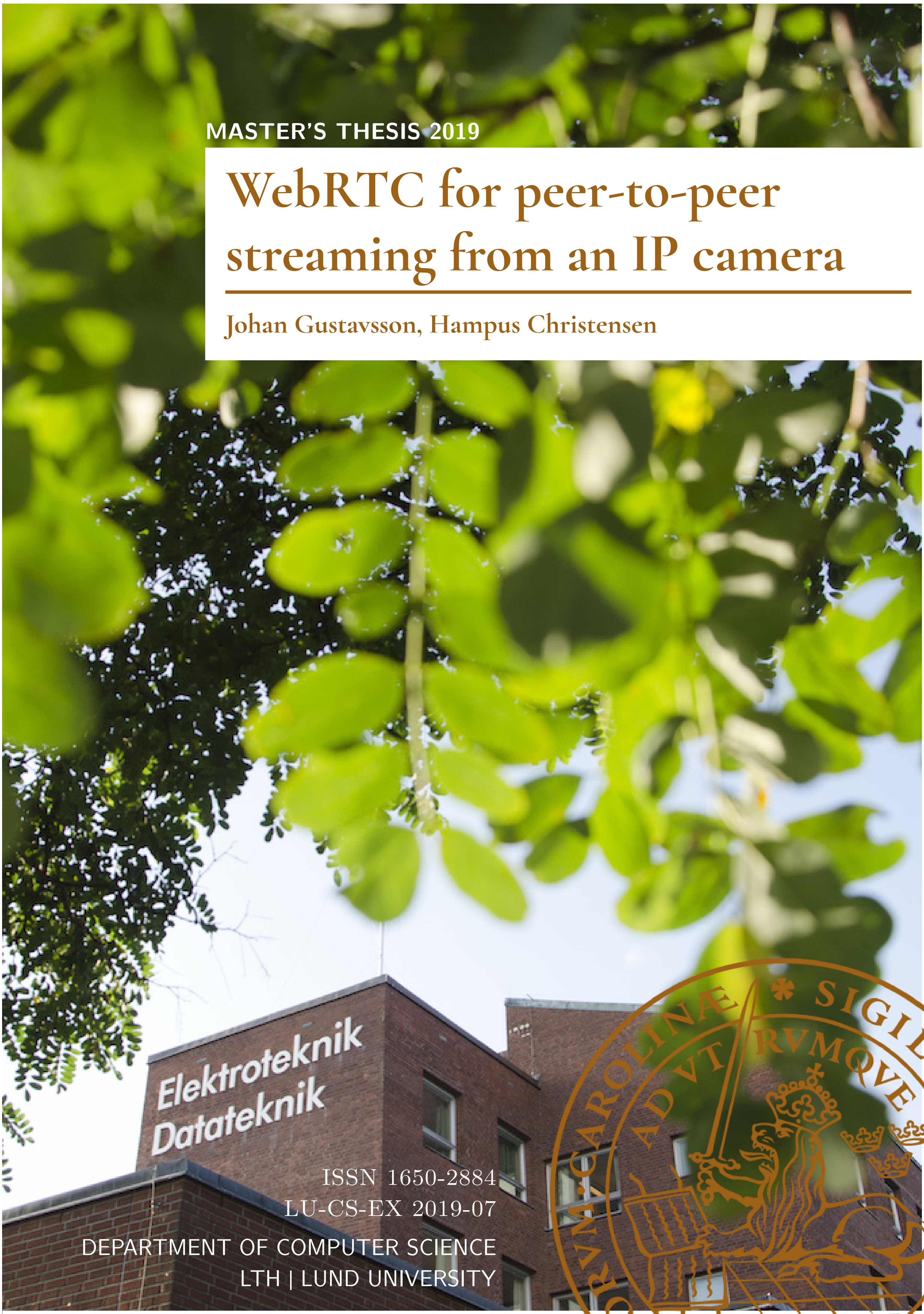


MASTER'S THESIS 2019

WebRTC for peer-to-peer streaming from an IP camera

Johan Gustavsson, Hampus Christensen

ISSN 1650-2884
LU-CS-EX 2019-07
DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2019-07

WebRTC for peer-to-peer streaming from an IP camera

Johan Gustavsson, Hampus Christensen

WebRTC for peer-to-peer streaming from an IP camera

Johan Gustavsson
`johan_gustavsson@tutanota.com`

Hampus Christensen
`hampusichristensen@gmail.com`

June 19, 2019

Master's thesis work carried out at Axis Communications.

Supervisors: Mathias Haage, `mathias.haage@cs.lth.se`
Jesper Forsberg, `jesper.forsberg@axis.se`
Svante Richter, `svante.richter@axis.se`

Examiner: Flavius Gruian, `flavius.gruian@cs.lth.se`

Abstract

Tunnelling video from surveillance IP-cameras through cloud services has both high monetary cost and latency. Streaming the video peer-to-peer can eliminate both these factors while upholding quality of service. WebRTC is an open web standard, streaming API which enables browsers and devices to communicate peer-to-peer without any plugins. While earlier IP-camera implementations have been dependent on intermediate servers, this thesis investigates the possibilities of moving the intermediate server to the camera. Focus is on device performance with regards to CPU and memory usage, network throughput, latency and capacity. A WebRTC server/gateway called Janus was used, installed on two different current generation Axis IP-cameras. The results showed that our WebRTC solution performed comparably to a RTSP (Real Time Streaming Protocol) over WebSocket based one. This does not only pave the way for more implementations built upon WebRTC for IP-cameras, but for all embedded devices.

Keywords: WebRTC, IP-Camera, peer-to-peer, streaming, Janus

Acknowledgments

We would like to thank Svante Richter & Jesper Forsberg as well as Julian Kroné & Ahmed Hulo for all assistance and support during our time at Axis.

A huge thanks to Johan Fagerström for his excellent help with cross compilation and overall rewarding discussions. Also thanks to Thomas Glad for giving us the opportunity to perform this thesis.

Finally, we would like to thank our supervisor from Faculty of Engineering, LTH, Mathias Haage.

Contents

Chapter 1
Introduction

Surveillance cameras have proven to be effective in preventing crime and there is no doubt that the world embraces this fact [1]. Axis and Hikvision, two of the largest surveillance IP-camera producers, are both showing steady growth over the last few years [2] [3].

A common surveillance method is to have alarm centrals monitor many cameras placed in multiple locations, by communicating with the cameras over the internet. This video stream is often, at least by Axis, tunnelled through a cloud service which is both expensive and complicated to setup for the provider. Even if streaming video through a cloud service works, there would be many advantages if the video could be streamed directly from the camera to the viewer.

WebRTC (Web Real-Time Communications) is a free, open web standard that provides browsers and mobile applications with RTC capabilities via simple APIs. It enables the client to send and receive media using only a modern browser without any plugins and with self signed certificates. Compared to other solutions where the two peers must trust each other, requiring signed certificates to be placed on all cameras, both peers must only trust the same server. In the case of WebRTC that trusted server is the signalling server which is covered further in Section 2.1. After performing a key exchange, which is used to encrypt the traffic peer-to-peer, the peers are no longer dependent on the signalling server.

The aim for this thesis is to investigate the use of WebRTC for peer-to-peer streaming from an embedded device, pictured in Figure 1.1. The focus will be to evaluate on-camera performance compared to other, competing technologies.

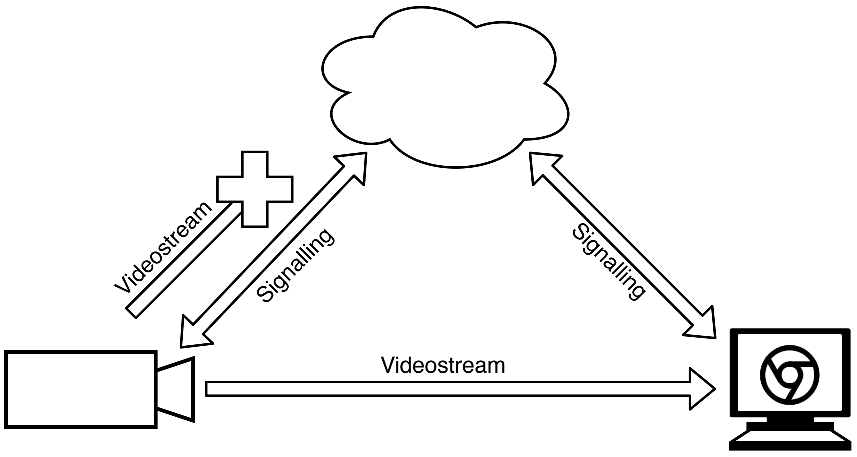


Figure 1.1: The solution using WebRTC, where video is not streamed through any intermediate server, but directly to a client.

1.1 Related Work

Earlier work by Bih Fei Jong has investigated peer-to-peer video streaming using WebRTC, he even used Axis cameras [4]. The system used by Bih Fei Jong had a physical intermediate server transcoding the video, while relaying RTSP (Real Time Streaming Protocol) (Section 2.2) from the camera and WebRTC to the client. The physical server was running a WebRTC server called Janus [4]. This approach differs from ours since the physical server between the camera and client is what we want to eliminate. We want the WebRTC server running on the IP-camera, not on any intermediate device. Aside from that, Bih Fei Jong's work is very relevant for our study, mostly since he used Janus, which is also used in this thesis.

In 2014, two students performed a thesis similar to ours [5]. They did mention WebRTC but as it was in its very early stages, they never used it. The goal of their work however, was very similar to ours. They wanted to implement peer-to-peer video streaming from a camera to a web browser without any extra plugins. Since then, a lot has happened with both WebRTC and video codec support (H.264, H.265, VP8, VP9) in browsers which paves way for this thesis.

Since the camera most likely will be behind various NATs (Network Address Translation) and strict firewalls, it is relevant to examine networking. Using WebRTC in such networks has been examined by A. Johnston et al. [6]. They describe issues with NAT-traversal as well as strict firewall policies and propose solutions. With TURN and/or STUN (Section **??**) servers a peer-to-peer connection can be established by using the Interactive Connectivity Establishment (ICE) algorithm (Section **??**). Even if networking isn't the focus of this thesis, this work is still important to ensure the relevancy of WebRTC streaming in this context.

In order to evaluate how well a WebRTC server running on an IP-camera performs under heavy load, the solution needs to be stress tested. The team behind the WebRTC server Janus performed stress tests on said server multiple times [7] [8]. The second time, they created a tool called Jattack [9]. In these tests, performed on high performance servers consisting of eight 4-core CPUs, Janus was able to distribute one video stream to 800 viewers before losing frames. We did not use Jattack because it is licensed. Also, since the tests will be run on a vastly slower device, the number of clients will not get close to the numbers seen in the Jattack tests.

The same team recently released a report where a long known issue in the library `libnice` was examined [10] in regards to its performance impact on Janus. `libnice` is an implementation of IETF's Interactive Connectivity Establishment (ICE) stack. The issue, which was affecting `libnice's` global lock contention, was patched with an unreleased patch and the performance differences were evaluated in a streaming scenario using above mentioned Jattack. Besides the patch a third alternative was tested, a custom built implementation of the ICE stack, named Jice. Both the patched version and Jice performed much better than the, at that time, master branch `libnice`. When streaming video, Janus was able to serve about 50% more clients by having higher CPU utilization. The patch to `libnice` was merged and released in late December 2018 with version 0.1.15.

1.2 Research Objectives

What has not yet been thoroughly studied is if a WebRTC server run well on embedded devices, in our case an IP-camera. Most tests have been performed on either server grade hardware or consumer PCs. The embedded devices have limited resources and performance which can stop modern solutions, such as a WebRTC server, from functioning as intended. The questions that this thesis aims to answer are:

- Is it possible to run a WebRTC server on an IP-camera?
- Does WebRTC perform well enough to be used in the defined use cases (Section 1.4)?
- How does WebRTC compare to similar streaming protocols, such as RTSP over WebSocket?

1.3 Method

To investigate the research objectives, a prototype WebRTC server will be constructed and installed on an IP-camera. The prototype will then be tested and evaluated in regards to **performance**, **capacity** and **deployability**. The WebRTC solution will then be compared to a RTSP over WebSocket solution that Axis currently has implemented. As a baseline comparison, the built in RTSP server will also be used.

Performance will be evaluated by comparing CPU and RAM usage on the cameras, network usage between the peers, number of dropped frames, latency and stream startup times between the WebRTC prototype and the current RTSP over WebSocket solution.

Capacity will be evaluated by examining the performance (as described above) of the WebRTC prototype when additional live video streams are streamed at the same time. The prototype will be tested with different number of concurrent video streams. The threshold at which the service is viable will be sought.

Deployability will be evaluated by examining camera hardware and camera software to see if it can support a WebRTC solution. How many generations of cameras that could potentially support this solution will also be examined.

The requirements that are put on the client will also be evaluated in the sense of what browsers and plugins that are necessary for the implementation to work.

Since the focus is on camera performance and the solution will be tested on a local network, we will not implement the signalling required to establish over internet peer-to-peer connections. The performance measured on the camera will be essentially the same regardless if the video is streamed on a local network or over the internet.

1.4 Use Cases

WebRTC was designed for many different applications, for example audio/video calling, chat rooms and even gaming. In our case of using WebRTC on IP-cameras, all the current applications do not directly fit our problem. Therefore we defined four use cases for WebRTC that will help us interpret the results when used together with IP-cameras.

The use cases are all not relevant for how Axis uses their cameras today, but possibly for future implementations. Some limitations are put on the use cases since the media communication with the IP-camera will mostly be unidirectional, but there are exceptions where bidirectional communication is possible.

According to the Axis Product guide, a resolution of 720p is enough to cover most cases [11]. For frame rate the camera defaults to 15 frames per second. Because of this, these values were used as the baseline for all use cases and in our tests.

1. **Local Network Surveillance:** The local network surveillance case is where the user is located on the same network as the IP-camera and wants to view its video stream. In this case, the number of concurrent streams will be only one or even zero. Most of the time the number of concurrent streams will be zero, therefore idle performance will be very important to consider.

2. **Over Internet Surveillance** This use case is very similar to the local network surveillance, with the exception that the users are not on the same local network. There are also more concurrent streams, about 0-5, since multiple users can connect at the same time. For example, a user at an alarm central can connect if an alarm goes off, while a user on site is streaming from the same camera. Important in this use case is bandwidth, stream startup time and performance with multiple active streams.

3. **Door station** A use case where bidirectional communication will occur, is together with door stations. Door stations, such as the Axis A8105-E, are "IP door bells" equipped with cameras, microphones and speakers. These door stations enables the user to both see and speak to a person at the door, like for example a delivery person. Some IP-cameras can also support audio playback so it is not limited to only door stations. This use case relates very much to a more common WebRTC use case, the video call. In this use case, latency is important since a long latency makes it hard to have a comfortable conversation.

4. **Education/Webinar:** In this case an IP-camera is used to stream classes and presentations live without the need of any external servers. The attendees connect directly to the camera and stream video peer-to-peer. The number of attendees, or concurrent streams, are expected to be 10-100 or even more.

1.5 Contributions

The work was divided as follows:

Related work Related work was investigated by both. The individual focus was put towards related work connected to the divided work as stated below.

Prototype development This was divided so that Hampus focused on the GStreamer pipeline investigation and the prototype before the Janus server was implemented. Johan focused on Janus configurations and the web front-end. The finished prototype was then cross compiled to the cameras by both.

Abstract Written together.

Introduction Mostly written by Johan.

Technologies Mostly written by Hampus.

Prototype Written together.

Experiments This chapter is divided so that Hampus focused on doing the tests and writing about Latency, Network and Stream Startup times. Johan focused on doing the tests and writing about CPU/Memory Usage, Load Average as well as writing the "Test Environment" section.

Evaluation This chapter is divided so Hampus focused on the parts about Latency, Network and Stream Startup times. Johan focused on the parts about CPU/Memory Usage,Load Average and constructed the diagrams.

Discussion Divided the same as for the Evaluation chapter.

Conclusion Written together.

Chapter 2
Technologies

This chapter will give the reader a technical and historical background to the technologies used in this thesis. For a full understanding of the results in this thesis, reading this chapter is recommended.

2.1 WebRTC

WebRTC aims to create a secure media connection between two or more web browsers without needing any additional plugins or other software and was first released in 2011 by Google. By taking inspiration and help from already existing protocols and implemented APIs, WebRTC manages to deliver video and audio streaming directly to and from web browsers over a peer-to-peer connection, with support from all modern browsers [12][8][13]. WebRTC has since its release grown, both in its functionality and its popularity. Popular applications like WhatsApp, Slack and Facebook Messenger use WebRTC for video and voice calls.

W3C's (World Wide Web Consortium) technical report regarding WebRTC was in November 2017 moved to Candidate Recommendation [14]. This means that W3C thinks that the technology and the documentation of it has reached a certain maturity and urges developers to use it [15].

One major benefit that WebRTC (or rather DTLS-SRTP that WebRTC uses) brings, is that a third-party certificate is not needed. As stated in RFC5763: "This specification does not depend on the certificates being held by endpoints being independently verifiable (e.g., being issued by a trusted third party)" [16].

WebRTC implements three different APIs that can be used by browsers and other clients [13].

RTCPeerConnection is the first API and handles the actual connection between the peers. This API performs many things, making real-time communication much simpler for developers and WebRTC users. Among other things, it handles packet loss concealment, noise reduction and bandwidth adaptivity.

The second API is the **MediaStream** API. This API is dedicated for the media that is being handled and transferred.

RTCDataChannel is the third API. This API handles all other data apart from audio and video. This API can be used to help set up the RTCPeerConnection and provides built in security.

Signalling

WebRTC does not implement or recommend any implementation of signalling, which is how two peers find each other and exchange information before setting up a connection (Figure 2.1). It (signalling) is, however crucial to be able to make use of WebRTC.

The signalling server is responsible for discovery, which helps clients learn each others IP-addresses. It is also responsible for exchanging SDPs (Session Description Protocol) [17]. A SDP is used to negotiate a session's parameters, for example video or audio, encryption and video encoding information.

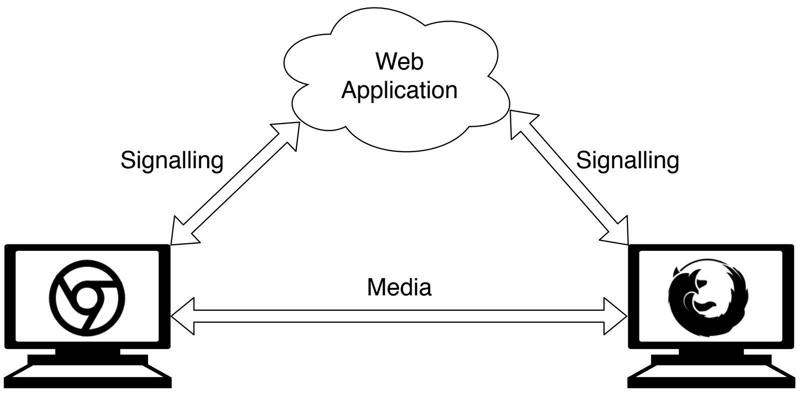


Figure 2.1: WebRTC signalling overview

Privacy and Security

Concerns have been raised regarding the privacy of WebRTC. In 2015 it was discovered that the hidden IP for users using a VPN could easily be retrieved [18].

With WebRTC it is mandatory to use DTLS-SRTP or SDES for key exchange, SRTP for media and DTLS for data channels.

For signalling and other data sensitive communication components and mechanisms, it is mandatory to encrypt the data. When increased security is needed it is up to the developer to implement it.

2.1.1 Other Streaming Standards

There are other alternatives capable of streaming video peer-to-peer worth mentioning apart from WebRTC. We use RTSP (Section 2.2) over WebSocket as a comparison with WebRTC, since this is a standard that is currently used by Axis.

RTSP over WebSocket

Utilising the WebSocket protocol for transport of the RTSP stream is an approach when it comes to live streaming. The WebSocket protocol helps with setting up a bidirectional TCP communication between the client and the server. It is a straight forward approach where the WebSocket connection handles the handshake process (the communication setup) and the data transfer. One cumbersome necessity that RTSP over WebSocket has, is that a signed certificate is required in order to be able to set up an encrypted connection.

The simplicity it brings is its main advantage but also a flaw. A WebSocket approach can not be as customised or optimised, and is not as designed around efficiency and performance as HTTP Live Streaming and WebRTC [19].

HTTP Live Streaming (MPEG-DASH)

HTTP Live Streaming (HLS) or MPEG-DASH is a more developed and refined approach to handle streaming media compared to RTSP over WebSocket [20]. It provides a more flexible framework and offers some more features like adaptive bit rate streaming.

It utilises HTTP which in turn makes it easier to handle, since it counts as HTTP traffic which has less restrictions in firewalls etc. The actual media stream is handled in a Media Playlist that consists of several Media Segments which are part of the media stream. The clients then download the Media Segments one by one. These Media Segments have a default length of ten seconds, but can be shortened. This Media Segment length results in HLS generally having high latency, which we in this thesis aim to avoid since it is not suited for live streaming.

2.2 Transport and Supporting Protocols

Real-time Transport Protocol (RTP)

RTP is an application layer protocol that handles end-to-end transport of data with real-time properties over a network. Some of the many things that RTP includes are payload type identification, sequence numbering, time stamping and delivery monitoring, all very good to have when handling video. If data being sent with RTP is somehow delayed or reordered, the sequence numbering helps with reconstructing the data in the correct sequence.

RTP also utilises the Real Time Control Protocol, RTCP. RTCP handles the monitoring of the communication service and also takes care of relevant information about the peers taking part in the actual communication. [21]

Real Time Streaming Protocol (RTSP)

RTSP is also an application layer protocol which allows extensive control over data being sent in real time. It has support for live data feeds and stored clips. RTSP usually utilises RTP to handle the actual transport of the data.

What makes RTSP different from many other protocols, is that it is not actually tied to a transport protocol. This allows RTSP clients to be able to open and close connections in order to issue new RTSP requests.

The RTSP stream is identified by using textual media identifier. This uses URL (Uniform Resource Locator, which refers to a web resource) in order to refer to the actual stream [22].

2.3 WebRTC Gateway & Server

When two browsers are communicating using WebRTC, e.g. in a video call, both browser are acting as both server and client. When communicating using WebRTC without a browser, some sort of WebRTC server is needed. Most WebRTC servers support relaying media over WebRTC, that media relaying feature is called a gateway. WebRTC gateways create a bridge between legacy infrastructures and WebRTC. Protocols like SIP (Session Initiation Protocol), RTP and RTSP are some examples that most WebRTC gateways support.

2.3.1 Janus

Janus [23] is an open source WebRTC server written in `C` that is designed to be lightweight in its original configuration, but expandable with plugins to suit many needs. It is described by its creators as “a general purpose WebRTC gateway”, since it can be configured in so many different ways. Janus helps with managing WebRTC communications between itself and browsers, between two browsers or managing video conference calls between multiple browsers.

Figure 2.2 describes the modular design of Janus. The core communicates with a browser using a JSON-based protocol over HTTP and through WebRTC PeerConnections, while the plugins provide the core with RTP streams. The core is then responsible of sending the RTP stream via a PeerConnection. Alongside using HTTP, Janus support a handful of other ways to communicate, all modular to keep the server as lightweight as possible. The other ways of communication are WebSocket, RabbitMQ, MQTT, Nanomsg and UnixSockets.

The plugin that is interesting for video streaming is the streaming plugin. With the streaming plugin, Janus is able to relay RTP and RTSP video/audio streams in three different ways. The first is on demand streaming of a server side file, where all viewers stream in their own context. The second is live streaming of server side file, where all viewers see the same stream. The third type is live streaming of media (RTP/RTSP), generated by an external tool like GStreamer or FFMPEG. When live streaming from an external tool, all viewers are watching the same stream.

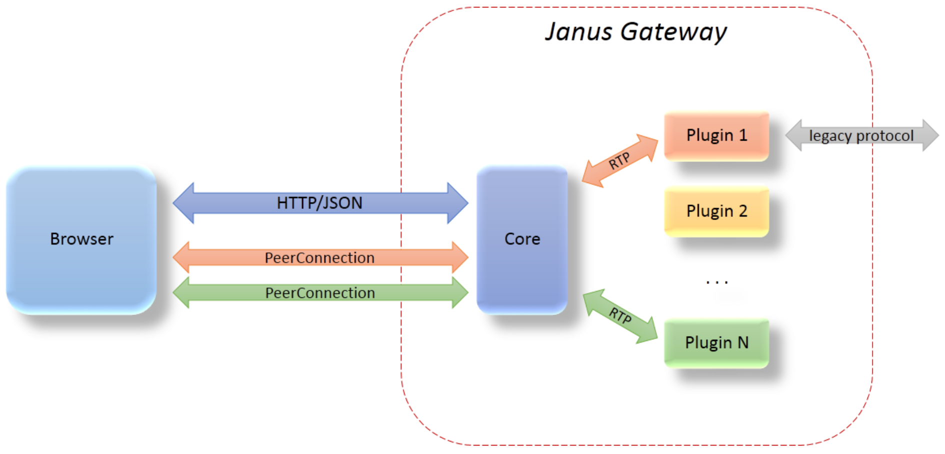


Figure 2.2: Figure from Lorenzo Miniero's presentation at FOSDEM 2016 [24]. The figure describes the extensible architecture of the Janus server

Media streams are added as "mountpoints" and a single Janus server can have multiple mountpoints attached at the same time which users can choose to stream from. RTP streams are added by addressing a port and for RTSP streams, an URL is needed as well as username and password if set. Regarding video codec support, while relaying RTP streams Janus supports anything since the video stream itself is not modified. Instead the limitations are set by WebRTC and the browser. The streaming plugin also exposes an API which can be used to list available streams, get info about the mountpoints and edit/add/remove mountpoints.

Janus also has an Admin API which can list active sessions, show statistics about connections and change server settings. This Admin API is very useful when debugging connection and plugin problems while developing software interacting with Janus.

2.4 Video Compression Standards

A video codec is software that can either encode (compress) video, decode (decompress) video, or both in different formats. This is a key process when streaming video, since reducing the used bandwidth by compressing the data is very important. By having a good video codec, lower bit rates can be achieved while not impacting performance nor picture quality too much.

All the video codecs mentioned below are lossy, which means that quality is lost in the compression and decompression process, making restoration of the exact original source impossible. Lossless video codecs are used in film production and for scientific purposes, where every detail counts and data usage is not as important.

H.264

H.264 was developed by VCEG and released as proprietary software in its first standardised specification in 2003 [25]. H.264 is supported by most browsers and devices (Table 2.1).

H.264 supports a number of different profiles, all with various set of features included. The three most prominent profiles are High, Main and Baseline. These three have in turn extended, constrained and special versions for different bit depth supports. Axis cameras support the three profiles and the user can choose whichever in the cameras settings.

WebRTC does currently only support the Baseline profile, which is the most simple of the three.

Table 2.1: Video Codec Support in browsers using the HTML5 video player[1]

	H.264	VP8	HEVC	VP9
Chrome	X	X		X
Firefox	X	X		X
Safari	X	X		
Opera	X	X		X
Edge	X	X	X	X
Android	X	X	X	X
IOS	X	X	X	

[1] Source: https://en.wikipedia.org/wiki/HTML5_video#Browser_support

VP8

VP8 is a video codec owned by Google that is open source, which probably is a key contributing factor to its popularity. One of the major drawbacks of VP8 is that it is not as common as H.264 when it comes to video systems, leading to worse hardware acceleration support [26].

WebRTC did at first only support VP8, mostly because of it being an open codec. Later, after a long "war", H.264 grew to become the standard [27]. This was mostly due to better hardware acceleration support by devices and support from other streaming standards such as HLS.

VP9 & HEVC

VP9 and HEVC (H.265) is the latest generation of the most popular video codecs. The main difference between the last generations of video codecs and this, is the reduced bit rate. This leads to several improvements such as improved latency and a reduction in bandwidth required [28]. Hardware acceleration support for the HEVC codec in Axis cameras was recently announced with a new chip, the ARTPEC-7 [29].

2.5 Axis specific software

Here we briefly mention Axis specific software that we were limited to using since all tests were run on Axis cameras.

ACAP - AXIS Camera Application Platform

In our work we are only using Axis cameras. Because of this we are restricted to using the way of deploying supported by Axis, ACAP. Cameras using firmware version 5.50 and newer support ACAP. An ACAP is a camera plugin that a user can install on their own at any point. The SDK is available for download on Axis website after signing up for a developer account, accessible for everyone [30].

An ACAP is basically a way to package compiled C-code, or other executables compatible with the cameras, so that it can be installed from a cameras web interface and can interact with the camera. With the ACAP comes an API that exposes various features on the camera, for example it is possible to fetch video and read sensors.

Chapter 3
Prototype

This chapter will cover the goals we had for the prototype, what problems we faced implementing it and how it was implemented.

3.1 Goals

A client should be able to connect directly to the server on the camera with a browser using only JavaScript, no external plugins. The server/gateway should relay a video stream (RTSP) using WebRTC. We have restricted the prototype to only work on local networks, since otherwise a signalling infrastructure has to be set up.

Other demands for the prototype that can be found in the use cases (Section 1.4), are support for multiple clients, from 0 to 5 all the way to the educational/webinar use case that requires up to at least 100 concurrent viewers.

3.2 Stages of Development

Before a WebRTC server was installed and could run on a surveillance camera, we performed multiple preparatory steps to get accustomed to both the WebRTC server and the handling of video streams. All steps were not planned, but since they all add some knowledge about the usage of WebRTC we chose to include them. The five steps are listed below, with the final step being having the WebRTC server running on an IP camera.

1. Choosing a WebRTC Server
2. Creating the Web Front-End
3. Running a Local WebRTC Server Test
4. Running a Server Between Devices
5. Running the Server on the IP-Camera

3.2.1 Choosing a WebRTC Server

There are multiple different WebRTC servers that we could have tried to implement as our prototype (Table 3.1). We chose to not try several different servers or developing our own WebRTC server/gateway, since the focus was on investigating WebRTC in general. Therefore we had to extra careful when choosing the server in regards to compatibility and previous work.

Table 3.1: List of WebRTC servers/gateways

Server	Language	Open Source
Janus	C	✓
Kurento	C++	✓
Jitsi	Java	✓
Pion	Go	✓

So to choose one, three had to be removed. Jitsi fell first because the lack of Java VM on the Axis cameras, the other three were not as easy to choose from. We had to skip Pion. It is an interesting project, but currently lacks support for CPU architectures (CRIS) that many Axis cameras still use. The choice between Janus and Kurento was harder, since both offer mature documentation and functionality. We decided to go with Janus because it offers a design where only the necessary features are compiled. Janus also offers good demos, was used in previous work [4] and by our supervisors in short tests. Janus also has good documentation, large user base and active developers ready to answer questions.

The version of Janus used was v0.6.2.

3.2.2 Creating the Web Front-End

For the WebRTC tests we used HTML, CSS and JavaScript bundled with Janus as their Streaming example page. The page includes a simple video streaming example where the user can connect to a Janus server, list the streams available and choose to stream any of them. This demo and multiple others are hosted at https://janus.conf.meetecho.com/demos.html.

Some modifications were done to the demo front end. A version that connected to the camera instantly, without waiting for a user action, was created to speed up testing. Another modification was the feature to be able to add and remove video mountpoints. This was done to eliminate high idle usage by Janus when connected to a mountpoint which will be discussed later in the idle part of Section 4.5.1.

This was done by adding two functions (listings 3.1 and 3.2) to the JavaScript code. The mount function is called as soon as the page is loaded and connection to Janus is established. The unmount function is called when the user stops the stream by clicking a button.

```
function mountStream() {
  var adminKey = "password";
  var message =  { "request": "create", "admin_key": adminKey,
     "type" : "rtsp", "id" : 1, "description" : "high", "audio"
      : false, "video" : true, "url" : "rtsp://127.0.0.1/axis-
     media/media.amp?streamprofile=high&videopsenabled=1", "
     rtsp_user": "user", "rtsp_pwd" : "pw"};
  streaming.send({"message": message, success: function(result)
     {
    Janus.log(result);
  }});
}
```

Listing 3.1: The function used to mount a stream to the Janus server

```
function unmountStream() {
  var message = {"request" : "destroy", "id" : 1, "secret" : "
     password"};
  streaming.send({"message" : message});
  Janus.log("Destroyed stream");
}
```

Listing 3.2: The function used to unmount a stream from the Janus server

This solution comes with some obvious drawbacks. Firstly, if the browser tab is simply exited by the user the mountpoint is not removed. This could be solved with the help of the event `onbeforeunload`, which happens right before the user leaves the page. Secondly, if two users are connected at the same time and one of them leaves, the stream will be shut down for the other user. Lastly, the user needs to possess the admin password or the feature have to be non-protected.

The best solution would be for this to be solved server side and not through API calls from clients.

3.2.3 Running a Local WebRTC Server Test

In the first step we ran the server and video stream solely on a single PC, without any real camera present. The video stream (H.264) was created with GStreamer [31] and the stream was picked up by the Janus server, which we then could connect to using the browser. The stream created by GStreamer was an RTP stream and the stream was then picked up by the Janus streaming plugin.

Browser Compatibility

Streaming WebRTC video to Firefox never worked in any of our initial tests. This was unexpected since the browser is supposed to support H.264 Baseline streaming. We found that the problem derives from the browsers different implementations of H.264 decoders. Firefox uses an open source variant developed by Cisco [32]. Chrome has, from what we could find, an inhouse built solution.

We received a SDP (Section 2.1) error when initiating a stream and after debugging we found that Firefox rejected the profile-level-id that described the incoming video stream. The *profile-level-id* is a hexadecimal string that looks like this: `0x428014` and consists of three parts which are described below.

profile_idc 0x42 = 66, Baseline profile

profile-iop 0x80 means constraint_set0_flag=1 (so it is Constrained Baseline profile)

level-idc 0x14 == 20 so it is Level 2.0

By simply replacing the *profile-level-id* in the browser side JavaScript files with a string compliant with Firefox, like for example 42e01f. We could then play all of our H.264 Baseline streams. With the *profile-level-id* switched we even managed to make Firefox play H.264 High encoded video, which suggests that the Cisco decoder is more competent than Firefox admits.

What is worth mentioning, is that this error is not tied to Janus in any way. Since Janus does not alter the video stream, Firefox simply does not accept the video format produced by the camera. Janus does however have the feature to override values like the *profile-level-id*, which can be used to solve the problem.

Due to these problems the feature in the front-end to add mount points using the Janus Streaming API (Section 3.2.2) was extended to also modify the *videofmtp* parameter, a mount-point parameter overwriting SDP-values such as *profile-level-id*, if Firefox is used.

3.2.4 Running a Server Between Devices

In this step we had access to an actual camera we could stream video from, so we used a real video stream instead of one created by GStreamer. This setup is the same as the one Bih Fei Jong used in his thesis but without any video transcoding on the server [4]. We could now try to use the streaming plugin in Janus to pick up the RTSP server directly. This did not work the first tries, instead, during this phase we first had to pick up the stream using GStreamer. The GStreamer pipeline that we ended up using can be seen in Listing 3.3. We used GStreamer to redirect the RTSP stream without any real treatment in the pipeline. The only difference is that the stream comes in as TCP packages, which is forced with the “t” in the URL, and leaves as UDP packages.

```
$ gst-launch-1.0 rtspsrc location=
"rtspt://x.x.x.x/axis-media/media.amp?videopsenabled=1"
user-id=xx user-pw=xx ! udpsink clients=0.0.0.0:9002
```

Listing 3.3: GStreamer pipeline with RTSP redirection

We later got the streaming plugin to pick up the RTSP stream directly, as can be seen in Figure 3.1. The problem was that the camera used at that time, was situated on a large lab network at Axis. That network only allowed UDP packages on certain ports, which is also the reason why the streaming only worked with TCP in the example above. Instead of using the cameras on the large network we put the cameras on a small local network with only our computers connected. With this setup the streaming plugin could pick up the RTSP stream and stream it using WebRTC.

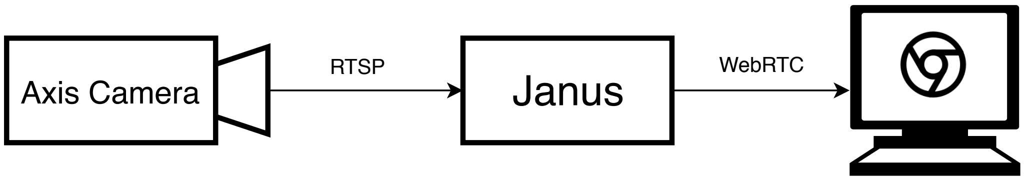


Figure 3.1: Janus placed between the camera and browser on separate device.

That was however not the only problem. Even if the streaming plugin could pick up the stream, it did not mean that the browser could display the video. The streaming plugin has the options to set multiple SDP related values, such as *videortpmap* and *videofmtp* overriding *profile-level-id* and similar fields. We tried changing multiple of these but without any luck. Finally we tried with an option in the cameras plain config, which is an advanced setting menu in the Axis web interface. The option is called *PS Enabled* and can be found under *PlainConfig->Image->H264*.

The option *PS Enabled* is interesting since it places the SPS (Sequence Parameter Set) and PPS (Picture Parameter Set) in the RTP stream. SPS and PPS contain information on a sequence of images and a single rendered image respectively. Without the SPS and PPS the browser does not have sufficient information to decode the video stream. It is also possible enable this option by adding a flag to the RTSP URL, videopsenabled=1.

Sending SPS and PPS in the same RTP over UDP stream as the video can be risky, they should be sent in a separate TCP stream since putting them in the RTP stream makes it more vulnerable to packet loss. This should be investigated further.

3.2.5 Running the Server on the IP-Camera

The next and final step was to run a Janus server on an IP-camera (Figure 3.2). The best way to deliver Janus to a camera and the way Axis wants to use it, is as an ACAP (Section 2.5).

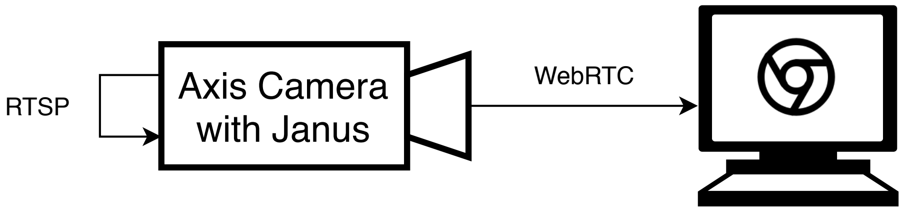


Figure 3.2: Janus running on the IP camera

Since the cameras do not ship with any compiler installed, Janus needs to be cross compiled for the architecture of the cameras we had on hand, armv7hf. Not only does Janus need to be cross compiled, but also its dependencies that are not present on the camera by default. Full list of these dependencies can be seen in Appendix A.1.

Here we ran into unexpected number of issues when trying to cross compile Janus. Several of the dependencies have dependencies themselves. That the cross compilation process also sometimes differed between the different libraries, did not help either. We also had issues with some cross compiled libraries not being able to find and link to other already cross compiled libraries that they required.

We eventually got all dependencies and Janus cross compiled. To minimise the footprint of the Janus server we chose to cross compile Janus with all plugins but streaming disabled, the only plugin needed for three of the use cases (Section 1.4). Besides the support for streaming, support for HTTP transport was also included since it is needed for signalling. See Table 3.2 for full list of plugins. The list can be interesting in order to see what Janus is capable of. For the third use case, door station, the videocall plugin should also be included. It was however never included because the focus was on the surveillance use cases, but performance wise the impact should be almost the same.

Table 3.2: Compiled features of Janus

plugin.streaming	✓
plugin.echotest	
plugin.videocall	
plugin.videoroom	
plugin.sip	
plugin.nosip	
plugin.textroom	
plugin.voicemail	
plugin.audiobridge	
plugin.duktape	
transport.http	✓
transport.rabbitmq	
transport.websockets	
transport.unixsockets	

To package Janus as an ACAP there are a few things that are necessary. First two files need to be created, `package.conf` and `param.conf`. `param.conf` is in our case an empty file, since it is only necessary if the ACAP is to use the built in Apache server. `package.conf` contains information regarding the application, such as name of the executable and how it should be started. The package is then compressed as a `.tar`, renamed to `.eap` and can now be installed on our cameras.

The configuration files for both Janus and ACAP can be found in Appendix B. The final uncompressed folder containing all necessary files used 9.3 MB of disk space.

With this we have answered the first half of our first research question about deployability, as we were able to deploy Janus on an IP-Camera.

Chapter 4
Experiments

In this chapter all tests and tools used to evaluated how well Janus performs are described and motivated. Effort was put into making the tests fair, reproducible as well as realistic.

4.1 Criteria

Data was sought in a number of criteria. These are listed with a reasoning why they are important below. How the data is collected is described in Section 4.3.

Capacity

Number of concurrent streams is relevant since two of the four defined use cases (section 1.4), require it to be possible for more than one user (0-5 in use case 1 and 2, 10-100 in use case 4) to simultaneously be streaming video from the same camera.

Performance

Memory: Memory usage is measured to see how much the WebRTC server solution affects the available memory on the cameras compared to the other solutions.

CPU Usage: The CPU usage is a relevant since it is a good measurement of how stressed the camera is. If the CPU is under heavy load, the system might fail to send frames.

Load Average: Load average represents the average system load during a period of time. Load average is represented as a number N, where a N lower than the number of CPU cores means that on average no process gets queued [33]. A load average above N means that processes are being queued, which could result in performance issues. Therefore a load average above N is not recommended, however, it does not mean that the device is overloaded. Instead it can be used to find issues not related to CPU performance since CPU usage will not show if a process is waiting for IO, which load average does. A high load average together with a low CPU usage would then suggest there being a bottleneck in the IO.

Network: When measuring network, we looked at the current bit rate. The bit rate is mostly impacted by the quality of the video stream. The bit rate is interesting to look at since Janus repacks the RTSP stream. We want to know if it adds any overhead in comparison to a pure RTSP and RTSP over WebSocket stream.

Disk usage: The storage space on the cameras is limited, so keeping the size of the program down is important to make it possible to have it on the cameras. The cameras that we will be running tests on, the M1065 and the Q3518 (Section 4.2.1), had around 25 and 70 MB available respectively. Having room for other applications is something that is preferable, so the application should not use all of the available space.

Stream startup times: Low startup times (0-2 seconds) is very important since the client/user can otherwise miss important information.

4.2 Test Environment

4.2.1 Hardware

In this thesis we used two different cameras (Table 4.1), Axis M1065-LW as well as a more high end camera, Axis Q3518-LVE. We wanted to test cameras with different performance to find if the bottleneck is in Janus, WebRTC or in the hardware. Having a second camera makes this much easier. The versions mentioned in the firmware column are the versions that the prototype has been confirmed working with. The version in bold font is the one used in the tests.

Table 4.1: Camera video specifications.

Name	Max res.	Max framerate	Firmwares
M1065-LW	1920x1080	25/30fps (50/60hz)	**9.10** & 8.40 (LTS)
Q3518-LVE	3840x2160	50/60fps (50/60hz)	**9.10** & 8.40 (LTS)

Axis M1065-LW

The M1065-LW was released in the beginning of 2018 and is one of Axis cheaper cameras with its plastic housing and moderate performance. Its brother M1065-L is basically the same camera, the difference being the L loses WiFi and gains PoE *(Power over Ethernet)*. The M1065-L(W) has an Ambarella S2L which consists of a single core 700Hz Cortex-A9 CPU and 512MB of RAM [34].

Axis Q3518-LVE

The Q3518-LVE was released at about the same time as the M1065-LW and is a higher-end camera. Q3518-LVE costs multiple times more than the M1065-LW, supports up to 2160p/60 and has an ARTPEC-6 chip with a dual core 1GHz Cortex-A9 CPU and 1Gb of RAM.

4.2.2 Network

The network consists of a simple wired local network (Figure 4.1). The normal 100/100 switch is a D-Link DES-1008D and the PoE-switch, which had to be added to support the Q3518, is an Axis Companion Switch 4P.

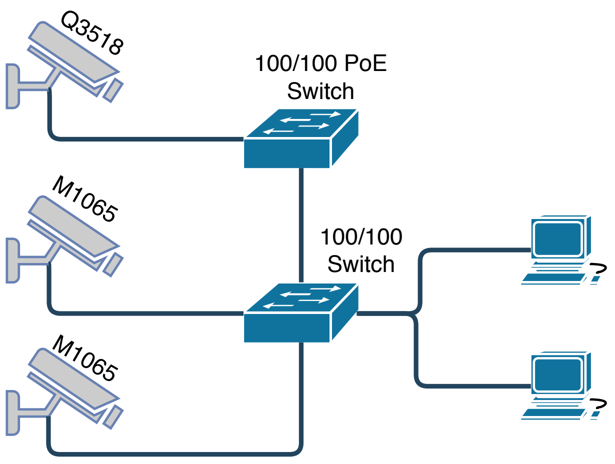


Figure 4.1: Map of network used in testing phase

4.2.3 Stream Profiles

For the testing phase, two different streaming profiles were used, the only difference being the resolution. Only changing the resolution was done to have as few differences as possible between the streaming profiles. Streaming profiles are presets for quality settings that can be created in the cameras web interface. In Table 4.2 the profiles we used are listed. The medium profile represents the most commonly used settings by Axis operators and is considered the baseline in our tests. The high profile utilises the highest resolution the M1065-LW camera can produce and is used as a benchmark.

Both profiles have variable bit rate without any limitations, which means that the bit rate varies depending on what the cameras sees. There was no audio recorded in any tests.

Table 4.2: Stream profiles

Name	Resolution	Frame rate	GOP	Encoding	Compression	Zipstream
High	1080p	15 fps	15	H264 Baseline	30	Low
Medium	720p	15 fps	15	H264 Baseline	30	Low

4.2.4 Camera Positions

The camera were in two different positions for the tests, one to create low bit rate stream and one for high bit rate. The latency test had a separate setup. It is important that the setups does not change during the day or between the tests so the results are even and reproducible. Therefore the cameras were fastened to the desks using adhesive and put visually shielded from the surrounding environment.

Low bit rate - Still Picture

In the first position the camera was pointed at a wall with some post-it notes (Figure 4.2). This represents the easiest case for Janus, since the camera can produce low bit rate that also is very stable. No movement at all can be seen in this position.

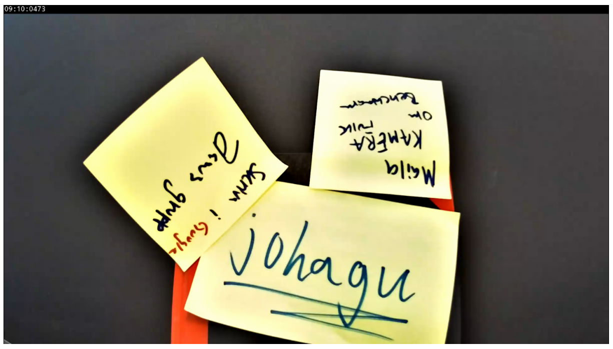


Figure 4.2: Screenshot of the still picture setting used for tests

High bit rate - Moving Picture

In the second setting we pointed the camera at a computer screen , displaying a long, fast and repetitive Youtube video (https://youtu.be/G1IbRujko-A). This test simulates a more exhausting situation. The video is constantly moving fast, making it hard for the encoding algorithm to compress the video, resulting in high bit rates. Worth noting is that since the movement is fast and repetitive, the bit rate is still quite even.

4.2.5 Players

Three different players were used in the testing, one for each streaming method.

WebRTC
For the WebRTC tests we used the Chrome browser, version 73. We settled with using only Chrome in the camera performance tests for its market share and because we only measure camera performance, one browser is enough [35].

RTSP over WebSocket
The current Axis solution uses a browser interface and a video player that streams the RTSP video using WebSocket. The player is reached by entering https://ip in a Chrome browser.

RTSP
VLC (v2.2.2) was used to stream RTSP video. We streamed the video via pure RTSP from a camera using the same URL Janus uses to pick up its stream.

4.3 Data Collection Method

This section describes how the specific data points were collected.

4.3.1 CPU Usage, Load Average and Memory Usage

To measure CPU and memory usage, an Axis tool called Xcam-Memlog was used. The tool is run on an external computer and polls the camera's proc filesystem using ssh every N seconds. The standard setting is N=60, but we used N=30 in our tests to get more data samples. The tool collects different data points, which are used to create visualisations. The various data points and where they are extracted from are described below.

This could have been done by simply reading these values using scripts but Xcam-Memlog simplified the process.

CPU-Usage is read from `/proc/stat`

The three servers each have a process and their names are:

monolith The RTSP server

httpd The RTSP over WebSocket server

janus_server The WebRTC server

Load Average is read from `/proc/loadavg`. There are 3 different values, the averages for the last 1, 5 and 15 minutes. We used the 15 min average because it is the most stable.

Memory Usage is read from `/proc/meminfo`

4.3.2 Stream Stability

We consider a stream unstable when packets are lost multiple times every minute. A singular hiccup once or twice during a 30 minute period does not count as being unstable.

To ensure whether or not a stream is stable we used three different tools, one for each server. See the list below for information regarding the tools. The stream can be unstable in multiple ways, packets can be lost between the server and client, frames can be lost by the camera and the video player can drop frames. Since the tests were run on a simple local network, packets lost are most likely not because of the network, but because of a server not being able to send out the packages. We assume that all packets and frames lost are caused by the server not being able to send them.

WebRTC
: Chrome has a tab accessible on chrome://webrtc-internals/. Here a lot of information about active WebRTC streams can be found. The interesting graphs for packet loss are `googNacksSent` and `packetsLost`.

RTSP over WebSocket
: In the Axis web interface it is possible to see the number of dropped frames. We use this value to determine the stream stability.

RTSP
: In VLC, under (Tools -> Media Information -> Statistics), it is possible to see how many frames VLC has dropped. This includes packets lost.

Aside from the tools, we also considered viewer experience when determining the stream stability.

4.3.3 Network Usage

We used Wireshark to measure network usage [36]. Packets that travel between the camera and the client were captured and reviewed. All packets were captured and then filtered in two different ways. With the first way we measured all packets between the camera and client. When measuring the second way, we only looked at the packets going from the client to the camera. This way we could see how much bandwidth was used to send control messages to the camera, not only how much was used to stream video.

The bit rate was calculated with a built in tool in Wireshark, that can be found under (Statistics -> Protocol Hierarchy). The number used is the bits/s for the "Frame", so all data is included.

4.3.4 Latency

The method for measuring latency was to film a computer screen that was displaying a live feed with a camera. In the video feed we put an overlay that displayed the current time in 1/100 seconds. This allowed us to take screenshots with the computer and then from those screenshots, calculate the latency. An example of said screenshots can be seen in Figure 4.3. Note that since the frame rate of the camera is only 15 fps, accuracy is not in 1/100 seconds.

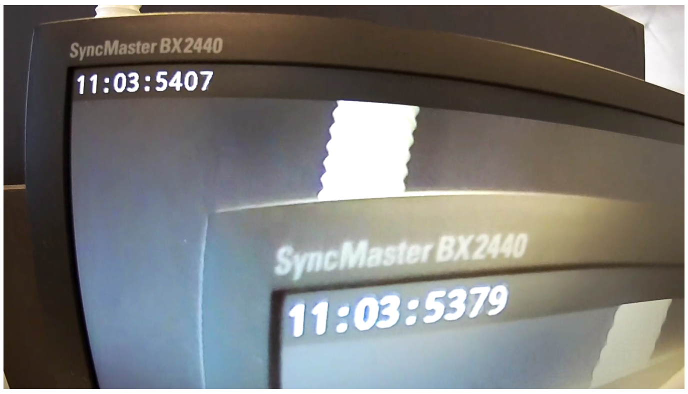


Figure 4.3: Screenshot used to measure latency

4.3.5 Stream Startup Time

Stream startup time is defined as the time until video is visible on the screen after the user has requested a stream, by for example clicking a button. This was tested by manually measuring the time from click to visible video.

4.4 Tests

With the data gathering methods described, the actual tests can be presented. In this section all tests are presented and motivated.

4.4.1 Scenarios

The largest and most important tests are the ones simulating normal usage (use case 1, 2 & 3 Section 1.4). We will call them scenarios. In these tests the medium profile was used and the camera was positioned in the two different positions described above (Table 4.3 for a summary). The tests started with one client connected and streaming video. After 30 minutes another stream was added and so on, until five concurrent streams were active. A test with no connected clients was also done to measure CPU/Memory usage and Load average in idle.

These tests gave an idea of how the different streaming protocols scale in performance when more clients connect. The most useful results are the ones with zero, one and two clients but to gather more data more clients were used as well.

In addition to the scenario tests, longer tests that ran over at least 24 hours were conducted in order to ensure that the solution was robust.

Table 4.3: Summary of usage tests

Target	Profile	No. of Clients
Still (Low bit rate)	Medium	1,2,3,4,5
Moving (High bit rate)	Medium	1,2,3,4,5
Idle (No bit rate)	N/A	0

4.4.2 Latency

The latency test was performed as described above and only done once for each of the tests. The accuracy of our latency test is not optimal, but enough to give us an idea if WebRTC has a significant latency disparity in comparison to RTSP over WebSocket and RTSP.

4.4.3 Stream Startup Times

The method to measure the startup time is described in Section 4.3.5. In order to get more accurate results, we performed the tests five times and took the average value as our result. We have also included the standard deviation for the tests.

4.4.4 Stress Tests

We performed two different stress tests. In the first the medium profile with a still background was used. Here the goal was to see how many concurrent streams the solution could handle, when having a low bit rate. This test is meant to test if the Education/Webinar use case (Section 1.4) is possible.

In the second stress test, the cameras were pushed with a higher quality stream profile and more concurrent clients until they no longer could deliver stable video streams. In our case this means that all streams becomes unstable at the same time, since the same stream is sent to all viewers. Here we also brought in the more advanced camera, Q3518, to be able to see if the camera was the bottleneck.

4.5 Test Results

In this section the results from the tests are presented and explained.

4.5.1 Scenarios

The results from the scenarios are presented below, in rising order of load. The first tests use the medium profile. The high profile is not used until the stress tests, since the high profile is considered being a stress test and the baseline setting is 720p/15 (Section 1.4).

When presenting the cameras' load averages, the recommended maximum (Section 4.1) is marked with a red line, 1 for M1065 and 2 for Q3518.

Idle

It was found that Janus put the camera under relatively heavy load even in idle (Table 4.4). This is because how the Streaming plugin handles its RTSP stream mountpoints. It keeps Janus connected to its RTSP stream even if no clients are connected and streaming video. Therefore we tested the idle performance while having a stream mountpoint connected and also without a mountpoint. Having Janus disabled represents the idle performance for both pure RTSP and RTSP over WebSocket.

Table 4.4: Idle performance

Mode	CPU Usage	Load Average
Janus with mountpoint	34.7%	0.58
Janus without mountpoint	20%	0.5
Janus disabled	18.5%	0.43

Still Frame Tests

In the still frame tests the camera was put under the lightest load, streaming a still picture. In this setting the camera could easily compress the video stream and the bit rate averaged at **300 kbit/s**. At this bit rate the performance difference between the pure RTSP, RTSP over WebSocket and WebRTC was almost none with one active stream.

WebRTC has a small overhead in CPU usage when there is only one active stream (Figure 4.4) and the load average results (Figure 4.5), are very close to being equal. RTSP over WebSocket scales a bit worse than the other two when there are more active streams.

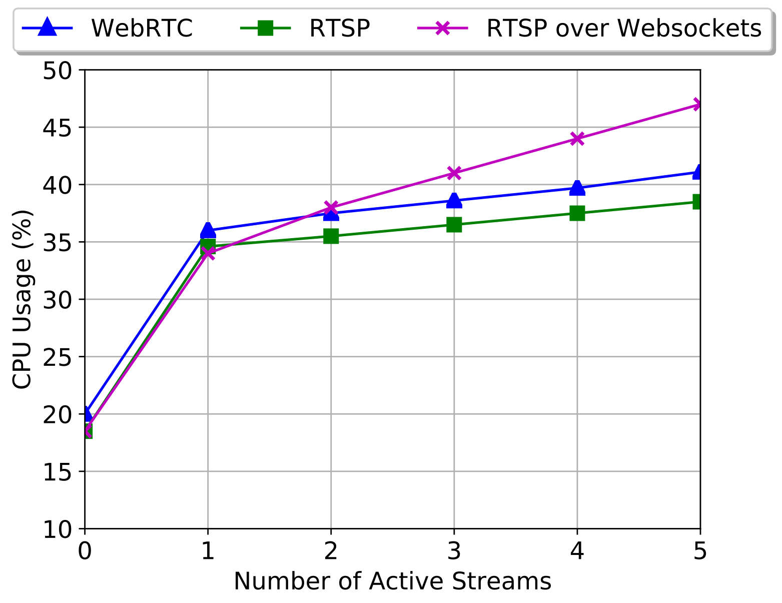


Figure 4.4: CPU usage with low bit rate video, medium profile

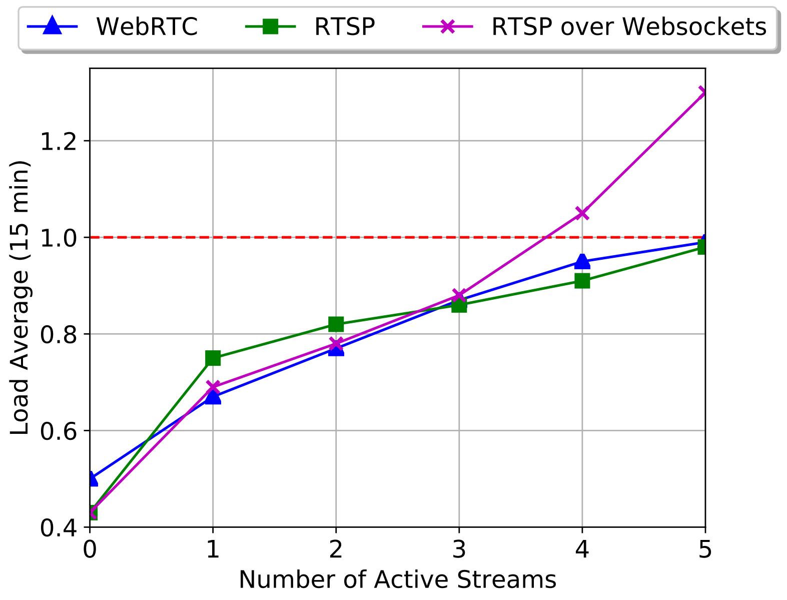


Figure 4.5: Load average usage with low bit rate video, medium profile. The red line represents the recommended maximum load average.

In memory usage (Figure 4.6), we could see that Janus had a higher usage with one client connected but did not use much per additional connected client. The RTSP server worked in the opposite way, using more memory per connected client.

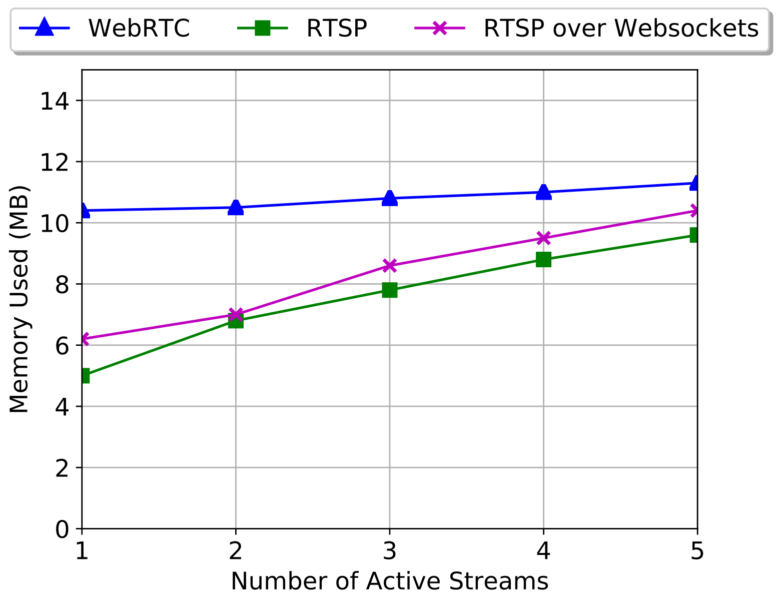


Figure 4.6: Memory used with low bit rate video, medium profile.

Moving Frame Tests

With moving picture, the camera struggles to keep the bit rate down. The video bit rate averages at **3000 kbit/s** using the medium settings, which is ten times higher than with the still frame shot. In the results (Figures 4.7 and 4.8) the fifth active stream was omitted from the results, due to errors in the data from the fifth stream and up. Since the results of more than four streams are of low relevancy, the test was never redone. The memory usage result diagram was omitted since the results were the same as in the still frame tests (Figure 4.6)

This is also done in the coming tests. The memory results did not differ as the bit rate did. Why this happens might be because the frame buffers kept by the servers have static sizes in MB, not in time or number of frames.

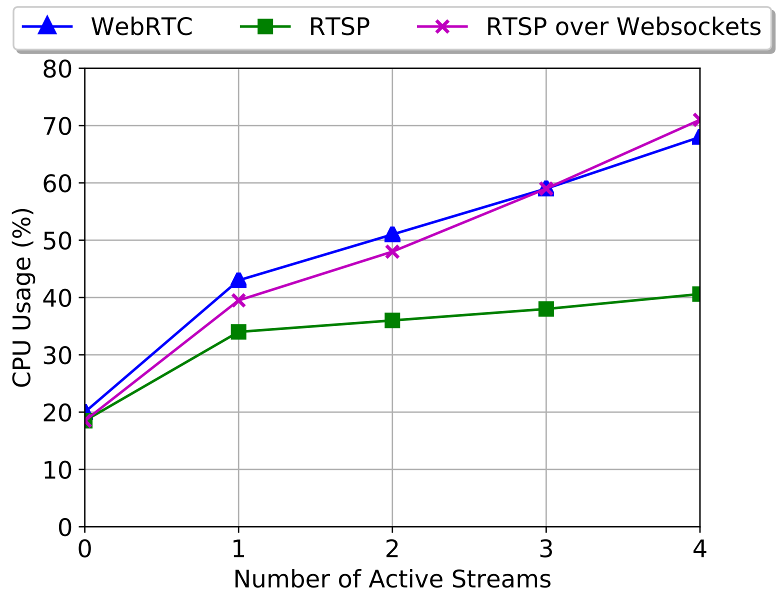


Figure 4.7: CPU usage with high bit rate video, medium profile

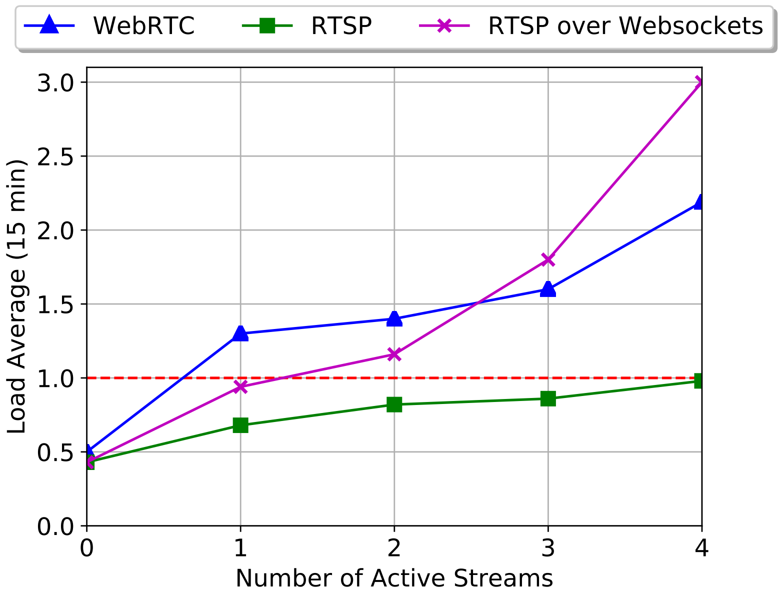


Figure 4.8: Load average with high bit rate video, medium profile. The line at y=1 represents the theoretical max load average.

4.5.2 Stress Tests

In the first stress test (Figure 4.9), the maximum number of clients possible was sought. The test used the medium profile with a still image. 52 clients were the maximum amount possible since when adding the 53rd, all 53 streams became unstable.

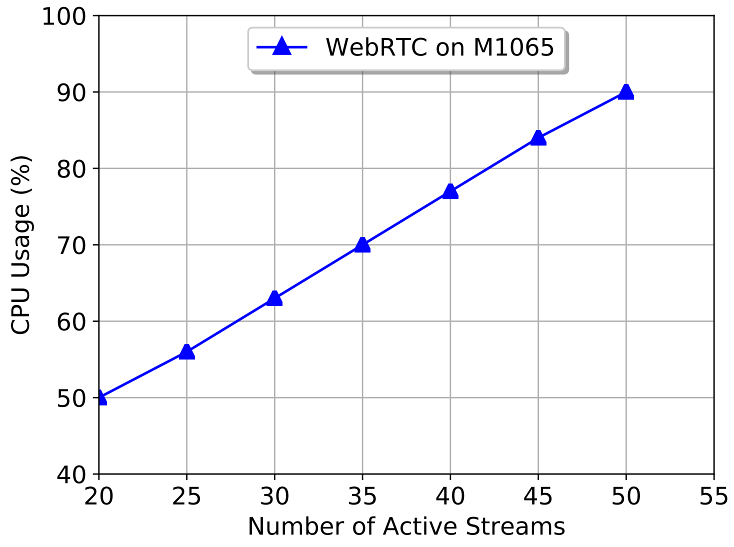


Figure 4.9: CPU usage with still picture, medium profile (≈300 kbit/s). We increased the amount of active streams until they became unstable, which happened as the 53rd stream was added.

For the second stress test (Figures 4.10 and 4.11), the stream profile was changed to the more challenging high profile and the target was moving. The still target was omitted since the focus was on the toughest load possible. The idle data was also kept from the diagrams to keep focus on the most relevant data. The video bit rates in these test were on average **8400 kbits/s**.

Three active WebRTC high profile streams at the same time, seem to be the utmost limit of the camera. The load average is as high as 2.88, which means that many processes are in queue.

At such high load average with the WebRTC server, any small increase in load impacted the stream and made it unstable. We found that opening an ssh connection and for example running `top`, caused the stream to lose packets. Adding a fourth stream resulted in all four streams turning unstable, which is why it was omitted.

Comparing WebRTC to RTSP over WebSocket in the stress tests, with a single stream WebRTC has 10 percentage points more CPU usage. The CPU usage with two and three streams are almost identical. With four streams the RTSP over WebSocket stream lost multiple frames and was therefore omitted from the results. Interestingly, the video player did not show much package loss, instead the frame rate dropped from 15 down to around 10 on all four active streams.

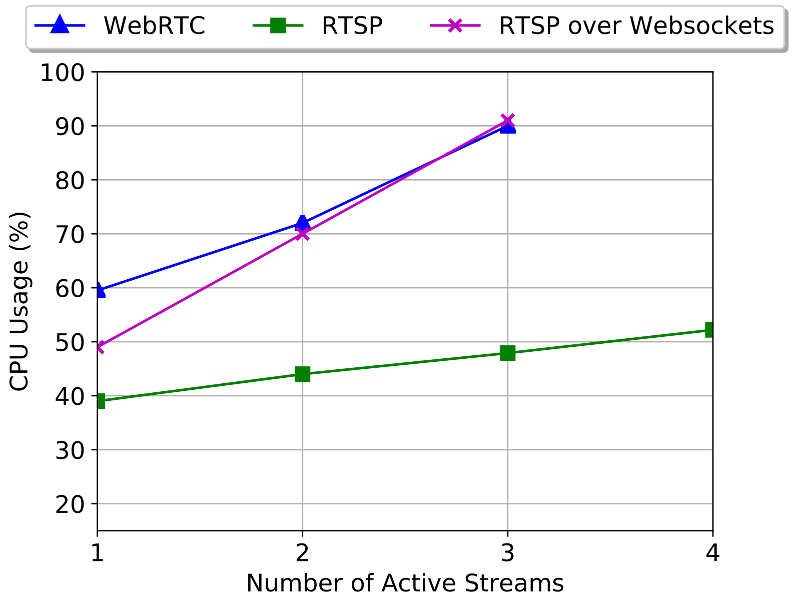


Figure 4.10: CPU usage with moving picture, high profile. Neither the WebRTC or RTSP over WebSocket stream was stable with 4 streams and was therefore omitted.

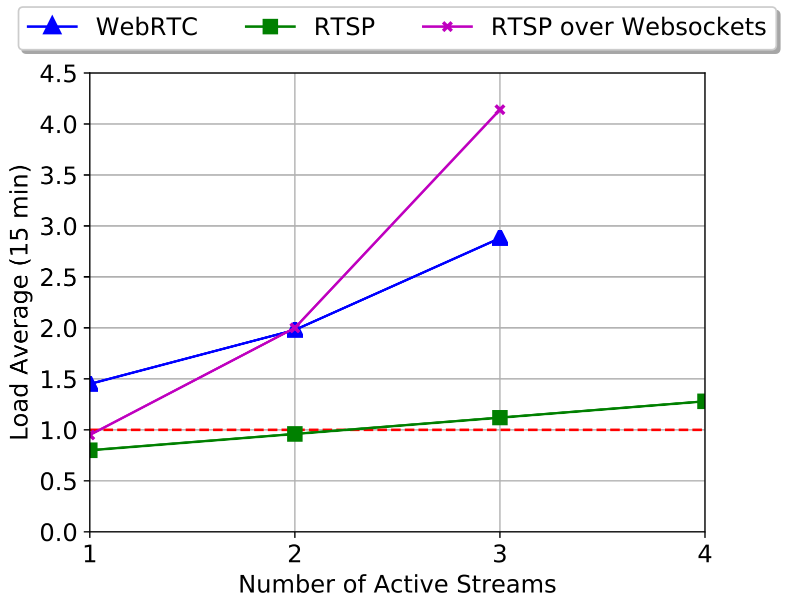


Figure 4.11: Load Average with moving picture, high profile. The line at y=1 marks the theoretical max load average.

Comparing to High End Camera

After finding the breaking point of the M1065, the same tests were performed on the Q3518. An interesting fact that impacts the result, is that the Q3518 manages to compress the video stream much better than the M1065. The bit rate of the Q3518 was on average **4700 kbits/s**, 44% lower than the M1065's **8400 kbits/s**.

The results (Figures 4.12 and 4.13) show that the Q3518 have no problem handling the load that caused the M1065 to lose frames.

Usually with dual core CPU's, such as the one in the Q3518, CPU load can reach 200%, counting each core individually and adding them together. Regarding the Q3518, the CPU usage is calculated differently, 100% meaning both cores being fully used. This was tested running OpenSSL speed test single and multitthreaded respectively. Using one thread, that thread used 50% CPU. With two threads the CPU usage was 100 %.

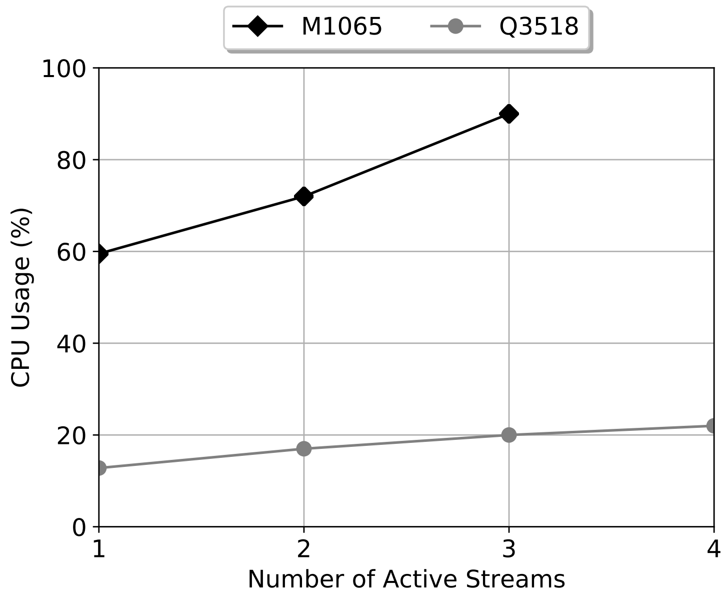


Figure 4.12: Camera comparison showing CPU usage while streaming WebRTC video with moving picture, high profile

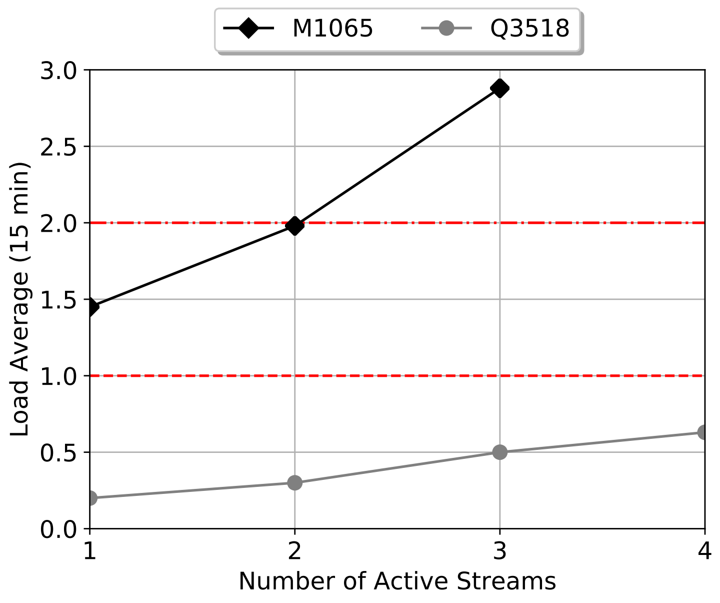


Figure 4.13: Camera comparison showing load average while streaming WebRTC video with moving picture, high profile. The dashed line at *y=1* represents the theoretical max load average for the M1065 camera, the dash-dotted line at *y=2* represents the same but for Q3518.

4.5.3 Network Usage

The data was gathered in the still picture test with the medium profile. This setting was chosen because it is the most common in the use cases. The bit rates outside of the parentheses were calculated from all data travelling between the camera and the client (Table 4.5). In the parentheses, only the data leaving the camera is shown.

The setup procedure that occurs when initiating the connection was not recorded in these results.

The parenthesis values were measured in order to see if there was a noticeable difference in control communication between the client and server. As the results in percentages show (Table 4.6), pure RTSP barely had noticeable amount of control communication. First when data from five active streams was recorded, 1 bit/s can be seen. WebRTC on the other hand, had about 1.4% of all data sent directed the opposite way of the video stream. RTSP over WebSocket had most of all, around 4% of all data was going back to the camera.

Table 4.5: Network usage in bit/s for WebRTC, RTSP and RTSP over WebSocket. The value in parentheses is the bit rate leaving the camera.

No. of streams	RTSP	WebRTC	RTSP over WebSocket
1	351 (351)	361 (356)	337 (324)
3	1077 (1077)	1108 (1093)	998 (957)
5	1793 (1792)	1847 (1821)	1684 (1615)

Table 4.6: Difference in network usage in bit/s for WebRTC, RTSP and RTSP over WebSocket when subtracting the data leaving the camera from the total bit rate. The bit rate shown is the bit rate sent from the client to the camera.

No. of streams	RTSP	WebRTC	RTSP over WebSocket
1	0 (0%)	5 (1.4%)	13 (3.9%)
3	0 (0%)	15 (1.4%)	42 (4.2%)
5	1 ($\approx 0\%$)	26 (1.4%)	69 (4.1%)

4.5.4 Latency

WebRTC through Chrome compares well to RTSP over WebSocket (Table 4.7). The large difference in latency between WebRTC and pure RTSP in VLC looks good initially, but occurs since VLC has a larger cache buffer of 1000ms. RTSP over WebSocket buffers dynamically with about 180-200ms in our tests and WebRTC with about 15-40ms.

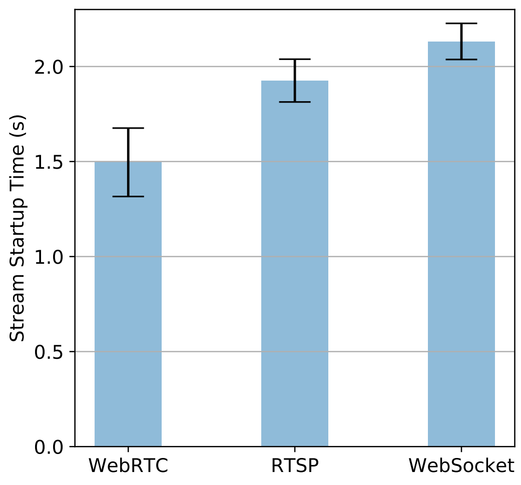
The cache buffer that WebRTC uses is called the jitter buffer and is changed dynamically. There is currently no way for a user to change it, making it hard to test reliably. In *chrome://webrtc-internals*, the current buffer length can be seen in the graph labeled googJitterBufferMs. The modified VLC has a smaller cache buffer with 400 ms instead of 1000 ms, which is the default value. By lowering this value, the latency improves.

Table 4.7: Video latency using different streaming methods

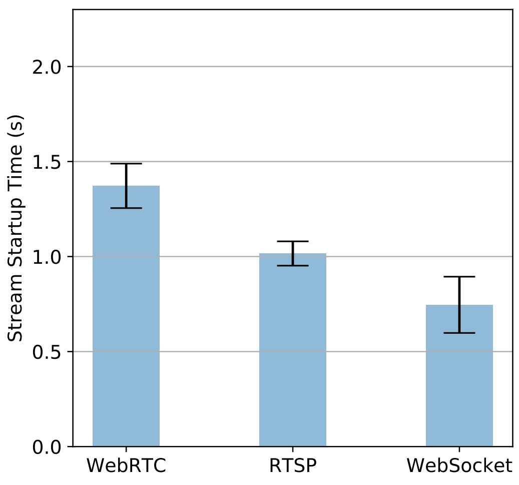
Receiver	Latency (s)	Buffer size (s)
WebRTC (Chrome)	0.27	0.040
RTSP over WebSocket (Chrome)	0.34	0.2
RTSP (VLC)	1.28	1
RTSP (modified VLC)	0.73	0.4

4.5.5 Stream Startup Times

As stated in Section 4.3.5, we measured the stream startup times by calculating the average of five startup times measurements. The difference in startup times for WebRTC (Figure 4.14) when adding more users is negligible. That the difference is negligible makes sense, since the CPU usage/load average (Figures 4.4 and 4.5) is so low with the tested amounts of active streams.



(a) First stream



(b) Sixth stream

Figure 4.14: Stream startup times when starting the first and the sixth stream

Chapter 5
Discussion

In this chapter the results will be discussed in regards to the use cases (Section 1.4).

5.1 Latency

When removing the buffering from the latency results, all receivers result in having almost identical latency (Figure 4.7), RTSP over WebSocket having a small advantage. Given the precision that we had in these tests, the results do not say more than that we could not see a major difference in latency. One has to consider that both WebRTC and RTSP over WebSocket have dynamic buffering. The goal with the tests, was partly to see if WebRTC added much extra latency compared to RTSP over WebSocket, which we can not see that it does.

What is interesting with the tests is the different buffering strategies. VLC has a long buffer (one second), ensuring a stable stream without any artifacts. WebRTC is keeping the buffer dynamic and as small as possible (15-40 ms). Such a small buffer means that latency is prioritised over a stutter free video, which makes sense in video conferencing or similar environment. RTSP over WebSocket buffers dynamically with roughly 180-200ms in our tests, keeping a sort of middle ground.

These results show that WebRTC should be able to handle latency well in all four use cases (Section 1.4).

5.2 Stream Startup Times

The stream startup time for WebRTC was much faster (around 0.5 seconds) when starting the first stream, compared to RTSP over WebSocket and pure RTSP. With the idle performance results in mind this makes sense. The RTSP servers go into deeper sleep states when not in use, while the Janus server does not. The reason for WebRTC being so fast when starting the first stream, is that there is already a RTSP stream connected in the background, since the default WebRTC case always has an active mountpoint. RTSP over WebSocket and pure RTSP both have to spend more time starting up in general, leading to increased stream startup times.

When there already are five active streams, RTSP over WebSocket got much faster and was noticeably faster than the other solutions.

Even if the startup times are slightly slower with more active streams with WebRTC, it is still not that noticeably slower than the others. One can still conclude that based on the results, the startup times for WebRTC should be fine for all four use cases (Section 1.4).

5.3 Network Usage

RTSP over WebSocket has noticeably more communication traffic compared to the other two. It has around 4% of all traffic going to the camera, compared WebRTC's 1.4%.

RTSP over WebSocket is always using TCP for its transportation while WebRTC most often uses UDP. This is the main reason for the large amount of communication traffic. TCP sends out more messages that need to be acknowledged compared to UDP. UDP is designed for real time communication and not as focused on reliability as TCP is, since UDP has very basic error checking mechanics compared to TCP. In a livestreaming scenario, where only the latest frame is of interest, there is no need for retransmission of lost frames.

WebRTC uses UDP as a first resort but can fall back to TCP if, for example, restricting firewalls force it to. Since our solution was used only on a local network, it never had to resort to TCP and could use UDP for all tests. This did not impact WebSocket, since WebSocket always uses TCP.

5.4 CPU, Load Average and Memory

We will in this section discuss using the names of the servers rather than the names of the ways of streaming, in order to improve readability. The names used are:

- **monolith** The RTSP server
- **httpd** The WebSocket server
- **janus_server** The WebRTC server

Idle

Having the system under unnecessary high load while being idle should be avoided, since it affects both power consumption and life length. It was therefore important to evaluate the idle performance both with and without the *janus_server* running, also with and without an active mountpoint.

If we first ignore the idle load with a mountpoint active, the total CPU usage and load average for WebRTC is about 10% higher than the two other (Table 4.4). This is not too surprising since *janus_server* has two web servers running, one for the Janus interactions and one for the admin interface. The admin interface is most likely worth disabling to see if that helps. At the same time, the RTSP server is also running in the background.

The issue with the high idle load with an active mountpoint can not be ignored, even if we found workarounds (Section 3.2.2). The reason why so much load is put on the camera is because the *monolith* is actively serving one stream. This can be supported by comparing the idle CPU usage for *janus_server* with an active mountpoint (Table 4.4) with the *monolith* serving one client (Figure 4.4), they are almost identical.

Idle performance is mostly relevant for the first three use cases. The results are promising and shows that WebRTC can perform well in an idle situation, making it competitive in this criteria.

Normal Scenarios

Under normal circumstances the *janus_server* performs well. One viewer is never a problem, no matter the stream profile. Using the medium profile, there are no issues while having more concurrent viewers than what the three first use cases require (Section 1.4).

The *janus_server* still requires more resources than the built in *monolith*. This is not surprising at all, since in this implementation, the *monolith* always has to feed the *janus_server* with video and the RTSP stream is not encrypted. The same goes for the *httpd*, which also is fed with a RTSP stream from the *monolith*.

The CPU usage of *httpd*, with the still and moving pictures (Figures 4.4 and 4.7) are in both cases linear. In the low bit rate case *httpd* uses more resources per added client, but still has almost no overhead with one client. We found this to be because how *httpd* shares its streams. Instead of fetching one RTSP stream and sharing it to many clients the server fetches one RTSP stream per connected client. This was found by looking at not only the total CPU, but CPU usage per process. For every client that *httpd* adds, the *monolith* uses more CPU. But for *janus_server*, the *monolith* always has the same CPU usage, no matter the amount of clients.

Janus_server has noticeable lower CPU usage than *httpd* with low bit rate, but they perform almost similar when the bit rate increases (Figure 4.7). Both servers apply the same encryption, AES 128 bit, and both gateway the RTSP stream in some way. The difference is in how the stream is relayed through a gateway. The RTSP over WebSocket sends the RTSP stream over a WebSocket connection (TCP), whereas the WebRTC server takes the RTP stream from a RTSP stream and transmits it using UDP. The RTP extraction together with some extra control messages that ensures the quality of the stream is probably why the results even out.

It is important to note that the CPU usage results are so even between the *janus_server* and *httpd*, that it is hard to say that one performs better than the other. By adding some margin of error to the results, they are almost indistinguishable. This should not be considered a limitation to the method, but rather speak about how comparably the two servers perform.

Looking at memory usage, the usage in all tests were generally low. *janus_server* used about 10 MB when a mountpoint was active and not more than additional 0.25 MB per connected client. The two other servers are not as easy to analyse at zero connections since they are always active on the cameras. The *monolith*, which is responsible for the RTSP stream, must be running for either of *httpd* and *janus_server* to work. Therefore the results at zero clients were not included. We will not put more effort into analysing the memory usage since the results were of so small importance.

The load average results (Figures 4.5 & 4.8) almost follows the CPU usage results perfectly.

httpd always had lower or equal load average with one and two active streams but as the amount of active streams increased to three to five the load average took off. A reasonable explanation why the load average increases at a higher rate for *httpd* than for *janus_server* is again because how *httpd* adds a new separate RTSP connection from the *monolith* for every client. This adds more processes that will be put in queue and with the extra load from the *monolith* the CPU usage also rises.

All the results from the normal scenario tests using the M1065 show that WebRTC performs well in the use cases 1, 2 and 3 (Section 1.4).

Stress Tests

janus_server performed very similar to *httpd* in the high profile stress test (Figures 4.10 & 4.11) and most of the discussion about the performance that was covered above applies here. For one active stream, the gap grows and seems to be larger with more active streams. However, the increase is linear to the bit rate when comparing to the lower bit rate streams.

After running the stress tests, we can conclude that the bottleneck is in the cameras hardware and not in Janus or the WebRTC stack. We could push the M1065 to almost 100% CPU usage and a load average well above comfortable levels without the stream breaking. When switching over to the more powerful camera, the task that overstrained the M1065, was effortless for the Q3518 camera.

janus_server performs almost as well as *httpd* and that is very good for software designed for server grade hardware. This is even without any optimisation for the embedded devices used in our tests.

During the stress tests we found that the WebRTC streams had higher chance of breaking and not being able to recover, if many packets were lost in a short amount of time, compared to the two RTSP based streams. The reason we can see for this is the option used when retrieving the stream from the cameras RTSP server, *videopsenabled=1*.

When learning about this option it was noted that it makes the stream more vulnerable to packet loss. The SPS and PPS should be sent in a separate TCP stream, also called “out of band”, to better accommodate packet loss, something that should be investigated further. However, setting up a separate TCP connection besides the UDP connection that WebRTC uses seems like a step in wrong direction.

The stress tests show that in its current state, the surveillance use cases (1, 2 & 3 in Section 1.4) should have no issues using WebRTC as a streaming method. Since 50 concurrent streams could be active at the same time with a still image and medium profile (Figure 4.9), there is potential that WebRTC could also do well in the educational/webinar use case.

As a last comment it is important to note that the stress test with the high quality stream profile was done in a way that does not reflect any currently real use case. It can however be seen as a test for future implementations where higher quality video is sought.

5.5 Video Codecs

Having established that the load put on a camera is tied to the bit rate of the video, we can discuss what more efficient video codecs can do. Newer codecs offer much better compression and with more chips offering hardware accelerated encoding with these codecs, the market will soon see a switch from H.264 and VP8. As mentioned in Section 2.4 , some Axis cameras released during 2019 will offer support for H.265 together with Axis Zipstream technology [29]. It is however just not that simple. The chips that support hardware accelerated H.265 encoding are newer, more powerful chips that have no problem streaming the H.264 streams used in this thesis. The newer codecs are more relevant in a network bandwidth sense.

Chapter 6
Conclusions

Our study has found that WebRTC, streamed using the server Janus, performs just as well as competing technologies while providing an array of added benefits, such as simpler certificate handling and being peer-to-peer. Streaming encrypted video in 720p poses no problem for the cameras in our tests, it is also future proof, as high bit rate 1080p streams works without any issues. Latency and stream startup times are also on par with the tested competing alternatives.

The WebRTC server Janus was deployable on the IP-cameras used in our tests without modifying the firmware on the cameras.

Janus also proved to be a robust and efficient server. Both Janus and WebRTC are still in very active development. For example, Janus sees multiple commits per day. This is a good sign for future users of both softwares.

The first two of the four studied use cases (surveillance) work very well. There is, with minor restrictions to video quality, no problem streaming to 1-5 viewers at the same time.

Even if no solution to handle bidirectional audio communication was developed, our tests indicate that the third use case, Door Station, has potential to work well.

The fourth use case, education/webinar, needs a little bit more performance from the camera in order to work well. In the tests 50 clients were the limit, and that at a very low bit rate.

We can conclude that we have answered all research questions with results that opens paths for future endeavours with WebRTC.

6.1 Future Work

This thesis only scratches the surface of what WebRTC can achieve together with surveillance cameras. WebRTC streaming from embedded devices is a relatively undiscovered area and after completing this thesis, we have multiple future work proposals.

GStreamer: Instead of using the cameras built in RTSP server and relaying the stream to WebRTC using a WebRTC gateway, a more effective solution could be to change the GStreamer pipeline on the camera. GStreamer recently added a WebRTC sink in version 1.14. By having GStreamer output WebRTC directly instead of RTSP, the conversion process from RTSP to WebRTC would be removed. This can potentially lead to not only camera performance enhancement, but also improvements in latency and other stream related performance metrics. The main reason for us not investigating this, is the deployability issue. Installing a WebRTC server on the cameras as an ACAP is much easier than updating GStreamer and using the sink, especially on older cameras.

More Cameras: At the start of our thesis work we did not know how the solution would perform, or if it was even possible to implement. This lead us to focus on producing and implementing the solution for a specific camera model and then performing tests with that. Ideally one would want to implement the solution for a much wider range of products, both high-end and low-end cameras. By doing this, one would get a much clearer and better picture on how well the solution works on different cameras.

Other Servers: We decided to use Janus but there are many other WebRTC servers that are worth trying. Even if Janus works well, it is hard to know if there could be some other WebRTC server that might be able to outperform Janus. Other WebRTC servers worth looking at are Pion WebRTC and Kurento.

Stability: Having done more tests in general would also help with identifying issues and making the solution more robust. More tests done with higher sample rate could potentially help identifying the issues to exactly why streams crash, as they did in the stress tests (Section 4.5.2).

Bibliography

[1] M. Priks. The effects of surveillance cameras on crime: Evidence from the stockholm subway. *The Economic Journal*, 125(588):F289–F305, 2015.

[2] Axis Communications. Axis annual report 2017. https://www.axis.com/files/annual_reports/Axis_AB_ars_och_hallbarhetsredovisning_2017.pdf, 2018.

[3] Hikvision. Hikvision annual report 2017. https://oversea-download.hikvision.com//uploadfile/Investment%20Relationship/Hikvision%202017%20Annual%20Report.pdf, 4 2018.

[4] B. F. Jong. Webrtc based broadcasting for physical surveillance. Master's thesis, Swinburne University of Technology, 2018.

[5] S. Thörnqvist M. Lindfeldt. Real-time video streaming with html5. Master's thesis, Lunds Tekniska Högskola, Lund, Sweden, 2014. http://sam.cs.lth.se/ExjobGetFile?id=687.

[6] A. Johnston, J. Yoakum, and K. Singh. Taking on webrtc in an enterprise. *IEEE Communications Magazine*, 51(4):48–54, April 2013.

[7] 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 '14, pages 7:1–7:8, New York, NY, USA, 2014. ACM.

[8] A. Amirante, T. Castaldi, L. Miniero, and S. P. Romano. Performance analysis of the janus webrtc gateway. In *Proceedings of the 1st Workshop on All-Web Real-Time Systems*, AWeS '15, pages 4:1–4:7, New York, NY, USA, 2015. ACM.

[9] 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 '16. IEEE, 2016.

[10] A. Amirante, T. Castaldi, L. Miniero, S. P. Romano, and A. Toppi. Measuring janus temperature in ice-land. pages 1–7, 10 2018.

[11] Axis Communications. Product guide: Network video solutions. http://ftp.axis.com/pub/users/goran/dvd_design_tool_40503/collateral/pg_video_40433_en_1010_lo.pdf, 6 2019. Accessed on 2019-06-13.

[12] C. Vogt, M. J. Werner, and T. C. Schmidt. Leveraging webrtc for p2p content distribution in web browsers. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–2, Oct 2013.

[13] S. Dutton. Webrtc tutorial. https://www.html5rocks.com/en/tutorials/webrtc/basics/. Accessed on 2019-02-22.

[14] W3C. W3c announcing webrtc as cr. https://www.w3.org/blog/news/archives/6619. Accessed on 2019-02-22.

[15] W3C. W3c recommendation track process. https://www.w3.org/2004/02/Process-20040205/tr.html. Accessed on 2019-02-22.

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

[17] C. Perkins M. Handley, V. Jacobson. Sdp: Session description protocol. Technical report, RFC Editor, July 2006. https://tools.ietf.org/html/rfc4566.

[18] E. Van der Sar. Vpn webrtc leak. https://torrentfreak.com/huge-security-flaw-leaks-vpn-users-real-ip-addresses-150130/. Accessed on 2019-02-27.

[19] ONVIF. Onvif streaming specification version 17.06. Technical report, 2017. https://www.onvif.org/specs/stream/ONVIF-Streaming-Spec-v1706.pdf.

[20] W. May R. Pantod. Http live streaming. Technical report, RFC Editor, August 2017. https://tools.ietf.org/html/rfc8216.

[21] R. Frederick V. Jacobson H. Schulzrinne, S. Casner. Rtp: A transport protocol for real-time applications. Technical report, RFC Editor, July 2003.

[22] R. Lanphier M. Westerlund M. Stiemerling Ed. H. Schulzrinne, A. Rao. Real-time streaming protocol version 2.0. Technical report, RFC Editor, December 2016.

[23] L. Miniero. Official website of janus. https://janus.conf.meetecho.com/. Accessed on 2019-02-22.

[24] L. Miniero. Fosdem 2016: "janus: a general purpose webrtc gateway". https://archive.fosdem.org/2016/schedule/event/janus/attachments/slides/967/export/events/attachments/janus/slides/967/fosdem2016_janus_presentation.pdf, 1 2016.

[25] T. Wiegand, G. J. Sullivan, G. Bjøntegaard, and A. Luthra. Overview of the h.264/avc video coding standard. Technical report, 2003. http://ip.hhi.de/imagecom_G1/assets/pdfs/csvt_overview_0305.pdf.

[26] L. Quillio J. Salonen P. Wilkins Y. Xu J. Bankoski, J. Koleszar. Vp8 data format and decoding guide. Technical report, RFC Editor, November 2011.

[27] T. Levent-Levi. 4 reasons to choose h.264 for your webrtc service (or why h.264 just won over vp8). 05 2016. https://bloggeek.me/h264-webrtc/. Accessed on 2019-05-13.

[28] M. Sharabayko. Next generation video codecs: Hevc, vp9 and daala. 11 2013.

[29] Axis Communications. Övervakningskameror med utökade säkerhetsfunktioner och stabil video under alla förhållanden. https://www.axis.com/sv-se/newsroom/news/overvakningskameror-med-utokade-sakerhetsfunktioner-och-stabil-video-under-alla -forhallanden, 4 2019. Accessed on 2019-05-03.

[30] Axis. Axis camera application platform. https://www.axis.com/support/developer-support/axis-camera-application-platform. Accessed on 2019-03-07.

[31] Gstreamer technical documentation. https://gstreamer.freedesktop.org/documentation/application-development/introduction/gstreamer.html. Accessed on 2019-04-18.

[32] Mozilla implements openh264. https://blog.mozilla.org/blog/2013/10/30/video-interoperability-on-the-web-gets-a-boost-from-ciscos-h-264-codec/. Accessed on 2019-03-29.

[33] D. Ferrari and S. Zhou. An empirical investigation of load indices for load balancing. Technical report, Berkeley, CA, USA, 1987. https://www2.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-353.pdf.

[34] Ambarella s2l product sheet. https://www.ambarella.com/uploads/docs/S2L-Product-Brief-Final.pdf. Accessed on 2019-03-29.

[35] Browser market share. http://gs.statcounter.com/browser-market-share#monthly-201707-201707-map. Accessed on 2019-03-29.

[36] Wireshark website. https://www.wireshark.org/. Accessed on 2019-06-12.

Appendices

Appendix A
Cross compilation

A.1 Janus Dependencies

Libs we did not have to cross compile:

libsrtp v2.X

Jansson v2.9

Libs that had to be cross compiled:

libnice v0.1.13 suggested, we used v0.1.15

libgnutls v3.6 (libnice dependency)

nettle v3.4.1 (libgnutls dependency)

gmp v6.1 (nettle dependency)

libtasn1 v4.13 (libgnutls dependency)

glib v2.40 (libnice dependency) Tests always failed while configuring and the option `-disable-tests` did not help. Instead a fix where environmental variables were set beforehand was used. A file named `armv7hf.config` was created and when executing `./configure`, `CONFIG_SITE=armv7hf.config` was prepended. The file consisted of the four lines which acts as the correct answers to the tests:

```
ac_cv_type_long_long=yes
glib_cv_stack_grows=no
glib_cv_uscore=no
ac_cv_func_posix_getpwuid_r=yes
```

libffi v3.1 (glib dependency)

openssl Version found on camera did not support an elliptic curve function necessary (was a special internally modified build). We used version 1.1.1b.

libmicrohttp v0.9.63

libconfig v1.7.2

libcurl v7.64.0

Appendix B
Configuration Files

B.1 janus.plugin.streaming.jcfg

```
axisCameraMedium: {
  type = "rtsp"
  id = 1
  description = "M1065-LW-Medium"
  audio = false
  video = true
  url = "rtsp://127.0.0.1/axis-media/media.amp?
          streamprofile=medium\&videopsenabled=1"
  rtsp_user = "user"
  rtsp_pwd = "password"
}
```

B.2 janus.jcfg

```
general: {
  configs_folder = "conf"    # Configuration files folder
  plugins_folder = "plugins"       # Plugins folder
  transports_folder = "transports"     # Transports folder
}
certificates: {
  cert_pem = "certs/mycert.pem"
  cert_key = "certs/mycert.key"
}
```

B.3 package.conf

```
PACKAGENAME="Janus"
MENUNAME="Janus"
APPTYPE="armv7hf"
APPNAME="janus"
APPID=""
LICENSEPAGE="none"
VENDOR="Axis Communications"
REQEMBDEVVERSION="1.10"
APPMAJORVERSION="0"
APPMINORVERSION="1"
APPMICROVERSION="0"
APPGRP="root"
APPUSR="root"
APPOPTS=""
OTHERFILES=""
SETTINGSPAGEFILE=""
SETTINGSPAGETEXT=""
VENDORHOMEPAGELINK="http://fredagskakan.se"
POSTINSTALLSCRIPT=""
STARTMODE="respawn"
HTTPCGIPATHS=""
CERTSETNAME=""
CERTSETACTOR=""
CERTSETPROTOCOL=""
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2019-06-11

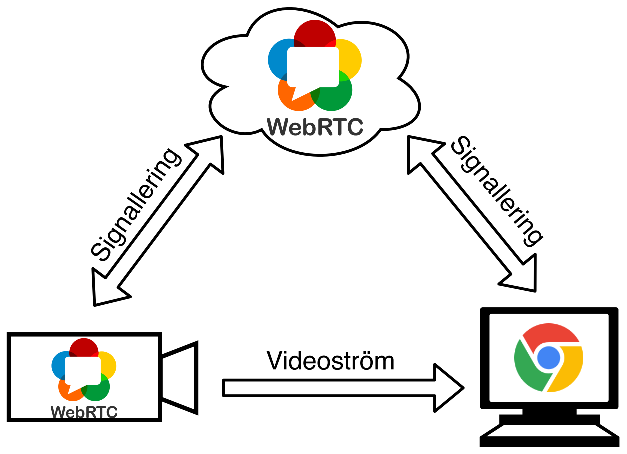
EXAMENSARBETE WebRTC for peer-to-peer streaming from an IP-camera
STUDENTER Johan Gustavsson, Hampus Christensen
HANDLEDARE Mathias Haage (LTH), Jesper Forsberg (Axis), Svante Richter (Axis)
EXAMINATOR Flavius Gruian (LTH)

Peer till peer videoströmning med WebRTC från övervakningskameror

POPULÄRVETENSKAPLIG SAMMANFATTNING
Johan Gustavsson, Hampus Christensen

Video från övervakningskameror strömmas idag ofta genom en mellanliggande server innan den når åskådaren. Att istället strömma video direkt, peer till peer, från kameran till åskådaren har många fördelar. Detta arbete har fokuserat på att utvärdera prestanda vid videoströmning med WebRTC.

Axis, en av världens ledande tillverkare av IP-baserade övervakningskameror, strömmar idag video genom att tunnla RTSP genom molnservrar med hjälp av WebSocket eller HTTP. Att istället strömma video peer till peer med WebRTC har många fördelar, så som enklare uppsättning och minskade serverkostnader.



WebRTC är ett mediaströmnings-API som möjliggör säker direktkommunikation mellan webbläsare över internet, utan extra insticksprogram. API:t är utvecklat av Google och har från början varit öppet, vilket gör det enklare för fler att använda sig av tekniken. För autentisering tar WebRTC hjälp av en så kallad signalleringserver som även hjälper parterna att finna varandra. Vanliga användningsområden för WebRTC är röst- och videosamtal, där några stora tjänster som implementerar detta är Facebook Messenger, Slack och Discord.

Med WebRTC kan webbläsare agera både klient och server. När det inte finns en webbläsare, som det inte gör på övervakningskameror, så behövs det en WebRTC-server. Vi installerade Janus, en WebRTC server med öppen källkod som bland andra Slack använder, på två olika kameror som tester sedan utfördes på. Testerna utformades för att utvärdera hur en WebRTC server presterar på övervakningskameror i flera olika användningsfall, så som övervakning men även för webbkonferenser och porttelefoner.

Resultaten visar att WebRTC presterar likbördigt, jämfört med konkurrerande tekniken RTSP över WebSocket. Eftersom WebRTC medför tidigare nämnda fördelar, kan utifrån detta slutsatsen dras att WebRTC är ett bra alternativ för videoströmning inom övervakning, porttelefoner och även webbkonferenser.