[28] U. Paščinski, J. Trnkoczy, and V. Stankovski et al. “QoS-Aware Orchestration of Network Intensive Software Utilities Within Software Defined Data Centres”. In: *J. Grid Computing* 16.1 (Mar. 2018), pp. 85–112.

[29] A. Perkis, S. Munkeby, and O. I. Hillestad. “A Model for Measuring Quality of Experience”. In: *Proceedings of the Nordic Signal Processing Symposium (NORSIG)*. June 2006, pp. 198–201.

[30] T. De Pessemier, K. De Moor, W. Joseph, L. De Marez, and L. Martens. “Quantifying the Influence of Rebuffering Interruptions on the User’s Quality of Experience During Mobile Video Watching”. In: *IEEE Transactions on Broadcasting* 59.1 (Mar. 2013), pp. 47–61.

[31] M. H. Pinson and S. Wolf. “A New Standardized Method for Objectively Measuring Video Quality”. In: *IEEE Transactions on Broadcasting* 50.3 (Sept. 2004), pp. 312–322.

[32] P.J. Rousseeuw and M. Hubert. “Robust Statistics for Outlier Detection”. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 1.1 (2011), pp. 73–79.

[33] M. Schmitt, S. Gunkel, P. Cesar, and D. Bulterman. “Asymmetric Delay in Video-Mediated Group Discussions”. In: *Proceedings of the International Workshop on Quality of Multimedia Experience (QoMEX)*. Sept. 2014, pp. 19–24.

[34] M. Schmitt, S. Gunkel, P. Cesar, and P. Hughes. “A QoE Testbed for Socially-aware Video-mediated Group Communication”. In: *Proceedings of the International Workshop on Socially-aware Multimedia (SAM)*. Barcelona, Spain, 2013, pp. 37–42.

[35] J. Skowronek, K. Schoenenberg, and G. Berndtsson. *Multimedia Conferencing and Telemeetings*. Ed. by S. Möller and A. Raake. Springer, 2014. Chap. 15, pp. 213–228.

[36] Doubango Telecom. *webrtc2sip - Smart SIP and Media Gateway to connect WebRTC endpoints*. 2018. URL: `https://www.doubango.org/webrtc2sip/` (visited on 02/14/2018).

[37] M. Venkataraman and M. Chatterjee. “Inferring Video QoE in Real Time”. In: *IEEE Network* 25.1 (Jan. 2011), pp. 4–13.

[38] M. Venkataraman, M. Chatterjee, and S. Chattopadhyay. “Evaluating Quality of Experience for Streaming Video in Real Time”. In: *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)*. Nov. 2009, pp. 1–6.

[39] D. Vucic and L. Skorin-Kapov. “The Impact of Mobile Device Factors on QoE for Multi-Party Video Conferencing via WebRTC”. In: *Proceedings of the International Conference on Telecommunications (ConTEL)*. July 2015, pp. 1–8.

[40] W. Wu, A. Arefin, R. Rivas, K. Nahrstedt, R. Sheppard, and Z. Yang. “Quality of Experience in Distributed Interactive Multimedia Environments: Toward a Theoretical Framework”. In: *Proceedings of the ACM International Conference on Multimedia*. Beijing, China, 2009, pp. 481–490.

[41] Yang Xu, Chenguang Yu, Jingjiang Li, and Yong Liu. “Video Telephony for End-consumers: Measurement Study of Google+, iChat, and Skype”. In: *Proceedings of the Internet Measurement Conference (IMC)*. Boston, Massachusetts, USA, 2012, pp. 371–384.

[42] Y. Connie Yuan, X. Zhao, Q. Liao, and C. Chi. “The Use of Different Information and Communication Technologies to Support Knowledge Sharing in Organizations: From E-Mail to Micro-Blogging”. In: *Journal of the Association for Information Science and Technology* 64.8 (Aug. 2013), pp. 1659–1670.