MASTER'S THESIS 2022

Integration and Evaluation of WebRTC in an Existing .NET Environment

Simon Tenggren, Martin Gottlander

ISSN 1650-2884
LU-CS-EX: 2022-05

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-05

Integration and Evaluation of WebRTC in an Existing .NET Environment

Integration och utvärdering av WebRTC i en existerande .NET miljö

Simon Tenggren, Martin Gottlander

Integration and Evaluation of WebRTC in an Existing .NET Environment

Simon Tenggren
si6187te-s@student.lu.se

Martin Gottlander
bte15mgo@student.lu.se

March 1, 2022

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Jörn Janneck, jorn.janneck@cs.lth.se
Tore Paulsson, tore.paulsson@axis.com

Examiner: Per Andersson, per.andersson@cs.lth.se

Abstract

WebRTC is a recently standardized technology for establishing peer-to-peer connections for transmitting real-time video, audio, and data. It can dynamically find the best way to connect two peers to each other depending on the networks the peers are located on. The authors alongside Axis Communications AB wish to explore whether WebRTC is a viable streaming solution for their current Video Management Solution (VMS) software, as well as if it is possible to integrate with existing features in their .NET environment. Two WebRTC implementations, GStreamer and SIPSorcery, were evaluated, both against each other and against an existing streaming solutions used at Axis Communications. They were evaluated on metrics such as end-to-end latency, the time it takes to establish a connection and start displaying a stream, as well as CPU and memory usage. The performance of the WebRTC implementations were also evaluated for situations when the connection could be established as a pure peer-to-peer connection and for situations when a server has to relay data to be able to traverse restrictive networks.

The WebRTC implementations generally performed better in the case of the time-to-stream metric, however significantly were effected when using a relaying server. The latency test implied that WebRTC was equivalent to the existing streaming solution, while during the CPU and memory usage experiments, the WebRTC implementations performed slightly worse. The integration with the current .NET environment used at Axis could be done, and several of the features that are present in the VMS software, while other require more work, or change of the currently used protocols.

The two WebRTC implementations perform similarly but differ in the available features and supported protocols. GStreamer is more mature and supported, and gives access to finer granularity of network and media processing, however it is more difficult to integrate into the existing environment compared to SIPSorcery, due to the additional software required to be installed and interacted with. Comparatively SIPSorcery which is semantically similar to the WebRTC API in most modern web browsers, which eases the transition from web development to .NET applications.

Keywords: WebRTC, Peer-to-peer, Evaluation, Streaming, WPF, .NET

Acknowledgements

The authors would like to thank the following Axis employees: Felix Kaaman for their support on understanding and implementing the Axis decoding and rendering components,

Jonas Cremon for their support in debugging the WebRTC implementation on the network cameras,

Anders Magnusson for answering our questions concerning the current peer-to-peer streaming solution.

The authors would also like to thank Aaron Clauson, developer and maintainer of the SIPSorcery repository, for his quick communication and feedback on the issues we posted during the writing of this thesis.

Contents

Chapter 1
Introduction

WebRTC (Web Real-Time Communication) is a standard for establishing peer-to-peer connections and streaming real-time video, audio, and data. The standard has been in development for several years and since January of 2021 it was announced by the Internet Engineering Task Force (IETF) and World Wide Web Consortium (W3C) as an official standard. [28] The WebRTC standard is built upon a set of technologies that are available through ECMAScript API's in many modern web browsers, which allow users to establish reliable and flexible peer-to-peer communication channels between a wide variety of different platforms such as mobile, Internet-of-Things (IoT) devices, and browsers. The WebRTC API[27] as defined by W3C has also been integrated in libraries targeting some native desktop applications, see Appendix table B.1 for full list. The shared standard and API allows developers and companies to reuse much of the code and infrastructure to support new devices and platforms. Several popular day-to-day applications such as Discord, Facebook Messenger and Google Meets, use WebRTC in both their desktop, web, and mobile clients to facilitate their video and audio services. [26]

Previously establishing fast and reliable peer-to-peer connections have been a big challenge. Peers can be on different networks, behind one or more NATs and/or firewalls and have no idea on how to reach each other through the internet. As such the traditional method has been to use a server as an intermediary which both peers are aware of which they can connect to and exchange data on. If a peer-to-peer connection would be an absolute necessity a non-standardized homebrew would have to be used that may or may not find the optimal way of establishing the peer-to-peer connection. One of the big advantages of the WebRTC is that it eliminates the need for complex homebrew solutions and the need for a server to act as an intermediary, WebRTC solves this problem in a open and standardized way, and can dynamically find the best way to connect two or more peers.

Axis Communications AB ("Axis") is leading the market within IoT video devices and wish to explore the possibility of using WebRTC in their Video Management Systems (VMS); AXIS Companion (ACC) and AXIS Camera Station (ACS). Furthermore, they wish to compare and evaluate WebRTC implementations against their existing solutions on aspects such

as; establishing peer-to-peer connections, latency, and the time from requesting video until a video stream can be shown. All of which are of great importance to their end customers.

ACC and ACS differ in scale and use case, however they have several things in common; these applications are written in the C# UI framework Windows Presentation Foundation (WPF) released in 2006 as part of the .NET Framework 3.0, and use common proprietary solution for decoding and rendering video, developed in-house at Axis, as well as supporting several video manipulation features such as Digital Pan-Tilt-Zoom (PTZ) and client side dewarping which allow the users to explore and manipulate the image which is rendered.

WPF is widely used by many developers and companies the world over, but as Microsoft develops their other UI-frameworks such as Universal Windows Platform (UWP) and WinUI, WPF has not been given the same treatment as it once had. A prime example of which is native support for WebRTC. WPF applications have to rely on solutions developed for cross-compatibility such as XAML-islands which supplies WPF applications with some of the support that is available in UWP. [21] As the common rendering solution currently available at Axis is built for rendering in WPF applications without the use of these cross-compatibility solutions it is in the interest of Axis to have a more modular and WPF oriented solution that can easily integrate with the features that Axis VMS solutions already offer, to retain many years of hard work in developing features which are unique to many of Axis' products.

Problem statement

The research questions posed are the following:

1. When integrating WebRTC as as a streaming solution that interacts with IoT devices made by Axis, how does it compare with the current streaming solutions at Axis in terms of various quality and reliability metrics?

2. Can the WebRTC solution(s) support the same features that exist in the current streaming solution at Axis? That is the following:

   (a) Client Side Dewarping and Digital PTZ

   (b) Audio Playback and Transmission

   (c) Scrubbing

1.1 Related Work

To answer the first research question the solutions need to be tested and benchmarked to be able to evaluate metrics regarding quality and reliability. In order for our work to be comparable to other works this project started with literature studies to review the current state of testing and benchmarking for WebRTC.

García et al. has published several papers on WebRTC. *Kurento: the Swiss army knife of WebRTC media servers* [13] presents Kurento which is a WebRTC media server in Java while the WebRTC stack and other media processes are implemented using the media pipeline framework GStreamer. Together with Kurento comes the Kurento Testing Framework [15][12] which offer automated functional, performance and Quality of Experience tests for WebRTC based web applications. In this case, functional tests include testing that media is

received through detecting color and recording media communication events. Performance tests considers the latency between sent frame to received frame and is measured automatically through synchronization of clocks and Optical Character Recognition, to be able to read a timestamp that the sender transmits. Kurento Testing Framework also allows for testing at scale through the creation of fake browsers to generate various amount of traffic, and finally it supports Docker containers to be able to configure network conditions, such as congestion or specific network configurations that affect WebRTC traffic.

Gouaillard et al.[16] also aims at creating a generic testing platform for WebRTC through the KITE project. This project test WebRTC at many different levels such as compliance testing for the W3C specification of WebRTC, as well as interoperability between browsers and even native desktop applications. In the future work section of their paper, Gouaillard et al. state that a programmatically configurable network with regards to firewall rules and bandwidth is a must have but it is not clear if this is has yet been implemented in the KITE engine testing suite.

Amirante et al. introduces *Jattack* [10] which is a stress test tool for WebRTC enabled server side components. This tool is based on Janus media server and the paper evaluates Jattack performance in terms of CPU and memory usage when multiple presenters and viewers are using the same Janus media server. This paper also measures number of negative acknowledgements (NACK) to gain insight on whether and when the media is degrading from adding more and more concurrent peer connections to the server.

Taheri et al. presents *WebRTCBench* [11] which is a tool for performance assessment of browser implementations of the WebRTC specification on different architectures. This work is dedicated to finding bottlenecks and thus gives a good indication on where improvements could be made. Three distinct parts are evaluated which are; session establishment time, latency on the channel that transmits arbitrary data (e.g not video or sound) and media engine performance. WebRTCBench has not been updated since 2015 but gives an approach to measure session establishment time which is an important quality for Axis.

1.2 Limitations

WebRTC requires several components to function, namely two peers, a signaling server, and optimally a number of Interactive Connectivity Establishment (ICE) servers. The work in this thesis is limited to the implementation of a receiving peer since the sending peer is currently being implemented at Axis, but has not been thoroughly evaluated for video streaming. In this context, the sending peer is an IP-camera sending live video to the receiving peer, and thus the communication between the peers after connection has been made is mainly focused on unidirectional video communication.

The signaling server has already been implemented with a defined protocol which allows for authentication using Axis' authentication services. As such the performance of the signaling server and the sending peer will not be modified and evaluated, however the authors hope that the work done can be used as support for decisions regarding eventual modifications to both the network cameras WebRTC implementation and the signaling service. The current streaming solution will not be discussed in detail due to disclosure concerns, as such the solution will only be compared in terms of performance and not in terms of implementation.

There exists several different video codes that can be used to decode and encode video frames in a stream, such as VP8, VP9, H.264, and H.265 among others. Different encodings can be either lossy or not, allow for different bitrates, quality, and have different speed when encoding and decoding the sent/received frames. Comparisons between encodings have been performed extensively before by other parties and as such is not a topic that is discussed in this thesis. The experiments carried out in this thesis will be done using the codec H.264, which is the most used codec in production.

1.3 Contributions

There have been a few papers published that evaluate WebRTC through different approaches but the target of the evaluations are browser implementations of the WebRTC specification and how they fare on different devices and networks. This report contributes to the research area through extending benchmarking to the .NET framework.

Simon Tenggren contributed primarily to the implementation of interacting with the signaling service, implementing the SIPSorcery package, the tools used to benchmark the implementation, developing the scripts used to extract the data from the benchmarks, as well as presenting the data. Martin Gottlander contributed primarily to researching the benchmarking methods, the GStreamer implementation, and performing the benchmarks. Both authors contributed equally to the report.

1.4 Description of remaining chapters

- Technical Background
  - The technical background describes the technologies that WebRTC use to solve the problem on how to establish peer-to-peer connections in different network and firewall configurations, why they exist, which role they perform and how they affect the quality of service and quality of experience of the end user.
- Approach
  - Describes the methods used to answer the research questions stated above. Describes the methodology behind evaluating the WebRTC implementations, and other quality of service metrics. As well as the implementations themselves.
- Evaluation
  - Discusses and presents the data which resulted from the methodology described in the previous chapter.
- Conclusions
  - The conclusions drawn from the experiments, the possible improvements that can be made in the future, and the authors recommendation to Axis on the best way forward if they decide to use WebRTC in their VMS solutions.

Chapter 2
Technical Background

2.1 WebRTC Related Technologies

The WebRTC standard published its first working draft in October of 2011. [18] Several changes has been made since, and the official standard was published in January of 2021 and is continuously updated. The WebRTC standard is built on several different existing technologies and standards to allow users to establish reliable and secure peer-to-peer connections to exchange live media.

These are primarily the Session Description Protocol (SDP), Interactive Connectivity Establishment (ICE), Session Traversal Utilities for NAT (STUN), and Traversal Using Relay NAT (TURN). The WebRTC standard specifies how and when to use the aforementioned technologies and how they relate to the API. WebRTC also requires the use of a signaling service to exchange information between the peers that wish to establish a connection, however this is intentionally left out of the standard to allow for developers to use which ever method and technologies they prefer, everything from WebSockets to carrier pigeons. [22] This allows for additional functionality to be moved to the signaling service such as authentication and keeping track of the available STUN and TURN servers the peers can use.

What follows is a description of relevant technologies that is required to understand WebRTC and the underlying problem which merits implementing WebRTC, and finally how they come together to actually solve this problem.

2.1.1 Network Address Translation (NAT)

NAT is a router function that modifies the network address in an IP-header and is often used to route traffic from local networks to the public network. A NAT is installed at the gateway to the public backbone network, where all addresses are globally unique. The NAT holds a translation matrix and binds local IP-addresses to a global IP-address. Since translation happens on the gateway where that NAT is configured, the endpoints that are located behind

this NAT does not know which address it can be reached with from the public network. This becomes a problem for peer-to-peer applications where both peers need to find out the address of the other peer to be able to start communication.

A NAT that is configured as dynamic has a pool of public IP-addresses but can reassign the mapping it has between local IP-addresses and public IP-addresses. If the NAT cannot route traffic for an internal host because it currently has no public IP-addresses available, it responds with an ICMP "Destination Unreachable" message. When a public IP-address that has been mapped to a local IP-address has not been used for a timeout duration, the NAT removes this mapping which allows it to use the public IP-address for another endpoint on the local network.

The most common configuration is referred to as Network Address/Port Translator (NAPT) or Port Address Translation (PAT). Using this configuration, multiple local IP-addresses can use the same public IP-address simultaneously through utilizing different ports. This is achieved by mapping the local IP-address that want to initiate communication, to a public IP-address and port. In a short summary, there are two ways this mapping can be done.

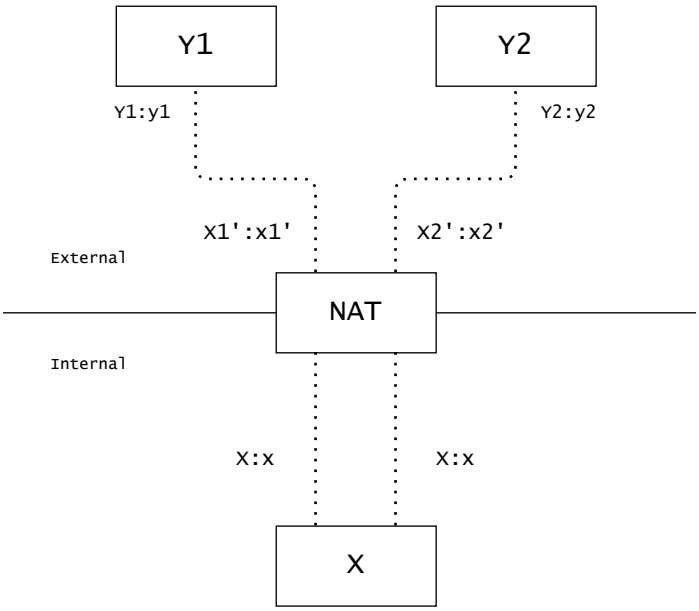


Figure 2.1: NAT Address and Port mapping

Figure 2.1 illustrates a NAT configured gateway working between an internal (local) network and an external (public) network. In the figure, the local endpoint with IP-address X wants to communicate with two different external endpoints with IP-address Y1 and Y2 respectively. When the packets that are destined for Y1 and Y2 reach the NAT the source IP-address and port values in the IP-header are reassigned from X:x to X1':x1' and X2':x2' respectively if the mapping is an *Endpoint-Dependent Mapping*. This means that different endpoints that are communicating with X through the NAT will use a different mapped address and port tuple to reach X.

However, if the mapping is an *Endpoint-Independent Mapping*, the reassigned IP-address and port X1':x1' and X2':x2' would be equal. All endpoints communicating with X through X:x would use the same mapping to reach X.

The NAT also filters inbound packets according to the same two categories. If a NAT is configured to use *Endpoint-Independent Filtering*, it allows all packets through independently of the source of the packet. Meanwhile, *Endpoint-Dependent Filtering* only allows packets from external endpoints that the internal endpoint previously has sent a packet to. These NAT behaviours were originally defined in RFC 4787[9] and the description is derived from that document.

The purpose of this section was to highlight the inherent problem that NATs impose on peer-to-peer connections. If two endpoints both are on a private network behind one or more NATs and they want to establish a peer-to-peer connection, they have to do so by some means of traversing these NATs.

The following sections describe how this is done through the use of the protocols STUN, TURN and ICE in the context of the mappings and filters that have been introduced in this section.

2.1.2 Session Traversal Utilities for NAT (STUN)

STUN enables an endpoint to determine the public facing IP and port that the NAT is using as a mapping for the endpoints actual IP and port. Furthermore it is often used to check connectivity between endpoints and it is also used as a keep-alive to maintain the bindings that a NAT has created to ensure that a binding stays alive as long as needed without the need for re-establishing the connection.

STUN works through STUN binding requests and STUN binding responses. The STUN binding request simply instructs the STUN server to reply with a STUN binding response. When the request passes a NAT, the NAT creates a mapping and forwards the packet with the mapping as the source of the packet to the STUN server. The binding response from the STUN server includes an attribute called "mapped-address" and the server sets the "mapped-address" attribute as the source of the incoming packet. When the originator receives the STUN binding response it learns the address which the NAT has mapped its local address to through reading the "mapped-address" attribute.

Keep-alive is practical because dynamic NATs will remove a mapping as soon as it thinks it is not being used so it can be reused by another connection. Therefore, a STUN server must make sure that this mapping is not removed until connectivity checks are completed. A simple diagram describing the STUN binding request and response can be seen in 2.2.

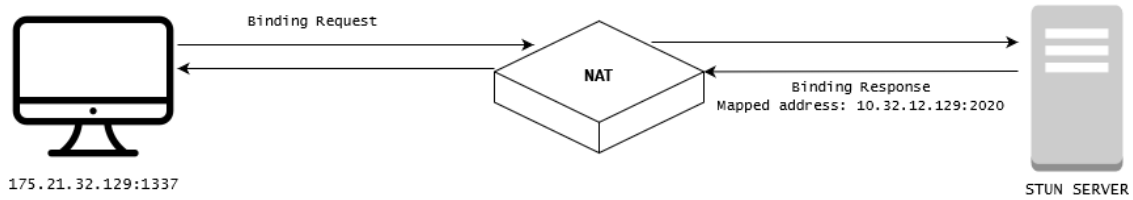


Figure 2.2: An example of the STUN binding request and response.

STUN is run using UDP, however it can also be implemented with the Transmission Control Protocol (TCP) and Transport Layer Security (TLS), which allows for additional

security to be implemented. To distinguish between which underlying protocols are used, when using TCP/TLS the name STUNS is used, while using UDP it is simply referred to as STUN.

2.1.3 Traversal Using Relay around NAT (TURN)

If both endpoints are behind a NAT that is using Endpoint-Dependent Mapping and Filtering, a direct peer-to-peer connection cannot be established. Each NAT that is traversed needs to know the address of the destination address before it can create a mapping for this session which makes UDP hole-punching impossible as this technique relies on both endpoints opening a connection towards the other at the same time. Furthermore, Endpoint-Dependent Filtering makes it impossible to open a session with a server that another endpoint can utilize. With Endpoint-Independent Filtering, an endpoint could ask a public facing server for the address that the server is using to communicate with it and then publish this information for another endpoint to use.

NATs that use Endpoint-Dependent Mapping is observed in less than 20% of cases but if a NAT is using Endpoint-Dependent Mapping it is probably also using Endpoint-Dependent Filtering[20]. As this combination makes peer-to-peer communication impossible, a workaround is needed if the service is going to be delivered at all.

An intermediate server on the public network can be used to relay messages since both peers can only open a connection towards an endpoint on the public internet and the NATs only allow incoming packets from external endpoints that internal endpoints have previously sent packets to. TURN is a relay extension to the STUN protocol that is created to solve this particular problem. A TURN server is a server that implements the TURN protocol and relays messages for clients.

TURN extends STUN through adding Allocate Requests, Allocate Success Response and Refresh Request/Response.

The Allocate Request is used to request an allocation of resources, in the form of a transport address, on the TURN server. The TURN server replies with the address it has allocated for communication and the mapped address it saw as the source of the request. When a peer wishes to communicate with another peer through a TURN server it does this through the allocated relay address. The TURN server holds a mapping between relay addresses and server reflexive addresses to be able to relay communication correctly. Figure 2.3 describes the process of allocating addresses on a TURN server and how the media is later relayed through the allocated addresses. As an allocation has been successfully added, peers can send data through a Send Indication message and the TURN server relays data to the intended endpoint through a Data Indication message. This introduces a problem as Send Indications are not authenticated and a malicious attacker could use Send indications to get the authenticated TURN server to relay for the attacker. This is mitigated through permissions that the client needs to install on a TURN server. This way, the TURN server does not violate the intended Endpoint-Dependent Filtering function that the TURN server traverses.

Just like how STUN can be implemented using TCP/TLS so can TURN, which is referred to as TURNS. This does not add any additional security when relaying the data using WebRTC, since the media is already encrypted using Datagram Transport Layer Security (DTLS), however it can aid in traversing firewalls.

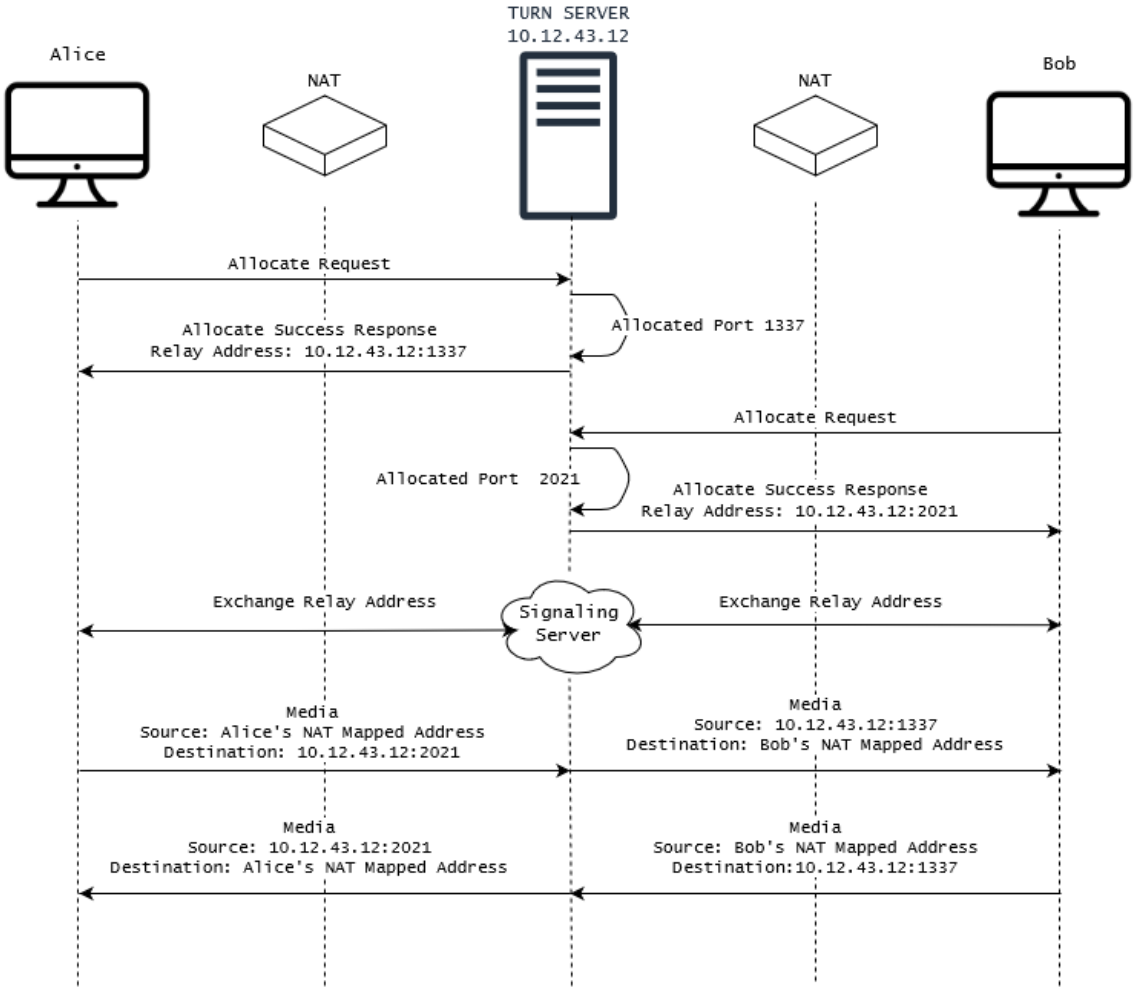


Figure 2.3: Allocating addresses in a TURN server and relaying media through it.

2.1.4 Interactive Connectivity Establishment (ICE)

ICE is a protocol used to find paths through private and public networks to be able to set up peer-to-peer connections. The beginning of this process is to gather ICE-candidates. There are four types of ICE-candidates; host, server reflexive, peer reflexive, and relay. Host candidates are gathered from the physical and logical interface on the host and are the IP-addresses as they are used within the private network. Server reflexive candidates are gathered by making STUN Binding requests to a STUN-server on the public network to be able to discover the mapping that the NAT has assigned between the public IP:Port pair and the local IP:Port pair. Relay candidates are gathered through TURN allocate requests. Relay candidates are the most reliable candidates as these candidates can be used to traverse all mentioned NAT behaviours, however, they are the least efficient because they can introduce latency when relaying data. Peer reflexive candidates stand out because they are or not discovered during the gathering phase but later, during the connectivity checks phase.

After the candidates have been gathered, the candidates need to be exchanged between the peers that want to initiate a peer-to-peer connection. This is done through signaling which, to reiterate, is a channel that both peers can use and it is used to transfer all the

information needed to set up the peer-to-peer connection. Usually this comes in the form of a signaling server.

When both peers have received their respective remote ICE-candidates, they start checking if they can reach the other peer. This is where the gathered candidates come in as they represent the possible ways to connect to the other endpoint. Each peer creates pairs of their locally gathered candidates and matches them to received remote candidates. Each peer use these candidate pairs by sending STUN binding requests from the base of the locally gathered candidate (which then maps to a "server reflexive/mapped address" upon entering the public internet), to the matched remote candidate. If the peer receives this request, the peer replies with a STUN binding response which contains the address that the NAT has mapped for the address of the source sending the request. If this mapped address is not already in the requesting agents list of local ICE candidates, it is created as a peer reflexive candidate and added to this list.

Let's cover a few scenarios to get a feel for how this achieves NAT-traversal.

First consider two endpoints which are both behind NATs employing endpoint independent mapping and filtering. This is the trivial case as each NAT will accept any incoming packets from the other endpoint and a peer-to-peer connection can be made as long as one endpoint knows the mapped address of its peer. If one NAT employs endpoint dependent filtering but the the rest of the configuration stays the same, a direct peer-to-peer connection is still possible. The peer with the restrictive NAT just needs to open a connection towards its peer, before this peer can reach back to the peer behind the restrictive NAT. As both endpoints will try all the candidate pairs, this will eventually happen. If both peers are behind NATs that employ endpoint independent mapping and endpoint dependent filtering, things start to become more complicated. Now both peers need to open a connection toward the other before it can receive packets from the other peer. This is what is known as "hole-punching". To achieve this in a timely manner, connectivity checks are performed in order based on a priority which is included in the ICE-candidates. The peers agree on a priority order through pairing remote and local candidates and combining the priority attribute of the pair with a well defined standardized formula. The priority order is calculated based on the type of the ICE candidate where the recommendation is that host candidates have the highest priority, then reflexive candidates, and finally relay candidates with the lowest priority. Then the peers open connections toward the other according to this priority order. An example of the structure of an ICE candidate, can be seen in figure 2.4.

Gathering of all ICE candidates must be completed before the connectivity checks can be started and this implies several round trips to potentially several STUN and TURN servers just to gather the candidates. To improve time to connection further, Trickle ICE [19] can be utilized. Trickle ICE allows ICE candidates to be sent through signaling as soon as they have been gathered which allows the connectivity check phase to start sooner.

```
candidate:2116 1 udp 659136 10.85.129.66 65396 typ host generation 0
```

Figure 2.4: Example ICE Candidate, with the type host with a priority of 659136.

2.1.5 Session Description Protocol (SDP)

The Session Description Protocol (SDP) [7] is used by WebRTC to exchange information regarding the session that the peers wish to establish. This session is negotiated between the peers to try to establish the various forms of media, encodings, transport protocols, and more. The anatomy of an SDP message consists of several lines of `<type>=<value>` where the type consists of a single case-sensitive character and the value is text where the structure is dependant on the type. Each SDP message has several required and optional type-value pairs. The required types are the following:

- `v`: The protocol version of the SDP message. As of writing the value is always `0`.
- `o`: The originator and the session identifier.
- `s`: The name of the session.
- `t`: The time the session should be active, specified as a start and stop time. If the session is managed by another protocol the value is `0 0`.

Even though not technically required for an SDP message, the type `m`, the media description, is implied if the peers wish to exchange and kind of media stream, be it video, audio, or application data. The media descriptions value is split into the following four sub-fields:

- `<media>`: The type of media, such as `video` or `audio`.
- `<port>`: The port number the media is sent and/or received on.
- `<proto>`: The transport protocol(s) that media sent over, such as UDP or RTP.
- `<fmt>`: The format of the media, depends on the protocol(s) specified in the protocol sub-field. For RTP the value is the RTP payload type number.

The SDP message may also include any number of attribute lines, which further describe the session and/or the media. These are in the format `a=<cat>:<category-value>`. Common attribute lines in the WebRTC sessions describe the framerate for the video or a dynamic RTP payload type which can be used to map from media format value to a encoding and clock-rate. In WebRTC the local ICE candidates that are generated can also be sent as additional attribute lines.

In WebRTC the SDP is used in conjunction with the Session Initiation Protocol (SIP) [6] using the Answer-Offer model. [5] The Offer-Answer model entails that the peers that wish to establish a session has one peer (initiator) who generates an offer describing the session they wish to establish, the receiving peer (respondent) receives the offer and generates their answer and sends it to the initiator.

The answer that is generated depends on the offer that was received, for example if the offer specifies a media stream that the respondent doesn't have the capabilities of providing the offer may be rejected when it arrives. For example if the initiator wants to establish a video stream, but the respondent has no possibility of generating such a stream, the offer may be rejected immediately. This can only occur once and is not an ongoing negotiation, if the answer or offer is not a session that can be provided the answer or offer is rejected

and the process has to restart again. The respondent may change several of the lines session description present in the offer. Importantly the respondent may remove codecs and/or protocols from the media descriptions that they cannot use for various reasons, such as lack of implementation.

An example WebRTC offer received from a network camera can be seen in appendix figure B.1.

2.1.6 Simple WebRTC Connection Establishment

The above mentioned technologies all come together to solve the issue of creating a reliable peer-to-peer connection which can stream live media, which in turn forms the WebRTC standard.

WebRTC in its simplest form can be described using a simple triangle model, where the peers sit at each base of the triangle while the signaling server sits at the top. (See figure 2.5) Both peers have information on how to reach the common signaling server and how to communicate with it. The signaling server has one job, to relay information necessary for establishing a peer-to-peer connection in a structured manner. There are usually steps before the WebRTC session starts which entails initial contact with the signaling server to register each peer so that the signaling server can relay messages to the peers when the actual session starts, however this step is highly specific to the implementation of the signaling server, and is as such not worth discussing further.

The WebRTC session starts by one of the peers (initiator) generating an SDP offer which contains a description of the session that it wishes to establish with the other peer (respondent), this offer is then relayed through the signaling server. The respondent generates an SDP answer message upon receiving the offer. The generated answer depends on the content of the offer and the capabilities that are available for the respondent, such as sending video or audio with specific codecs. This message is again relayed through the signaling server, and the initiator can verify that the answer is still valid. Thus the peers that are initiating a session have an agreement on how the media can be transmitted. Once the offer has been sent by the initiator, or received by the respondent, the peers can start gathering their candidates, by contacting their ICE servers, such as TURN and/or STUN they might be aware of, as well as gathering from their local interfaces. These candidates are relayed through the signaling server as they are gathered, and received at the other peer. This process is often done by both peers simultaneously. After the peer receive the other peers ICE candidates they can start performing connectivity checks, until they find a pair that can be used to send media. This check uses the ICE candidates priorities set by the respective peers to establish the connection that they both would prefer and is possible depending on the NAT and firewall configurations that both peers are behind. Once this process is done, either a connection can be established and the peers can start sending media directly to each other, or no connection could be established due to the candidates sent are not sufficient to support the connection due to various reasons such as NAT or firewalls prohibiting the connection.

Figure 2.6 describes the process of establishing a peer connection. Once the peer connection is established the signaling server is technically no longer needed unless the peers are disconnected and the session has to be restarted, or the peers wish to renegotiate their peer connection.

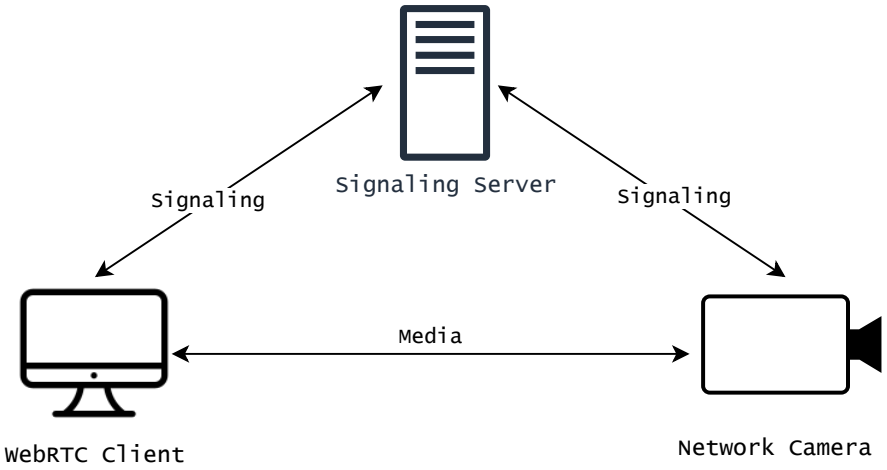


Figure 2.5: WebRTC Triangle Model

2.1.7 WebRTC at Axis

Axis have established infrastructure to support WebRTC, including private STUN, TURN, and signaling servers, as well as support for streaming data, audio, and video from some of their latest devices. Recently support for tunneling HTTP data over the WebRTC data channel was released for ACC which allows users to manage devices which support WebRTC, in an embedded web browser by forwarding the HTTP traffic received through the WebRTC data channel. Attempts have been made to allow streaming of video in their VMS using the open source, Microsoft developed, Mixed-Reality WebRTC package, but limitations of unsupported video codecs for WPF have made the efforts difficult and time consuming. Recently the use of WebRTC has been implemented for its capabilities to act as a proxy where users can use the HTTP traffic transferred by the data channel to interact with the cameras settings. This feature has already been released earlier as a part of ACC.

2.2 Current Streaming Solutions at Axis

Axis have several different ways of streaming video from the network cameras to different applications, one of the more common ones is to run a proxy program which relays traffic on a specific port to a relay server which in turn connects to a webserver located on the network camera. The camera responds with video inside a Matroska container, which is continuously streamed through the response body. This process is fast and flexible, allowing it to be rendered in both Axis VMS solutions and even web browsers which has support for depacking Matroska and decoding the encoded video data. However the solution is limited, allowing only for a one-way channel where only video can be transmitted. The proxy itself is also quite self contained and requires additional configuration and registration of the devices to function. This is achieved through additional HTTP requests which are sent to the proxy.

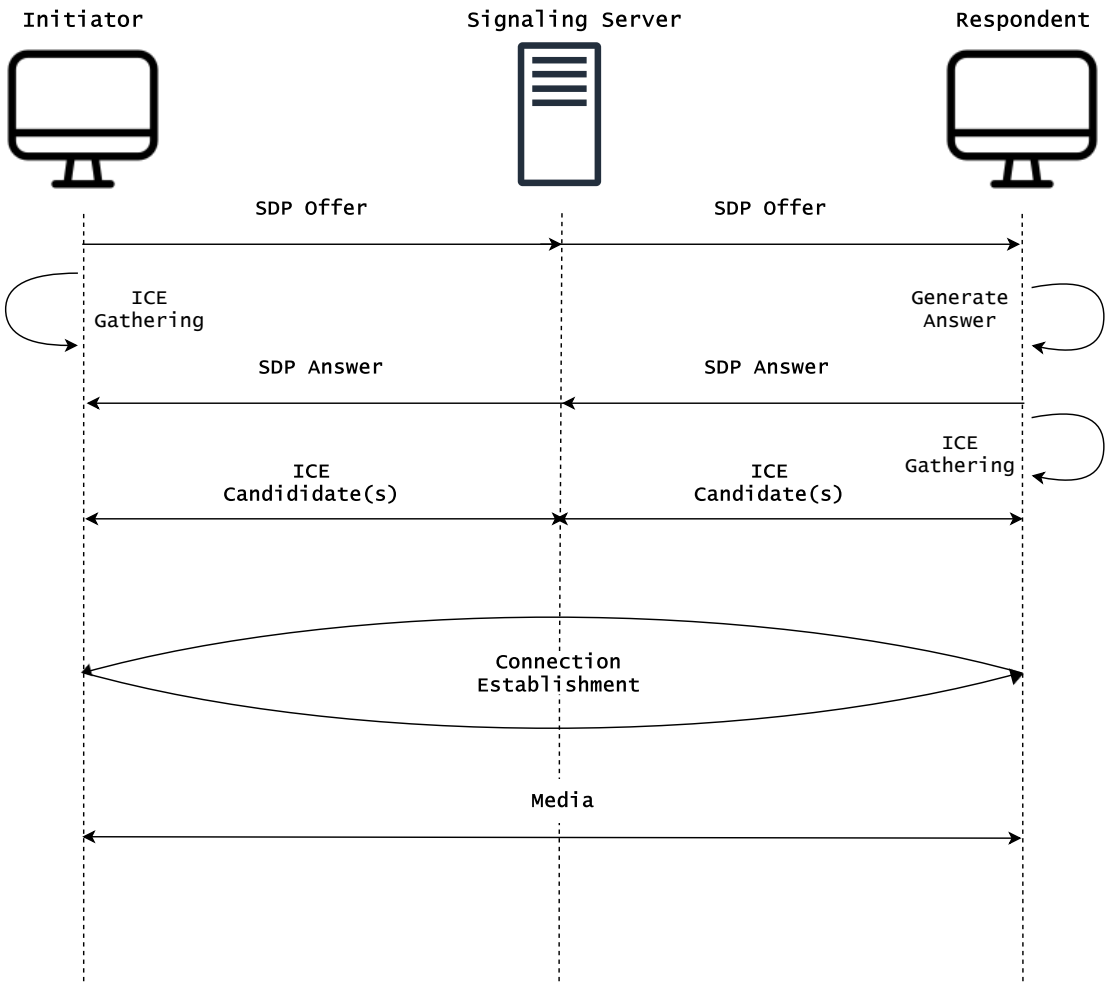


Figure 2.6: Simple Peer Connection establishment timeline detailing the interaction between the peers and the signaling server.

Chapter 3
Approach

3.1 Method

There are several interesting metrics when evaluating the performance of a streaming solution, concerning both the quality of service (QoS) and quality of experience (QoE). For QoS the metric measured is latency, while for the QoE metrics the time-to-stream is measured. Additionally the CPU and memory usage is measured to ensure that the WebRTC streaming solutions are not too resource expensive compared to Axis contemporary streaming solutions. The end customer should be able to run WebRTC in their VMS software without noticing a significant difference in performance.

As such an implementation is developed in the form of a WPF application using several common components and easily isolated and exchangeable implementation specific components that can be used in evaluating the various QoE and QoS metrics, as well as the CPU and memory usage metrics for the different WebRTC implementation and the current Axis developed streaming solution.

3.2 Implementation

Three different implementations of receiving and rendering H.264 video in a .NET Framework 4.8 WPF applications were developed, using two different open source projects SIPSorcery, and GStreamer. As well as the Axis proprietary HTTP Proxy solution as a reference to evaluate WebRTC. The WebRTC implementations utilize common components, such as interacting with the signaling server using a predefined protocol, and using the Axis authentication service. The application is a proof-of-concept for streaming and rendering video to the user using the .NET environment currently in use at Axis. The application is built upon existing Axis components, such as their dependency injection package, UI package, and rendering and decoding package. This is all to mimic the actual environment that is cur-

rently used in Axis' VMS solutions. The dependency injection package also allowed for easy swapping of implementation specific components, such as the WebRTC implementation by simply changing a single line and recompiling the program.

H.264 Encoding

H.264 encoding was used for both WebRTC implementations, as well as the HTTP proxy to which they are compared. H.264 is a well supported (92% support by developers in 2018)[8] ISO-standardized video encoding standard is designed to be used in several different areas, from live video conferencing, to high definition Blu-Ray recordings. [17] Since the different use cases for H.264 vary and have different requirements, the H.264 standard supports several different profiles for these different use cases. There are three main profiles in H.264; baseline, main, and high, which all have their variations. The baseline profile is the most basic of the profiles, best suited for embedded devices which have to make the most use of their limited power. The baseline profile requires less computation to encode the video, however the compression rate will suffer as a result. Generally using the main and high profiles will require more of the encoders and decoders while using less bandwidth due to the higher compression rate.

In H.264 the encoded frames have three different types, namely Intra-coded (I) frames, Predictive (P) frames, and Bi-predictive (B) frames. The I-frame acts as a checkpoint and contains all the necessary information to render a single image. The P-frames include the changes in the image data and can be used in conjunction with other frames to create a new image using a previously encoded frame. B-frames are similar except that they use two frames as reference points instead of one. A stream using the baseline profile will not use the B-frames and only relies on the I- and P-frames. Generally the more total B-frames the better the video compression rate. As the frames differ in function they also differ in size and is dependent on the stream that is captured. The size of the frames vary, where I-frames are the largest since they contain all the information necessary to decode and render a single frame, the predictive frames however depend on the previously used frames as well as the changes in the image, e.g. a stream with lots of movement result in larger will generally result in larger predictive frame, while a stream with small to low amounts of movement will result in smaller frames.

3.2.1 Common Components

There are components reused in both implementations, and these are components related to Axis infrastructure and rendering solutions.

The common components are the following:

- Authentication Service
- Signaling Service
- Decoding and Rendering pipeline

An image describing the implementation and the common components can be seen in figure 3.1.

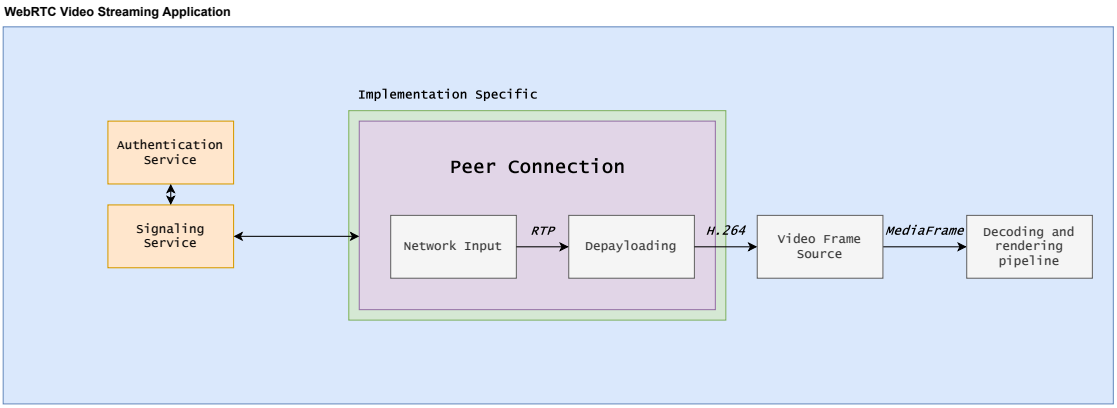


Figure 3.1: An overview of the components in the implementations.

The common components are included in some of the measurements, with the exception of the authentication service which could be isolated and ignored in all of the measurements. The rendering pipeline is a necessary part of measuring the latency, since the decoding and rendering of the frames are taken care of here. The signaling service is an essential part in the Time to Stream measurement, since it is essential in relaying messages between the peers to establish a peer-connection.

Authentication Service

The authentication service allows the user to authenticate itself to be able to interact with both the signaling server and the devices to which he wishes to establish peer connections with. This process is currently very manual, requiring an access token to be generated by manual username and password authentication at least once every hour. However this can be done ahead of time of any experiments, allowing this process to be isolated, and as such the procedure is not reflected in any of the experiments.

Once the authentication procedure has been performed the client has been supplied with a token that allows them to interact with the signaling server. The signaling service uses this token to verify that the client has registered the camera that it wishes to establish a WebRTC session with and that the camera is registered with the signaling server. If the token is valid the signaling can proceed and the session can be established.

Signaling Service

For signaling between peers and server to function, a common protocol must be used. This is left out of the WebRTC standard by design choice and can be implemented in what ever manner the developers may choose. At Axis a signaling server and its protocol was already established which uses WebSockets to communicate with both the client and the camera. Two important and notable features of the signaling service includes the validation of the access token received from the authentication service, as well as assigning the STUN and TURN servers to both peers. As such the peers do not need to make any separate look-ups to other services or have a predetermined list of possible ICE servers that may or may not be available.

The Decoding and Rendering Pipeline

Axis uses a common rendering and decoding pipeline in several of their available VMS solutions, this pipeline will not be discussed in detail due to issue of possibly disclosing several techniques that might hurt competitive advantages, but an abstracted view will be given to supply the reader of an overview of how it relates to the work the authors have implemented. The rendering pipeline uses several proprietary data structures and formats to determine how and when to render specific frames which are received through different media connections. The pipeline is constructed of a chain of several components which manipulate the data into a format which can be rendered in a WPF application using Axis' UI packages. The pipeline needs two important types of information from any media connection using H.264 encoded video, the sequence parameter sets (SPS) and picture parameter sets (PPS), as well as obviously the H.264 frame data. The SPS and PPS contain information concerning how H.264 frames should be decoded and rendered, such as the resolution and profile information that the video is encoded with. For WebRTC these can be found in the SDP offer received from the network camera and are encoded as a Base64 string, and can be seen in the example SDP offer in figure B.1. This information is extracted and converted into an Axis proprietary format, called a MediaInfo-frame. It is important that this frame is passed before any H.264 frames are, otherwise it is impossible to decode and render the frames properly as a live video.

The H.264 frames need to be aligned properly for the information in the MediaInfo-frame to be valid, as such the raw H.264 data has to contain the NAL unit header which describes the type of frame that is sent through the pipeline (I/P/B frames), the I-frames additionally need to be marked as such. When the H.264 has been extracted and marked they are converted to another of Axis proprietary formats, a MediaFrame which can be sent through the pipeline.

For WebRTC, both the extraction of the SPS, PPS, and the depacketization of the raw H.264 data is dependent on the implementation. The HTTP Proxy solutions for the same process are already implemented by Axis.

Implementation Specific Components

Both WebRTC implementations have components specific to themselves, that relate to the WebRTC internals such as networking, depayloading the received data, as well as managing the peer connection. There are several features and differences between the SIPSorcery and GStreamer implementations. These differ from the language which they are implemented in and the support of different network protocols. A list detailing some of the most notable differences can be seen in table 3.1.

Feature	SIPSorcery	GStreamer
Native C#	X	
IPv6		X
TURN	X	X
STUN	X	X
RTP Jitter buffer		X
TCP/TLS		X
UDP	X	X

Table 3.1: The features present in the WebRTC implementations

3.2.2 SIPSorcery WebRTC Implementation

SIPSorcery is an open source library which aims to simplify the process of real-time communications in .NET applications, supporting audio, video, and data transfer depending on the intended usage.

SIPSorcery notably supports WebRTC and closely follows the official API documentation by W3C [27] making it simple for developers to implement a working WebRTC solution if they are already familiar with how it is done in modern web browsers.

SIPSorcery is a pure C# library without the use of any wrappers making it easy to implement into existing .NET applications. It also flexible in how it is used, allowing users to specify many common codecs for audio and video, even though it might not support decoding and/or rendering them itself, but being able to produce the correct SDP answers and offers to negotiate the media transfer. When the media later arrives, e.g. encapsulated in a RTP packet, the user can subscribe to the received packets and themselves use a custom depayloader to handle the received encoded media, however for H.264 payloads SIPSorcery has support for extracting the encoded video frames which are present in one more RTP packets.

SIPSorcery however lacks some critical features which are necessary for a complete WebRTC implementation, most notable SIPSorcery have no support for TLS, which results in that the TURNS and STUNS protocols not being supported, only supporting regular TURN and STUN. This can be crucial when considering privacy, as the stream is at risk for leaking the IP-addresses of the peers when relayed. Note that the data content itself will be encrypted and that this is a privacy concern and not a security concern.

It also currently has no support for generating IPv6 ICE candidates, limiting the number of possible host candidates that can be used.

When streaming video over RTP, SIPSorcery currently has no jitter buffer implementation. A jitter buffer is used to reconstruct the RTP packets in the order they are meant to be received, as well as remove any possible duplicates. A jitter buffer is often used when streaming over best-effort protocols such as UDP to improve the user experience by intentionally causing a small delay to reconstruct the stream as it was intended by adding the RTP packets in the correct order according to their sequence number. Without a jitter buffer encoded frames can be decoded and rendered out of order causing video artifacts, or the stream rendering entire frames out of order. It is especially important when streaming encoded video such as H.264 where it is crucial that the frames arrive in the correct order.

Using SIPSorcery

SIPSorcery is available to download and use in .NET application through the NuGet package manager, the version evaluated in this thesis concern version 5.2.3 released in June of 2021. To establish a peer connection in SIPSorcery the only requirement is the `RTCPeerConnection` object that manages SDP, STUN and TURN servers, and ICE candidates. The peer connection object is entirely event based and fires events when needed, such as when a local ICE candidate is ready to be sent via the signaling server. The developer, just like how it behaves in WebRTC in browsers, is responsible to capture these events in the application and forward it to the signaling server. The opposite is also true. When video data is available it is also fired as an event, in this case as a H.264 video frame reconstructed from one or more RTP packets. When the frame is ready SIPSorcery represents it with several parameters, however the most

important of which is the timestamp of the encoded frame, as well as the encoded frame in the form of any array of bytes. This information can then be used to pass to video decoding and rendering pipeline.

Optimizing Connection Establishment Time

For a WebRTC connection to be established a X.509 certificate needs to be created and signed. This is to ensure that the traffic sent between the peers is encrypted using DTLS. The default in SIPSorcery is to create a new self-signed certificate for each connection that is made, in the constructor of the PeerConnection-object. This is a costly operation that hinders the overall time for the connection to be established. However the certificate can be created ahead of time, and still be unique for each connection. This reduces the time that is needed for the connection to be established significantly. As such this optimization was performed in the measurements taken.

3.2.3 GStreamer WebRTC Implementation

GStreamer is a multimedia framework that is based on linking media processing elements into a pipeline to complete a media related workflow. The elements that constitutes a pipeline contains pads and filters. The pads are the interface to the outside world in the form of sink pads and source pads where the sink pad is the pad that receives data while the source pad is the pad that output data after the data has been processed by the filter within the element. An element can contain multiple source pads and/or sink pads which is useful for multiplexing audio and video into a single stream. In our case, video and audio is received on the same channel and needs to be demultiplexed before it can be decoded and rendered.

GStreamer has a plug-in element called webrtcbin which is an abstraction of the technologies that WebRTC builds upon and is used to make it easier for developers to create WebRTC applications through GStreamer. Moreover, GStreamer already had support for the technologies that WebRTC builds upon before WebRTC was even drafted and webrtcbin actually just connects these technologies in a single interface to conform with the WebRTC standard. This means that RTP (and RTCP) over UDP (or TCP) exists within webrtcbin as well as the ICE agent that is responsible for creating STUN and TURN messages to gather ICE candidates and check for connectivity. In essence, it does everything the PeerConnection defined in the ECMAScript API can. The sinkpad(s) of this element receives media from a device or file and the sourcepad(s) outputs data that has been received on the negotiated channel. As there can be multiple channels open at the same time and since each channel can be have a multiplexed stream, we need to subscribe on each pad that the webrtcbin creates. In 3.2 it is shown how a multiplexed stream with audio and video is demultiplexed within the webrtcbin and how we subscribe on the source pads that are created. As rendering functionality comes from an Axis proprietary solution we let GStreamer unpack the payloads and then sink the raw bytes into an element called appsink. From appsink we can emit the raw bytes and pipe it into the Axis proprietary solution for decoding and rendering.

The webrtcbin plugin is labeled by the GStreamer community as "bad" which denotes that it is not up to par with the standard of GStreamer (missing documentation, tests, and/or active maintainer) and if the plug-in breaks, it is up to the community to patch it. The

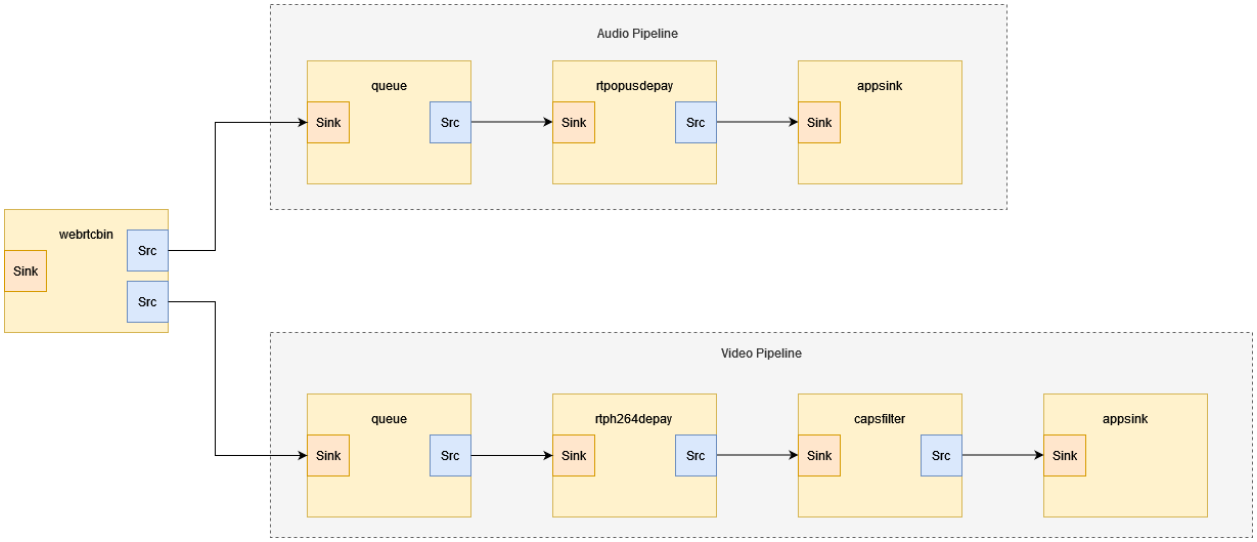


Figure 3.2: GStreamer Pipeline for video and audio

somewhat lacking documentation can make this plugin hard to work with, especially when the developer is not used to the GStreamer framework.

GStreamer is written in C but has bindings for C# in the library GStreamer-Sharp which are the bindings that are used for the experiments carried out for this thesis.

The HTTP Proxy Implementation

To compare WebRTC as a viable streaming solution one of the many peer-to-peer streaming solutions used at Axis is implemented in the application. This comes in the form of a HTTP Proxy which relays traffic from the network camera to the application using H.264 encoded video contained inside the Matroska container format. [24] The proxy runs as a separate process which is spawned while running the program. The HTTP Proxy comes with much of the functionality built into the WebRTC standard, but is not built on any publicly available standard itself. For example the data is relayed much like it would be when relaying data through a TURN server, but instead uses a proprietary solution which is not built to the TURN specification.

The HTTP Proxy uses the same common components above, with the exception of the signaling server. Instead the HTTP Proxy is configured using several HTTP request. One to configure the server to relay the data, one to add the network camera as a remote peer.

The stream is also supplied through HTTP request, where the response body contains the stream which is continuously read from and fed into the Axis decoding and rendering solution.

Unlike the WebRTC implementations, the HTTP Proxy can be configured in much more discrete steps, where the connection can be established first, before requesting the actual stream. To give the a fair comparison between the implementations however, the entire time to stream is measured as the time it takes to configure the HTTP proxy, with the exception of the authentication service, until the HTTP request which contains the stream can be read and fed into the decoding and rendering pipeline.

An overview of the configuration and request of video streams in HTTP proxy can be

seen in 3.3

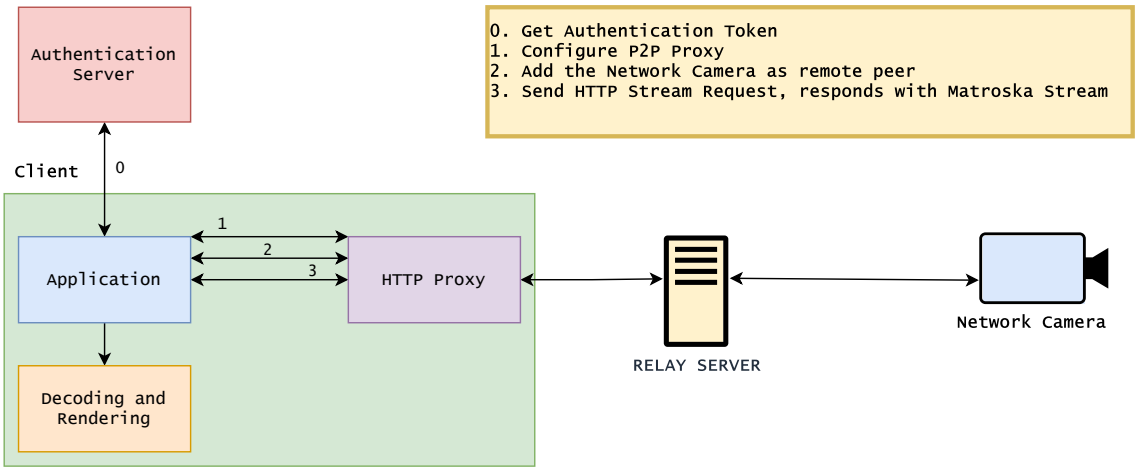


Figure 3.3: Overview of the HTTP Proxy used to retrieve Matroska streams.

Differences between Network Setups

The peer-to-peer connection in WebRTC can vary depending on the network the peers are located on as well as the type of the peer connection that was established, of particular interest is the difference between using TURN to relay the stream, compared to using a direct connection. This can be controlled by changing the transport policy that is being used for the peer connection in the WebRTC implementations. To change which type of connection that is used, the choice between two transport policies is available, the *all* and *relay* transport policies. When using the *all* transport policy the WebRTC implementations will consider all types pairs that can be used, that includes host, server reflexive, and relay connections. If the networks allow for it, a host connection will always be preferred and used due to the host candidates being given a higher priority. However if the *relay* transport policy is used, the implementations will only consider candidates which use TURN to relay the data, by only producing candidates with the relay attribute and only testing the connection on received ICE candidates which also posses the relay attribute. Even though a TURN server does not do any processing of the media data an additional delay can be expected both when establishing the connection due to the process of authenticating against the TURN server and allocating the relay address.

3.3 Theory

The measurements are taken using different methods, which are described below.

3.3.1 Evaluating Managed Languages

Important to note is that C# is a Just-In-Time (JIT) compiled language and is also managed, in this case using the .NET Framework. Measuring performance in JIT compiled and man-

aged languages are different from measuring in Ahead-Of-Time compiled and un-managed languages such as C or C++. [23] As the JIT compiler will optimize code on the fly and the runtime management will perform garbage collection at non-deterministic times during the execution of the program. As such a proper benchmark should measure both the *warm-up* state of the program, where the JIT compiler has yet applied much of its optimization, and the *steady-state* performance where optimization is finished and the performance improvement is statistically determined to be insignificant. There is however some limitations which did not allow us to measure the steady-state performance during the benchmarks, namely that the network camera is limited in its capabilities and cannot supply us with the many connections that is required for the .NET application to reach it's steady-state. There is also the issue of measuring performance of the application when one of inescapable variables that affect the performance is the network which is used. The measurements could therefore only be taken from the first connection that is established when the program is run for the first time, this done several times to get data to see the worst and best case performance for the start-up state of the application. All the benchmarks were measured with the programs compiled with release compilation options, which performs some minor ahead-of-time optimizations, such as dead code elimination.

3.3.2 Evaluating the WebRTC implementations

To evaluate and find potential bottlenecks in the SIPSorcery and GStreamer WebRTC implementations a method closely relating to the method used in WebRTCBench was developed. [11]. WebRTCBench was developed for a similar use case, namely to evaluate the different WebRTC implementations present in the web browsers Mozilla Firefox and Google Chrome. WebRTCBench is used to measure several primitives relating to the internals of the implementations and how well they perform, these primitivies are isolated as much as possible from variables such as the network they are located on. These measurements are taken as the difference between timestamps at different events relating to a specific part of the implementation. These measurements relate to the overall performance of a WebRTC implementation, such as the time to perform the hole-punch procedure which is necessary for the peer connection to be established. WebRTCBench was written for web browsers and couldn't be directly ported to .NET, however the same methodology and many of the same measurements is applied to measure the internals of the SIPSorcery and GStreamer WebRTC implementations. A table detailing how each primitive is measured, and what it measures is found in 3.2.

It is important to note that several of the metrics overlap each other, therefore the combined time does not represent the total time it takes to establish a peer-connection, but all steps are needed for it be established.

3.3.3 Time to Stream

The time to stream is measured in a similar way to the WebRTCBench primitives for the WebRTC implementations. An additional timestamp is taken when the signaling server is initially contacted to start the session, and the timestamp when depacketization has finished were used to see when the Axis decoding and rendering pipeline can start reading the frames. For the HTTP Proxy the timestamp is taken instead when the first configuration message is sent to the HTTP Proxy, and then another timestamp is taken when the HTTP Stream

WebRTC Implementation primitives			
Measurement	**Beginning Event**	**Ending Event**	**Description**
Initialize Peer Connection	Before creating a new PeerConnection object.	After creating a new PeerConnection object.	The time it takes to initialize a new PeerConnection object.
Offer Time	Before setting the offer SDP in the PeerConnection object.	After setting the offer SDP in the PeerConnection object.	The time to parse and set the remote description generated by the peer.
Answer Generation	Before generating the answer.	After the local description has been generated and set.	The time to generate and set the local description. (SDP Answer)
ICE Hole Punch Time	The first ICE candidate from the remote peer is added to the PeerConnection object.	The peer connection has been established.	The time it takes to find a connection between the peers.
Depacketization Time	First RTP packet arrives.	The first encoded H.264 frame is extracted.	The time it takes to depacketize one or more RTP packets to extract a full H.264 frame.

Table 3.2: List of the measurements to evaluate the WebRTC implementations.

response has been resolved and the response body can be read. As such the Time to Stream can be calculated using 3.1 for the WebRTC implementations and using 3.2 for the HTTP Proxy.

$$TTS_{WebRTC} = t_{First\ depacketized\ frame} - t_{Contact\ Signaling\ Server} \tag{3.1}$$

$$TTS_{HTTP} = t_{HTTP\ Stream\ Response\ Resolved} - t_{Start\ HTTP\ Proxy\ configuration} \tag{3.2}$$

3.3.4 Latency

The method for evaluating the end-to-end latency was recommended by employees at Axis, as it accurately reflects the perceived latency by the user of the VMS solutions. It consists of using a network camera to record its own stream, thus showing the stream that is being rendered as the stream is simultaneously being recorded and streamed. Both the "real" and rendered stream have timestamps which shows the time to the millisecond. The latency is then derived from the difference between the "real" stream and the stream that is rendered. This is done by recording five minutes of continuous streaming with the screen recording tool OBS Studio [3], which records the stream at 60 frames per second. Measurements are taken from when stream starts, and then every 30 seconds to ensure that the streams latency does not vary drastically during the duration of the experiment.

A figure of the method used to measure latency can be seen in figure 3.4 where the upper timestamp corresponds to the "real" stream and the timestamp below corresponds to the rendered stream. In the figure a latency of 660 milliseconds can be observed.

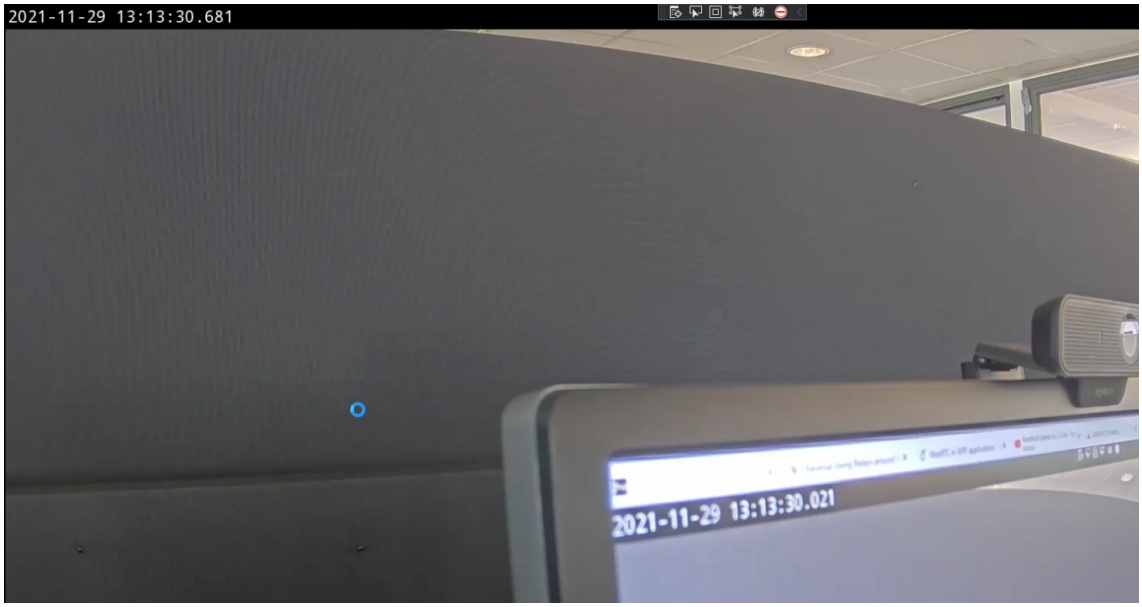


Figure 3.4: The setup used to measure the latency.

3.3.5 Memory Usage

The memory usage is measured using the dotMemory profiling software developed by JetBrains. [1] dotMemory attaches itself to the running application and measures both managed and unmanaged memory. With dotMemeory the possibility of taking snapshots of the memory usage at different moments can be utilized. A conditional snapshot option was used to capture the memory usage of the different streaming solutions every minute of a five minute stream. This is also used to identify any potential memory leaks in the implementations. The memory is measured in total megabytes used by the running process, which includes the entire application, including the decoding and rendering pipeline and in the case of the WebRTC implementations, also the signaling service.

3.3.6 CPU Usage

To capture the average load of the CPU for the different streaming solutions the profiling tool dotTrace developed by JetBrains is used. [2] dotTrace much like dotMemory attaches itself to the running process, but instead of profiling the memory usage, instead collects statistics on the CPU, such as the most expensive functions which is usually used to find information which can be used to optimize the program. Importantly for our measurements however is that the total load the program causes on the CPU is also captured. The load is measured from the start of the program and during a stream for a duration of six minutes. dotTrace can then be used to find the average CPU load in a window of time specified by the user. When the program is first starting the CPU usage spikes, which could be the result of a great number of things, such as the JIT compiler performing optimizations. After a while the program reaches a steadier state, where the CPU usage is stable. As such two measurements are included, the minute where the stream is starting up, and the last five minutes where the CPU usage is in a streaming in a steadier state. See figure 3.5.

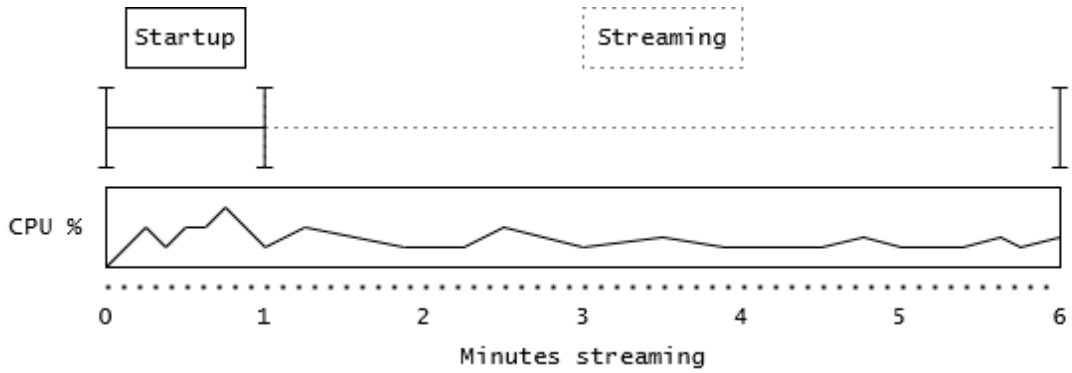


Figure 3.5: Visualization of the CPU Usage measurement.

Chapter 4
Evaluation

4.1 Experimental Setup

To be able to test the connection between using TURN relay and a direct peer-to-peer connection the ICE agent was restricted to one of two transport policies, *relay* or *all*. If the ICE agent is configured to use *relay* as its transport policy it will only gather relay candidates, and if it use *all* as its transport policy it will gather all candidates it can. As relay candidates have the lowest priority and host candidates have the highest priority, the *all* transport policy yielded connections established using host candidates. In a real life scenario, the transport policy would most likely always be set to *all* since it is preferable to use a host or server reflexive candidate over a relay candidate. However not all connections can be established using only host candidates, which is why it important to test the differences when using a TURN server, and using the different transport policies achieves this without the need to emulate different networks.

The transmitting network camera used was the Axis M1135, which streamed 1920x1080p resolution video.

4.2 Results

The results for the experiments can be seen below.

4.2.1 WebRTC Primitives

The results for the experiment evaluating the primitives for the WebRTC implementations, inspired by the WebRTCBench methodology, for both transport policies can be seen in table 4.1.

The experiments revealed that the largest difference when changing the transport policy from *all* to *relay* was the time for ICE Hole punching, where it increased the average time by 1775.34 ms, and 425.91 ms for the SIPSorcery and GStreamer implementation respectively. The other significant changes were seen in the depacketization time for the SIPSorcery implementation when using the *relay* transport policy, where depacketization time was increased by on average 49.1 ms, and the initialization of the peer connection object in the GStreamer implementation where the change of transport policy to *relay* had a minor increase.

The other primitives did not show a significant change while changing the transport policy.

Mean±SEM result times of WebRTC Primitives					
Implementation and Transport policy	**Init. Peer Connection**	**Offer Time**	**Answer time**	**ICE Hole punching**	**Depack**
SIPSorcery All	74.63 ± 1.37 ms	153.76 ± 3.04 ms	32.31 ± 1.50 ms	209.83 ± 3.08 ms	14.45 ± 1.40 ms
GStreamer All	49.72 ± 1.43 ms	300.51 ± 43.39 ms	93.82 ± 2.34 ms	155.14 ± 8.53 ms	14.86 ± 0.47 ms
SIPSorcery Relay	78.90 ± 2.49 ms	152.10 ± 3.57 ms	28.14 ± 1.00 ms	1985.17 ± 203.19 ms	63.96 ± 4.88 ms
GStreamer Relay	62.94 ± 9.82 ms	294.69 ± 28.40 ms	92.69 ± 6.25 ms	581.05 ± 17.44 ms	13.57 ± 0.60 ms

Table 4.1: The resulting Mean ±Standard Error of the Mean (SEM) times for the different WebRTC primitives

4.2.2 Time to Stream

The time to stream was measured 30 times for both WebRTC implementations with both of the transport policies, as well as the HTTP Proxy. A box plot, including outliers found in the measurements can be seen in figure 4.1. While a table of the data can be found in A.4. The experiment showed that the WebRTC implementations outperformed the HTTP Proxy when using both transport policies.

The median time to stream was 1211.7 ms for the GStreamer using All, 929.0 ms for SIPSorcery using All, 1802 ms for GStreamer using Relay, 3717.15 ms for SIPSorcery using Relay, and 5193.95 ms using the HTTP Proxy. Using SIPSorcery with the relay transport policy gave an increased spread, resulting in an Interquartile Range (IQR) of 1999.72 ms. The other measurements had a comparatively low IQR, where the IQR was 204.25 ms for GStreamer All, 170.43 ms for SIPSorcery All, 202.3 ms for GStreamer Relay, and 172.13 ms for the HTTP Proxy.

These metrics also show that using the relay transport policy effects the time to stream significantly, in the case of the GStreamer implementation mostly in the actual time to stream, with a difference in median time to stream of $1802ms - 1211.7ms = 590.3ms$. And in the SIPSorcery implementation it manifested in an even larger difference with $3717.15ms - 929ms = 2788.15ms$ and with a much larger spread between the values.

The data gathered on SIPSorcery with the relay transport policy, split itself into two very distinct groupings, where the 14 lowest values were observed between $1401ms$ and $1946ms$, while the remaining 16 values ranged between $3717.1ms$ and $4488.5ms$. This leaves a gap of about $1700ms$ were no values were observed.

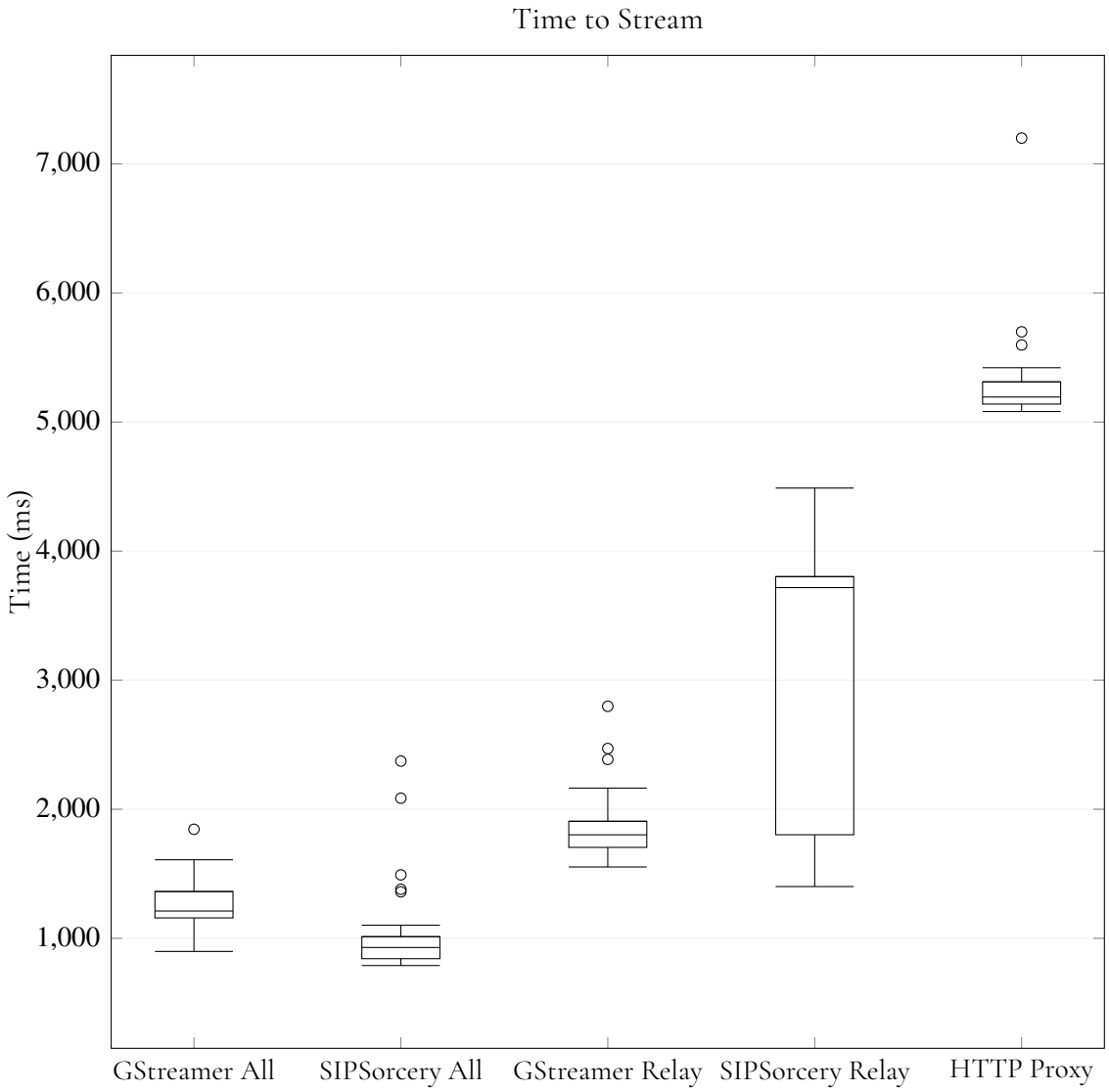


Figure 4.1: The Time To Stream for the three streaming solutions with different transport policies.

4.2.3 Latency

The stream latency was only measured once for each of the implementation and both the transport policy, and in the case of the GStreamer implementation another two measurements with a RTP jitter buffer configured with an extra 200 milliseconds of delay, to examine if the stream quality was effected. The results from the stream latency measurements can be seen in figure 4.2 and the raw data can be seen in table A.6.

The best (lowest) mean latency was observed in the GStreamer implementation using the all transport policy without a jitter buffer, where mean latency was observed to be **397.5** milliseconds, while the worst (highest) latency was observed in the GStreamer implementation using a 200 millisecond jitter buffer using the relay transport policy, with a mean latency of **631.1** milliseconds.

In both WebRTC implementations using the relay transport policy increased the ob-

served latency. In the case of the SIPSorcery implementation the mean latency compared to using the all transport policy increased by 21.3 milliseconds, resulting in an observed mean latency for the relay transport policy of 421 milliseconds. While the GStreamer implementation the mean observed latency increased by 42.7 milliseconds, resulting in a mean latency using the relay transport policy of 440.1 milliseconds.

As a reference using the HTTP Proxy, a mean latency of 418.9 milliseconds was observed.

During the experiment the streams did have variations in the latency, where the largest difference during the stream could be observed in the GStreamer All with the jitter buffer, where the lowest observed latency was 549 milliseconds, and the highest was 664 milliseconds. This was followed by the HTTP Proxy where the lowest observed latency was 364 milliseconds, and the highest was 366 milliseconds. The most stable latency was observed in the SIPSorcery implementation using the all transport policy, where highest observed latency during the stream was 401 milliseconds, while the lowest was 397 milliseconds.

The variations where larger when using the relay transport policy for both WebRTC implementations, where changing the transport policy in the SIPSorcery implementation caused the variation during the stream to increase from 4 milliseconds using the all transport policy to 82 milliseconds using the relay transport policy. In the case of the GStreamer implementation without a jitter buffer, the change caused the largest variation of latency to change from 40 milliseconds to 87 milliseconds when changed from the all transport policy to relay.

4.2.4 Memory Usage

The memory usage was measured five times for each implementation, the average for each minute of streaming for each of the implementations can be seen in 4.3, and individual experiments for each of the implementations can be seen in figures A.1, A.2, and A.2. The memory usage showed that the WebRTC implementations both had higher memory footprint compared to the HTTP Proxy. The GStreamer implementation had the highest average memory usage, with an average of 380 MB when considering all data. The highest observed memory usage was also observed in the GStreamer implementation, where 408.41 MB was used. In second place was the SIPSorcery implementation, with a total average memory usage of 367.4 MB used. With a maximum observed memory usage of 377.07 MB. The best performing implementation was the HTTP Proxy implementation, where the total average memory usage was 350.1 MB, and maximum observed value of 358.18 MB. However, as previously mentioned the HTTP Proxy uses a second process to manage the connection to the relay server which was not included in the measurements.

4.2.5 CPU Usage

The CPU usage was measured five times for each of the implementations. The results of the CPU usage experiment can be seen in 4.4, and the data can be seen in table A.7. Both the WebRTC implementations had higher CPU usage, both during the start-up and streaming phases. Where the on average the GStreamer implementation used 3.6% and 3.28% during start-up and streaming respectively, followed by the SIPSorcery implementation with 3.46% and 3.06%. The HTTP Proxy performed best with on average only 2.14% and 1.82% used during the start-up and stream phases respectively. But as mentioned above, the HTTP Proxy

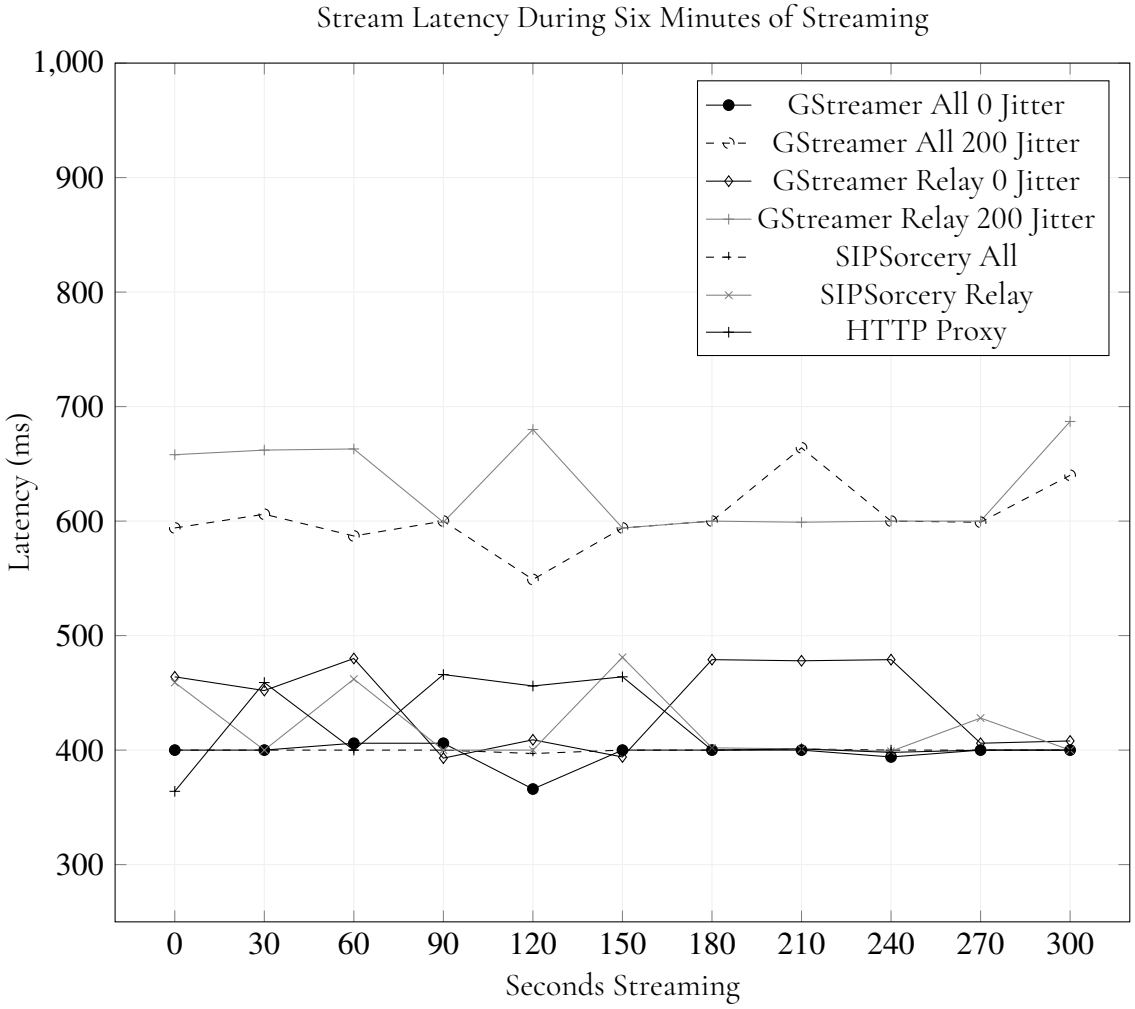


Figure 4.2: Stream Latency

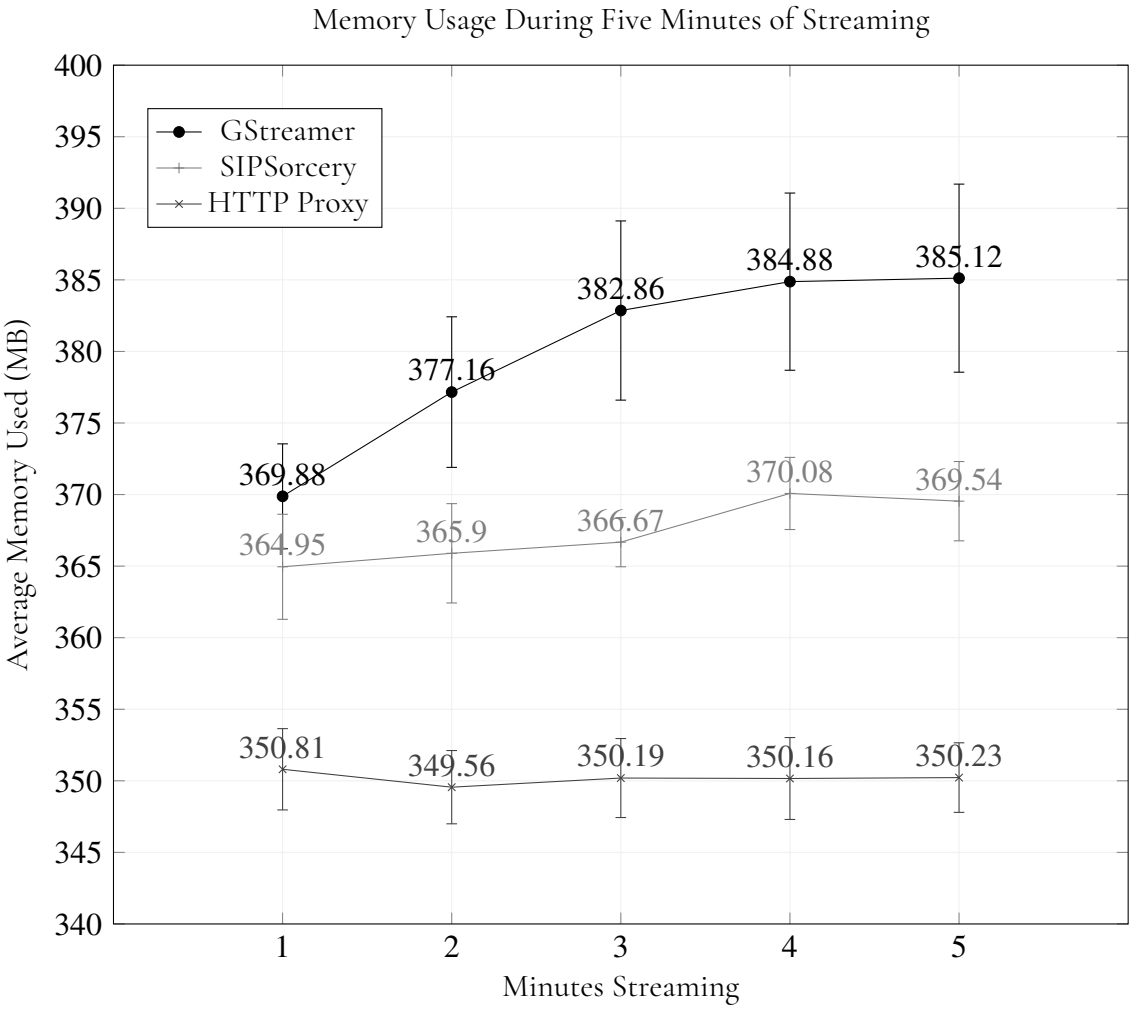


Figure 4.3: The average total memory usage of the implementations at each minute during five minutes of streaming.

uses an additional application in addition to the main application, which was not included in the measurements.

4.2.6 Data

The data for all experiments can be found in Appendix A.

4.3 Discussion

WebRTC is a more complex and has several other features which are not present in the HTTP Proxy, and one should take that into account when observing the data gathered, especially in the case of memory and CPU usage, as the HTTP Proxy is in essence a very lightweight streaming solution.

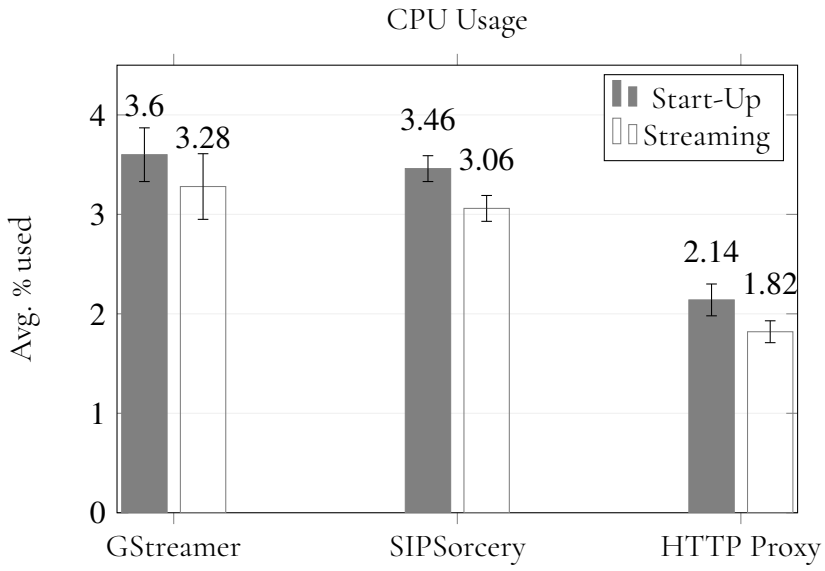


Figure 4.4: CPU Usage during startup and streaming.

4.3.1 WebRTC Implementation Primitives

The measures show that the largest difference between transport policies was the time spent for the ICE Hole punching process, this was true both for the SIPSorcery implementation and the GStreamer implementation. It was especially apparent when viewing the SIPSorcery data, where the time spent for hole punching was almost eleven times the amount of time it took for the all transport policy. This drastic increase was not seen in the GStreamer implementation, where the difference on average only increased by slightly less than four times.

That relay introduces a delay is not surprising, after all, the connectivity checks have to be relayed through a public facing server. However the reason for why the SIPSorcery implementation causes such a significant increase is unknown since both implementation were tested on the same network against the same TURN server. If there would be any efforts made to optimize the time to establish a peer connection for the SIPSorcery implementation the recommendation would be to focus on why the ICE Hole punching process is increased much more compared to the GStreamer implementation. One observation we made was that GStreamer generated many more ICE candidates, which would mean that there are more ICE candidates that need to be tested during the connectivity checks. However, this does not seem to affect the connection setup time negatively. One consideration is that the camera uses the GStreamer framework for WebRTC and interoperability between implementations could be a factor. The standardisation of WebRTC sets out to give different platforms a common specification for real-time communication but as the WebRTC specification develops to incorporate new technologies, adherence to the current specification may vary and this does leave some problems for interoperability.

The offer and answer generation times did not change when changing the transport policy, which is excepted since they are not related to the type of connection that is established and only concerns itself with the session that is established in terms of the type of media that is going to be exchanged and which protocols it will use.

As the time for ICE Hole Punching was so large the GStreamer outperformed the SIPSorcery implementation when using the *relay* transport policy, and thus the time-to-stream was faster when relaying. The SIPSorcery implementation outperformed the GStreamer implementation when using the *all* transport policy, although the consequences were not as great in comparison and did not affect time-to-stream as drastically.

4.3.2 Time to Stream

Both WebRTC implementations, using either of the transport policies was shown to outperform the HTTP Proxy. The GStreamer implementation was shown to be more consistent and established a connection faster in the case of the *relay* transport policy, while in the case of the *all* transport policy the SIPSorcery implementation was slightly faster. The difference in the median time to stream for the *all* transport policy was $1211.7ms - 929ms = 282.7ms$ milliseconds in favour of the SIPSorcery implementation. While the difference in the median time to stream for the *relay* transport policy was much greater, $3717.5ms - 1802ms = 1915.5ms$ milliseconds in favour of the GStreamer implementation.

However, as mentioned above the SIPSorcery implementation using the *relay* transport policy seems to have split itself into two distinct populations, with a significant gap between them. To visualize this, a simple histogram can be seen in figure 4.5

The measurements were taken at different moments, during multiple sessions, which could explain the large gap by the network, or the TURN server being overloaded during one of the sessions, however due to how the data was processed, it is unfortunately not possible in hindsight to trace individual measurements, as the exact date and time were not included.

This problem became apparent after performing the benchmarks, and might have been mitigated by emulating a network and using a private TURN server for testing, instead of using a shared network, as well as TURN server which is in production. Similarly to how it was done by Garcia et al. [15].

As mentioned above the HTTP Proxy is configured in multiple steps, where time to stream includes both the time it takes to configure the proxy and request the stream. This allows users to configure the proxy ahead of time and request the stream later, which is not possible in WebRTC. If one where to look at the act of requesting the stream in isolation however, (see table A.5) the median time for the stream request to resolve is 2694 milliseconds, with lowest observed time of 2579.8 milliseconds. If compared to the WebRTC implementations worst observed case, both the GStreamer and SIPSorcery implementations using the *all* transport policy still perform better with 1844.7 and 2373.1 milliseconds respectively. However when looking at the median time when using the *relay* transport policy, only the GStreamer implementation outperforms the HTTP Proxy, while SIPSorcery implementation is significantly worse.

As such it is important to take into account the amount of traffic that will be using TURN to relay the data in the VMS applications, as the observed time to stream will be depending on the customers network setup, and as such the time to stream metric will vary significantly between local connections, peer-to-peer connections over the public internet and those who use a TURN server to relay the traffic over the public internet.

Some outliers were present in the measurement, both in the WebRTC implementations, as well as the HTTP Proxy. These could have occurred for several reasons, for instance relating to the services that are used in establishing a connection, and the camera which the

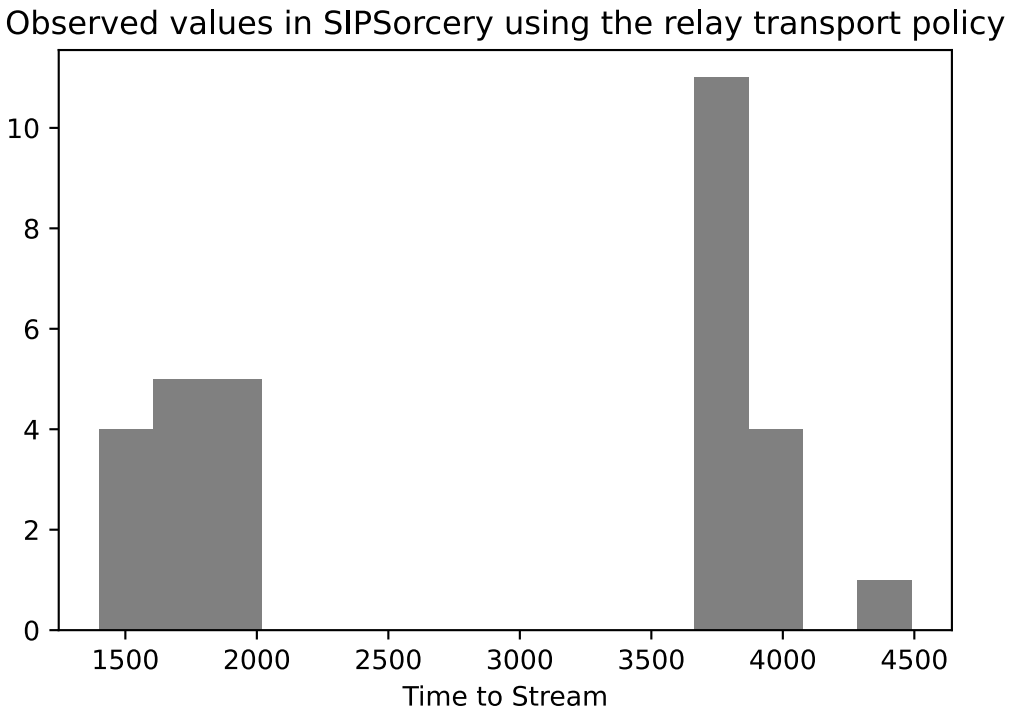


Figure 4.5: Visualization of the distinct split of observed values in the SIPSorcery implementation using the relay transport policy.

WebRTC connection is made to. The services, such as the signaling service and the TURN servers, are known to be slower when subject to a "cold start" where the servers have been idle for some time when the first request is made. It is however not possible to single out these instances, since the service is used not only by the authors but also by customers of Axis.

The camera itself also only supports a maximum of three concurrent WebRTC streams, which would share the already limited resources of both memory and the CPU, some efforts were made to minimize the resources used on the network camera, by reading the log output of the software managing the WebRTC connections, however due to the authors having limited access to the software running the WebRTC client it is still a possibility that resources were used when starting a new connection.

Comparison between Optimized and Default SIPSorcery Connection Establishment

When collecting the data for the SIPSorcery implementation a optimization was made that made the time to stream significantly faster, which entailed generating a DTLS certificate ahead of time of creating the Peer Connection object, however the default behaviour is to create the certificate inside the Peer Connection objects constructor. Depending on which method the time to stream will be affected significantly. The difference can be seen in figure 4.6.

The Time to Stream metrics for the SIPSorcery implementation was significantly affected by the connection establishment optimization. Reducing the overall time to stream by an

average of **1299** ms for the all transport policy, and **796** ms for the relay transport policy. The spread of the total time to stream for the all transport policy was also significantly smaller, **170.43** IQR compared to **812** for the all transport policy.

It is possible this could be improved even further, since there are several algorithms to that could be used to create a self-signed certificate. Using ECDSA (Elliptic Curve Digital Signature Algorithm) is a promising alternative. This is because ECDSA is a lot faster than RSA in general, which is the algorithm that SIPSorcery use for creating certificates if no certificate is provided. Furthermore, the WebRTC specification [27] states under section 4.9 Certificate Management that "If an application does not provide the certificates configuration option when constructing an RTCPeerConnection a new set of certificates MUST be generated by the user agent. That set MUST include an ECDSA certificate with a private key on the P-256 curve and a signature with a SHA-256 hash". Thus SIPSorcery does not comply with the specification on this and if developers do not take this into account one can expect the connection establishment time to be delayed because of the extra time of generating an RSA key for the certificate [25].

4.3.3 Latency

The latency was measured for different transport policies, and in the case of the GStreamer implementation, also different jitter buffer duration. All implementations had a stable latency during the entire five minutes that it was measured. The lowest (best) latency was measured for the GStreamer implementation using the all transport-policy with a jitter buffer duration of 0 milliseconds, with an average latency of 397.45 milliseconds. While the highest (worst) latency was measured using the GStreamer implementation using the relay transport policy using a jitter buffer with 200 milliseconds of extra latency.

The latency measurements split itself into two distinct groups, the GStreamer implementation using a jitter buffer with an extra 200 milliseconds of latency with both transport policies, and the rest of the measurements. This is not surprising since when using a jitter buffer latency is traded off for better stream quality.

Both WebRTC implementations using the relay transport policy showed a reduction in stream quality, where the image was sometimes rendered out of order and had significant video artifacts. As such the results of the latency measurements had to be taken at times around the thirty second intervals where the image was clear and both timestamps could be read clearly.

The process of collecting the latency was very manual and tedious, a better alternative which would automate the process but requires additional work and has been implemented in previous end-to-end latency tests for WebRTC, would be to use Optical Character Recognition (OCR) using the same method. Using OCR would result in more measurements per experiment and without the need for manual recording and playback of the stream. This solution does require synchronized clocks between connected peers.

4.3.4 Memory Usage

The memory usage showed only a small difference between the different streaming solutions if one where to consider the total memory used as relative to each other. The highest average

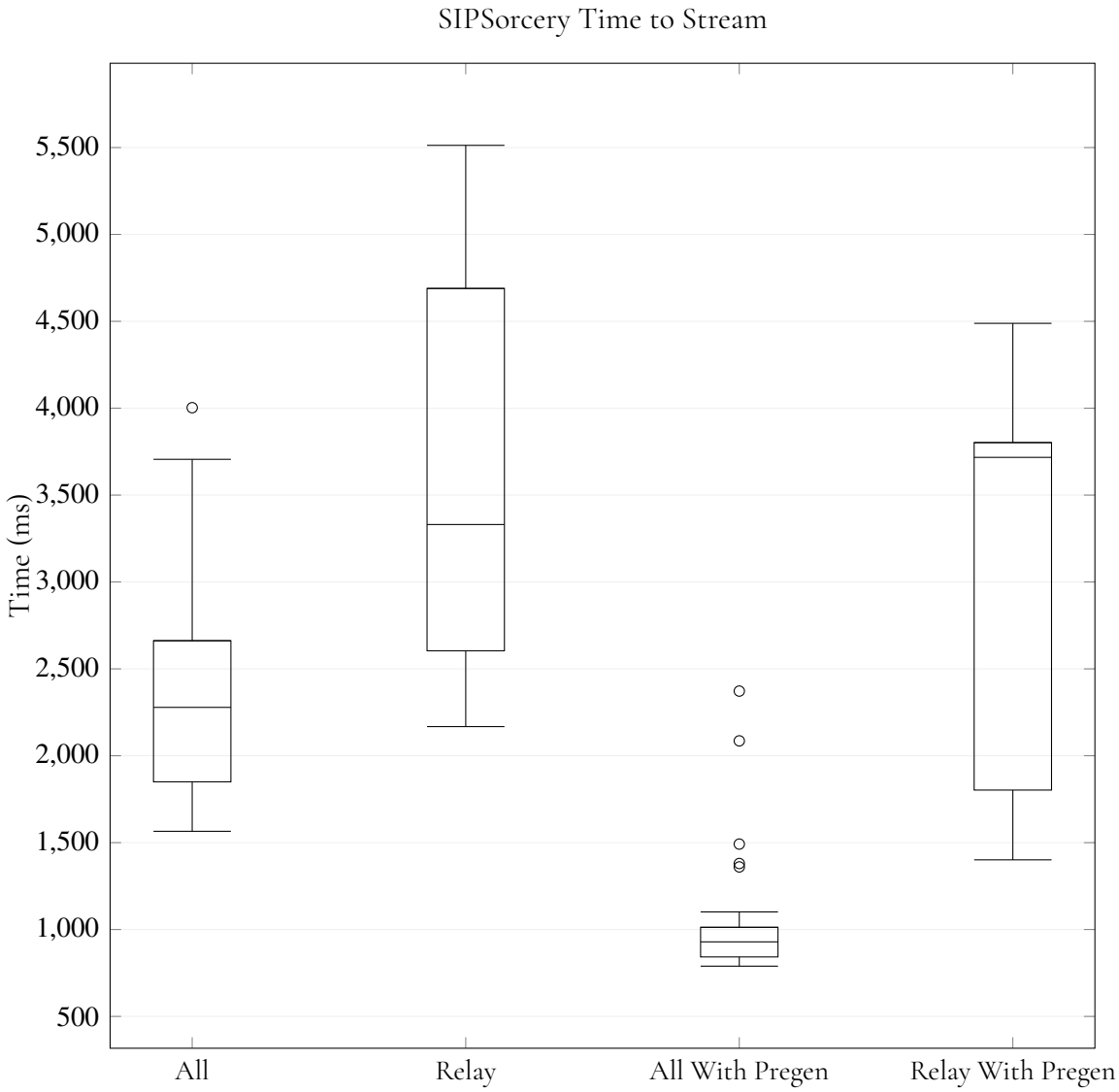


Figure 4.6: Difference in Time to Stream when generating the certificate ahead of time.

memory usage was observed in the GStreamer implementation, but the SIPSorcery implementation used 96.7% of the memory of the GStreamer implementation, while the HTTP Proxy, not including the additional process that is required for it to function, used on average 92.4% of the average memory of the GStreamer implementation. This can probably be attributed to that many components, such as the Axis decoding and rendering pipeline is used by all of the implementations.

When performing the experiments the GStreamer implementation had one measurement which impacted the averages significantly, where the memory usage jumped from using 377.8 MB in the first minute of the stream, to using 408.4 MB at the fifth minute. The other measurements however saw a somewhat consistent grouping, where the memory usage expectantly fluctuated during the stream, since memory is acquired when needed, and released whenever the garbage collector has deemed it as no longer in use. This also caused the spread in the data to be significantly larger than if the measurement would be ignored. What caused

this significant jump in memory usage is unknown, a theory could be that the garbage collector deemed that the memory was still in use and did not release it, however this is hard to both prove and disprove as it would not be repeatable due to the unpredictable nature of the garbage collector.

Using the data gathered we can observe that WebRTC applications have an extra cost compared to the very lightweight HTTP Proxy solution. If one where to invest no extra time in making any additional optimizations, i.e. establishing one peer connection for each stream that is shown, as it is done in the proof-of-concept WebRTC implementations, one could naively draw the conclusion that WebRTC implementations would use about 15 - 35 extra MB (depending on the implementation) of extra memory per stream compared to the HTTP Proxy. However the signaling service component could be reused in subsequent streams, which would in practice mean that the next stream would not increase the memory usage as much as the first. It is therefore difficult to give an exact number, since one would need to investigate the cost of each of the components which are present in the WebRTC implementations, but an increased cost would be expected nonetheless.

4.3.5 CPU Usage

All streaming solutions did not show a large stress on the CPU, this can be mainly due to that Axis decoding and rendering solution uses the GPU to accelerate the process.

The HTTP Proxy was found to be the clear winner where, the WebRTC implementations used almost twice the amount of the CPU on average. As mentioned, the HTTP Proxy is simple and lightweight, and the CPU Usage certainly echoes that statement. The WebRTC implementations require more several moving parts in one application to function, but at the same time WebRTC offers more features, but it comes with the cost of extra CPU usage. However since the auxiliary process for the HTTP Proxy was not profiled and measured, the exact cost is difficult to estimate.

4.4 Features

To answer research question 2, the application used the existing Axis packages to provide a minimum viable example of the features.

4.4.1 Client Side Dewarping and Digital PTZ

Client Side Dewarping, as the name suggests, is the process of taking a warped image, and render a dewarped that is easier for the user to digest. Digital PTZ emulates the physical PTZ capabilities of cameras by digitally zooming, panning, and tilting the image without the need to interact with actual physical hardware. The feature was implemented, but not in the same scope as the solution that is present in the current VMS solutions, but sufficient as a proof-of concept. An image demonstrating the effects of digital PTZ can be seen in 4.7. More advanced client side dewarping techniques could be applied, e.g. dewarping a fish-eye lens to a panoramic view, however as of writing the Axis' Rendering and decoding solution does not have the capabilities for these transformations.

4.4.2 Audio playback and transmission

Audio playback and transmission is one of the staples of WebRTC, however for it to function, i.e. actually playing back the audio, the encoded data needs to be depayloaded and decoded. Currently the network cameras WebRTC implementation only supports the OPUS codec for audio. [4] This posed to be a problem, since Axis' proprietary decoding and rendering solution does not support it at the time of writing.

However the current WebRTC implementation on the network cameras is implemented using GStreamer, which has support for sending encoded audio with a variety of different encodings which are supported in the decoding and rendering pipeline. One of these encoding is the common G.711 encoding, which is a mandatory to support in browser implementations of WebRTC. Using this encoding the GStreamer has support for depacketization from RTP packets, the SIPSorcery implementation would however require the implementation of the depacketizer as it is not currently included in the library.

In the case of transmission any of the available codecs that is supported by GStreamer could be used, the same goes for GStreamer on the client side. For SIPSorcery encoding and packetization would have to be implemented to successfully transmit audio over WebRTC.

4.4.3 Scrubbing

Scrubbing entails that during a playback session the user can pause, play, and move back and forward while watching the video. However as WebRTC is primarily used as a live streaming solution, this feature would require an extension which could support this feature. One of the features of WebRTC which as gone largely unmentioned in this thesis is the possibility of sending arbitrary data over data channels. This could unlock several features, including scrubbing, while still enjoying a fully peer-to-peer connection. A proposed solution would be that during a playback session the user can, using the data channels, send a request to change the current streamed video to certain point in time, for which the network camera can change the currently streamed video by sending the closest I-frame before the time requested and the additional P and B-frames which are required to reach the requested time. The client then needs to collect the frames required to reach their requested time. Using this proposed solution, the change in content is achieved without the need for renegotiation of the stream.

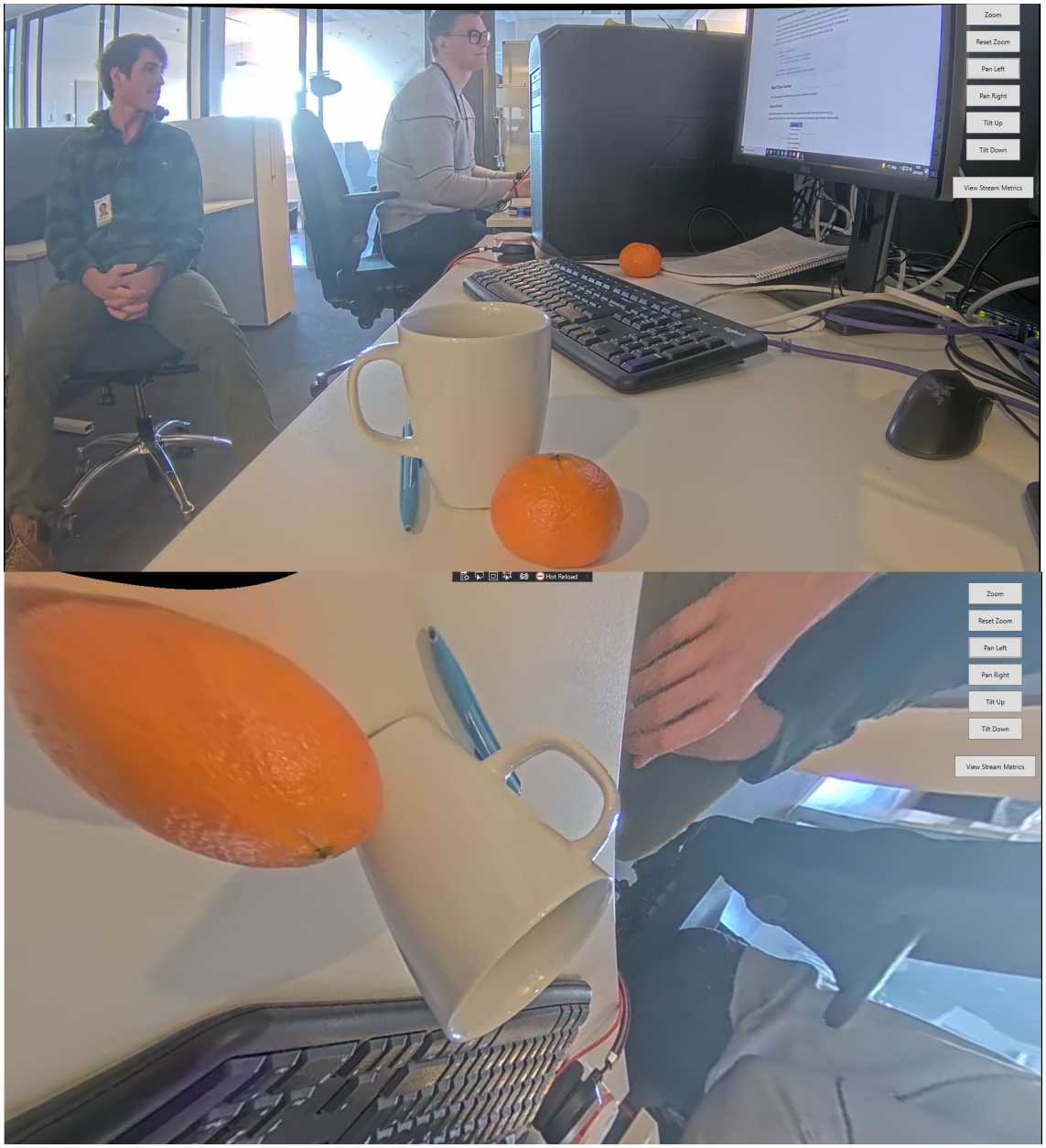


Figure 4.7: Example of digital PTZ being applied to a live video stream.

Chapter 5
Conclusions

The benchmarks performed on the WebRTC implementations, using one of Axis' current streaming solutions as a reference showed that WebRTC can perform as good in the time to stream metric under certain circumstances, near equivalent during the latency test, and worse in the measurements concerning the CPU and memory usage measurements. The use of the relay transport policy showed that using TURN servers to relay traffic had a strong impact on the time to stream metric, and gave varied results which impacted the QoE negatively.

The features were achieved by successfully integrating WebRTC with current decoding and rendering solutions developed at Axis, while WebRTC is still in the development stage at Axis, where the software that run on the devices and infrastructure is still being developed. As such more work has to be performed to get all the features working, which mostly includes support for the protocols that are specified in the WebRTC specification.

As such we answer the research questions with the following:

When integrating WebRTC as a streaming solution that interacts with IoT devices made by Axis, how does it compare with the current streaming solutions at Axis in terms of various quality and reliability metrics?

When integrating WebRTC as a streaming solution that interact with the Axis current environment and devices the quality and reliability metrics varied with the use of TURN servers to relay the traffic, which led to decreased performance and quality of experience, especially in the time to stream metric. Latency from the data gathered seemed to perform better or equivalent with some expected additional latency when using TURN servers to relay the data. CPU and memory usage were all worse when compared to the HTTP Proxy while running the experiments with a Axis M1135 Network Camera.

As for research question 2:

Can the WebRTC solution(s) support the same features that exist in the current streaming solution at Axis? That is the following:

- **(a) Client Side Dewarping and Digital Pan-Tilt-Zoom (PTZ)**
- **(b) Audio Playback and Transmission**

- **(c) Scrubbing**

Some of the same features are supported, but some requires additional work.

- (a) Has support through the Axis decoding and rendering solution.
- (b) Requires additional support for the OPUS codec in the decoding and rendering solution, or a change from the current codec on the network camera to one which is currently supported in the rendering and decoding pipeline, such as G.711.
- (c) Requires additional work, since controlled playback is not the primary use case of WebRTC, but the features present in WebRTC could hypothetically be leveraged to achieve features such as scrubbing.

5.1 Reflections on the project

The method used by WebRTCBench was very helpful when evaluating the internals of the WebRTC implementations, as it could identify which of the steps of the connection establishment process was acting as bottlenecks which would effect QoE metrics such as time-to-stream. During the experiments the observation could be made early that the initialization of the peer connection object in the SIPSorcery library was a clear bottleneck and as such the authors could investigate further, which resulted in the optimization where the DTLS certificate could be generated ahead of time which resulted in better time-to-stream measurements.

The dotMemory profiling tool was used in a similar manner. When performing the memory usage experiments a severe memory leak could be found in the GStreamer implementation which could be fixed. dotTrace could be used in the common WebRTC components to find the most expensive function which could decrease the over-all read loop for converting the H.264 frames into the MediaFrames which are then passed to the decoding and rendering solution.

As the WebRTC application that is used on the network camera used in experiments is still in development it did not allow us perform all the experiments that we would have wished to perform. For example we could not reach steady state performance in the client application since the camera allows only three concurrent video session to be active at any given moment.

Even though it is out of scope for the thesis, the authors feel that it is important to mention that when using both WebRTC implementations with the relay transport policy, several video artifacts have been present in the video stream, as well as the streaming freezing occasionally. The authors suspect that this is due to dropped packets which is likely because of network congestion. Upon further investigation, neither GStreamer's webrtcbin nor SIPsorcery's PeerConnection seems to implement congestion control which could be the explanation to why the stream sporadically freezes.

Performing all benchmarks on a deployed black-box infrastructure has both upsides and downsides. The downside includes the previously mentioned problem of not knowing the exact reason for the video artifacts and freezes, while also being unable to test for the different network configurations that the system could be exposed to. While the upside is that of a realistic view of the performance when using the currently deployed infrastructure.

5.2 Authors Recommendations

The results showed that WebRTC in both evaluated implementations could compete with the currently used peer-to-peer streaming solution on metrics such as latency, memory usage, CPU usage. While outperforming the currently used streaming solution on the time-to-stream metrics.

Both WeRTC implementations performed similarly and could be used as a suitable alternative to the currently used streaming solutions, however only after some additional work to support the current features in their VMS, such as audio and scrubbing which currently are not supported. Furthermore, congestion control needs to be added as well as proper certificate generation for SIPSorcery.

The two open-source solutions which were evaluated both performed similarly. If all features are essential, such as IPv6 and TCP/TLS support proves to be essential, the GStreamer implementation is the only one which supports them as of writing, however GStreamer comes with it own drawbacks. The most notable is that GStreamer is not a native C# library, and additional software has to be installed for the library to be used, in contrast the SIPSorcery library is simple to integrate into an existing application.

GStreamer is a powerful media processing framework but with power comes complexity and likely implies higher cost in terms of development time. GStreamer does have a (much) larger community and a developed ecosystem which is meritable considering how the WebRTC specification is likely to have continuous revisions and updates. As the cameras are using GStreamer's WebRTC implementation, it may prove useful to use GStreamer for the client as endpoints would share the same framework for WebRTC. This would allow for easier exchange of knowledge within Axis and a code-base that can easier be collectively understood between teams. Furthermore, interoperability between WebRTC implementations is not guaranteed, especially if (or when) the specification is updated, and using the same framework would eliminate this issue.

5.3 Future Work

The measurements were taken on a single M1135 Axis Network Camera, due to limitations of the current firmware needed to run WebRTC on the network camera, the authors could not perform many tests on a device which has less powerful hardware, this may impact the performance of metrics such as time to stream and latency, since many operations, e.g. encoding video, can be quite intensive. As such we recommend performing the same experiments on cheaper hardware to ensure that the performance meets the requirements for all of Axis devices.

As this technology grows within Axis and more devices and servers needs to be connected through WebRTC, it will be important to monitor how the system would behave before pushing into production. For this purpose it would be beneficial to emulate network configurations and conditions, as done by García et al. and proposed by Gouaillard et al.[16] and include this in a testing framework. Furthermore, work has been performed in *Analysis of video quality and end-to-end latency in WebRTC* by García et al. [14] that gives an approach for automating testing for video quality and end-to-end latency which could be included in such a testing framework to automate testing in both emulated networks and real networks.

In this work only one camera is displayed at a given time and the client is connected to the camera through pure peer-to-peer except when a TURN server is relaying. This setup would make scalability on the client difficult as the client would have a hard time processing many concurrent peer-to-peer connections. In future work, it would be interesting to look at implementing demuxing at the client side to allow for a media server to multiplex the stream of many cameras and send the media in a single stream. This could increase bandwidth efficiency, while also decreasing the amount of ports that has to be used by the device running the client. Furthermore, the client would not have to decode multiple streams which could reduce the computational efforts for the device running the client.

The performance when using TURN to relay the peer-to-peer traffic was significantly worse when compared to when the traffic was sent directly between the peers. As for latency, this is expected as the traffic is relayed through a server located abroad while when using STUN the traffic runs on the local network. However, artifacts and freezes should be avoidable when using relay and an investigation to why the performance deteriorates while relaying the traffic would allow for a great impact of the usability of WebRTC at Axis. As discussed, we suspect that congestion control will improve these qualities but further investigation could be done on metrics such as the bandwidth, CPU, and memory requirements that is required for the TURN server to support the requirements on QoE that is expected by the users of Axis VMS.

Chapter 6
Appendix

References

[1] dotMemory: a Memory profiler & Unit-Testing Framework for .NET by JetBrains. https://www.jetbrains.com/dotmemory/.

[2] dotTrace: .NET Performance Profiler by JetBrains. https://www.jetbrains.com/profiler/.

[3] Open Broadcaster Software | OBS. https://obsproject.com/.

[4] Opus Interactive Audio Codec. https://opus-codec.org/.

[5] An Offer/Answer Model with the Session Description Protocol (SDP). Rfc, June 2002. https://datatracker.ietf.org/doc/html/rfc3264.

[6] SIP: Session Initiation Protocol. Rfc, June 2002. https://datatracker.ietf.org/doc/html/rfc3261.

[7] SDP: Session Description Protocol. Rfc, July 2006. https://datatracker.ietf.org/doc/html/rfc4566.

[8] Bitmovin Video Developer Report 2018, September 2019. https://go.bitmovin.com/hubfs/Bitmovin-Video-Developer-Report-2018.pdf.

[9] F. Audet, Ed. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP . Rfc, January 2007. https://datatracker.ietf.org/doc/html/rfc4787#section-4.1.

[10] A. Amirante, T. Castaldi, L. Miniero, and S. P. Romano. Jattack: a webrtc load testing tool. In *2016 Principles, Systems and Applications of IP Telecommunications (IPTComm)*, pages 1–6, 2016.

[11] Taherri Sajjad et al. WebRTCBench: A Benchmark for Performance Assessment of WebRTC Implementations. Institute of Electrical and Electronics Engineers (IEEE), 2015.

[12] Boni Garcia, Francisco Gortazar, Luis Lopez-Fernandez, Micael Gallego, and Miguel Paris. Webrtc testing: Challenges and practical solutions. *IEEE Communications Standards Magazine*, 1(2):36–42, 2017.

[13] Boni Garcia, Luis Lopez-Fernandez, Micael Gallego, and Francisco Gortazar. Kurento: The swiss army knife of webrtc media servers. *IEEE Communications Standards Magazine*, 1(2):44–51, 2017.

[14] Boni Garcia, Luis Lopez-Fernandez, Francisco Gortazar, and Micael Gallego. Analysis of video quality and end-to-end latency in webrtc. In *2016 IEEE Globecom Workshops (GC Wkshps)*, pages 1–6, 2016.

[15] Boni García, Luis López-Fernández, Micael Gallego, and Francisco Gortázar. Testing framework for webrtc services. 01 2016.

[16] Alexandre Gouaillard and Ludovic Roux. Real-time communication testing evolution with webrtc 1.0. In *2017 Principles, Systems and Applications of IP Telecommunications (IPTComm)*, pages 1–8, 2017.

[17] Hari Kalva. The H.264 Video Encoding Standard. *IEE Multimedia*, 13(4):86–90, 2006.

[18] Ian Hickson. WebRTC 1.0: Real-time Communication Between Browsers, October 2011. `https://www.w3.org/TR/2011/WD-webrtc-20111027/`.

[19] IETF. Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol . Rfc, November 2017. `https://tools.ietf.org/id/draft-ietf-ice-trickle-15.html`.

[20] Anna Maria Mandalari, Miguel Angel Diaz Bautista, Francisco Valera, and Marcelo Bagnulo. Natwatcher: Profiling nats in the wild. *IEEE Communications Magazine*, 55(3):178–185, 2017.

[21] Microsoft Corporation. Host WinRT XAML controls in desktop apps (XAML Islands). `https://docs.microsoft.com/en-us/windows/apps/desktop/modernize/xaml-islands`.

[22] Ryan Poonolly. Pigeon RTC: Setup video chats using carrier pigeons! `https://cpoonolly.com/pigeon-rtc/`.

[23] Stephen M. Blackburn et al. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Communications of the ACM*, 2008.

[24] The Matroska Organization. What is Matroska? `https://www.matroska.org/what_is_matroska.html`.

[25] Dhanashree Toradmalle, Rohan Singh, Het Shastri, Nikita Naik, and Vishal Panchidi. Prominence of ecdsa over rsa digital signature algorithm. In *2018 2nd International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC), 2018 2nd International Conference on*, pages 253–257, 2018.

[26] Jozsef Wazz. How Discord Handles Two and Half Million Concurrent Voice Users using WebRTC, September 2018. https://blog.discord.com/how-discord-handles-two-and-half-million-concurrent-voice-users-using-webrtc

[27] World Wide Web Consortium. WebRTC 1.0 API Documentation. https://www.w3.org/TR/webrtc/.

[28] World Wide Web Consortium, Internet Engineering Task Force. Web Real-Time Communications (WebRTC) transforms the communications landscape; becomes a World Wide Web Consortium (W3C) Recommendation and multiple Internet Engineering Task Force (IETF) standards, January 2021. https://www.w3.org/2021/01/pressrelease-webrtc-rec.html.en.

Appendices

Appendix A
Data

Individual experiments for SIPSorcery memory usage, MB					
Experiment	Minutes Streaming				
	1	2	3	4	5
1	371.15	374.12	366.83	373.86	375.86
2	360.92	362.74	366.49	367.14	363.23
3	366.79	367.74	364.67	369.97	368.84
4	352.87	354.2	362.54	362.65	363.85
5	373.04	370.68	372.84	377.07	375.91

Table A.1: Data for the Memory usage during five minutes of streaming in the SIPSorcery implementation.

Individual experiments for GStreamer memory usage, MB					
Experiment	Minutes Streaming				
	1	2	3	4	5
1	377.83	394.12	405.09	407.9	408.41
2	361.24	368.52	383.96	384.76	388.5
3	362.19	366.71	371.51	374.29	370.73
4	378.35	384.54	383.52	383.64	382.76
5	369.78	371.91	370.21	373.79	375.2

Table A.2: Data for the Memory usage during five minutes of streaming in the GStreamer implementation.

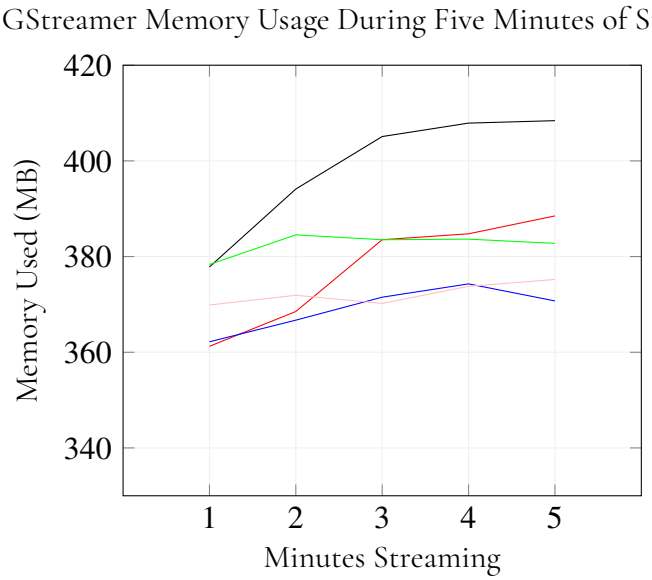


Figure A.1: Individual Experiments on the Memory usage of the GStreamer implementation.

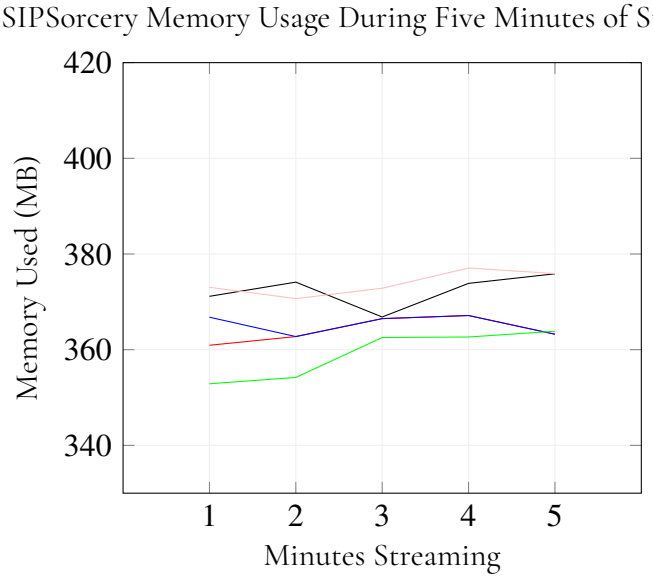


Figure A.2: Individual Experiments on the Memory usage of the SIP-Sorcery implementation.

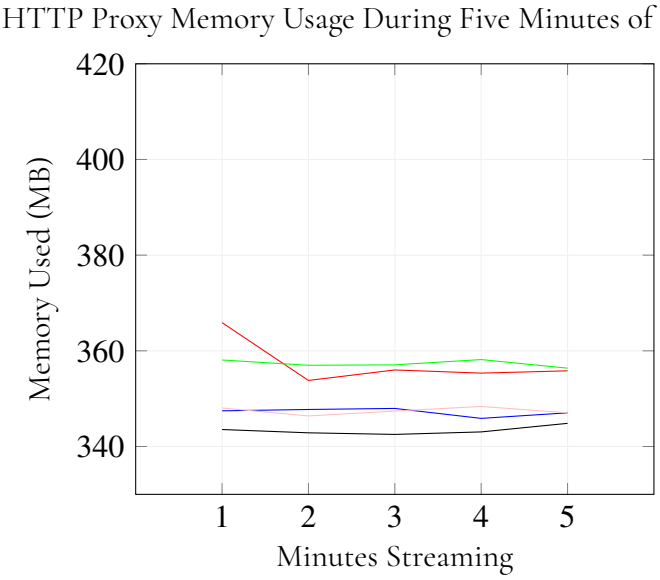


Figure A.3: Individual Experiments on the Memory usage of the HTTP Proxy implementation.

Individual experiments for HTTP Proxy memory usage, MB					
Experiment	Minutes Streaming				
	1	2	3	4	5
1	343.55	342.86	342.53	343.05	344.84
2	356.89	353.81	356	355.34	355.82
3	347.46	347.74	347.96	345.88	347.04
4	358.08	356.98	357.07	358.17	356.38
5	348.05	346.39	347.41	348.37	347.05

Table A.3: Data for the Memory usage during five minutes of streaming in the HTTP Proxy implementation.

Sorted Raw Time to Stream Data in milliseconds				
GStreamer All	**SIPSorcery All**	**GStreamer Relay**	**SIPSorcery Relay**	**HTTP Proxy**
898.6	789.2	1552.9	1401	5081.6
946.3	821.4	1553	1560.7	5086.7
977.3	823.4	1581.7	1562.2	5088.4
1022.3	829.2	1582.6	1594.1	5091.1
1055.1	832	1597.6	1691.6	5102.3
1095	840.9	1620.9	1700.3	5117.5
1139.9	842	1673.6	1723.5	5125.1
1158.1	842.4	1702	1800.3	5137.7
1158.4	842.4	1712.4	1809.2	5143.7
1165.3	844.8	1739.9	1840.7	5164
1172.7	850.6	1744.3	1858.7	5177.6
1174.8	885.9	1777.6	1886.4	5182.6
1182.4	917.1	1784.4	1940.3	5184.9
1198.6	920	1798.7	1946	5190
1206.7	928.9	1800	3717.1	5190.9
1216.7	929.1	1804	3717.2	5197
1218.8	946	1815	3717.7	5203.8
1235.6	966.5	1825	3726.9	5222.8
1236.1	966.6	1837.9	3727.9	5227.9
1290.3	976	1841	3753.9	5239
1298	990.4	1860.2	3758.7	5248.2
1338.2	1002.1	1880.1	3782.3	5256.5
1370.5	1016.4	1915.8	3808.9	5329.6
1375.9	1046.8	1921.4	3809.6	5354.8
1377.7	1101.4	2066.6	3830.1	5366
1429.9	1360.3	2074.6	3872.4	5373.7
1561.1	1379.9	2164.1	3876.6	5420.9
1572.3	1491.7	2386.4	3907.5	5597.1
1608.7	2085.7	2470.7	3927	5698.9
1844.7	2373.1	2797.7	4488.5	7199.9

Table A.4: Time To Stream data in milliseconds for all streaming alternatives and different transport policies.

Time for the steps in the Time to Stream metric for the HTTP Proxy		
Session Establishment	**Stream Request**	**Time To Stream**
2505.6	3191.3	5698.9
2428.5	2773.3	5203.8
2645.7	2706.1	5354.8
2514.8	2730.4	5248.2
2476.3	2718.7	5197
2513.2	2626.5	5143.7
2515.5	2620.1	5137.7
2502.4	2584	5088.4
2479.6	2939.3	5420.9
2476.5	2639	5117.5
2508.7	3082.4	5597.1
2471	2901.7	5373.7
2491	2671	5164
2532.9	2646	5184.9
2524.8	2837.2	5366
2506.3	2579.8	5091.1
2539.1	2585	5125.1
2575.6	4622.3	7199.9
2515.5	2740	5256.5
2544.7	2781.9	5329.6
2486.9	2597.8	5086.7
2478.9	2621.4	5102.3
2473.6	2715.3	5190.9
2529	2648.6	5182.6
2560.3	2662.6	5227.9
2446.2	2633.4	5081.6
2535.9	2681.9	5222.8
2471.5	2714.5	5190
2611.3	2624.7	5239
2465.4	2710.2	5177.6

Table A.5: The time for the two steps in the time to stream metric for the HTTP Proxy. Note that there is code run between the end of the session establishment and the beginning of the stream request, which results in the time to stream to be slightly larger than the sum of the two.

Stream Latency during 300 seconds of streaming in milliseconds												
Implementation	0 s	30 s	60 s	90 s	120 s	150 s	180 s	210 s	240 s	270 s	300 s	**Mean**
SIPSorcery All	400	400	400	400	397	400	400	401	400	400	400	399.8
SIPSorcery Relay	459	400	462	400	400	481	402	401	399	428	400	421.1
GStreamer All 0 Jitter	400	400	406	406	366	400	400	400	394	400	400	397.5
GStreamer All 200 Jitter	594	606	587	600	549	594	600	664	600	599	640	603.0
GStreamer Relay 0 Jitter	464	452	480	393	409	394	479	478	479	406	408	440.2
GStreamer Relay 200 Jitter	658	662	663	599	680	680	594	600	599	600	600	631.1
HTTP Proxy	364	459	400	466	456	464	400	401	398	400	400	418.9

Table A.6: The stream latency during 300 seconds of streaming.

CPU Usage Data in Pairs, % CPU used.					
GStreamer		SIPSorcery		HTTP Proxy	
Start-Up	Streaming	Start-Up	Streaming	Start-Up	Streaming
3.6	3.5	3.6	3.3	1.8	1.6
3.1	2.2	3.6	3.2	2.4	2
2.9	2.9	3.8	3.3	1.7	1.5
4.1	3.7	3.1	2.7	2.4	2
4.3	4.1	3.2	2.8	2.4	2

Table A.7: The data for the Start-Up and Streaming CPU usage in the streaming implementations. In the pairs they were measured.

Appendix B
Examples and Information

List of Native WebRTC Implementations		
Name	**Language**	**Source**
WebRTC native APIs	C++	`https://webrtc.github.io/webrtc-org/native-code/native-apis/`
Pion	Go	`https://pion.ly`, `https://github.com/pion/webrtc/`
aiortc	Python	`https://github.com/aiortc/aiortc`
RawRTC	C++	`https://github.com/rawrtc/rawrtc`
GStreamer	C	`https://gstreamer.freedesktop.org/documentation/webrtc/index.html`
webrtc-rs	Rust	`https://github.com/webrtc-rs/webrtc`
SIPSorcery	C#	`https://github.com/sipsorcery-org/sipsorcery`
MixedReality-WebRTC	C#	`https://github.com/microsoft/MixedReality-WebRTC`

Table B.1: Native WebRTC Libraries

```
v=0
o=- 4385423089851900022 0 IN IP4 0.0.0.0
s=-
t=0 0
a=ice-options:trickle
a=group:BUNDLE video0 application1
m=video 9 UDP/TLS/RTP/SAVPF 96
c=IN IP4 0.0.0.0
a=setup:actpass
a=ice-ufrag:lsJx+7d6hsCyL8K6m8/KbgcqMqizaZqy
a=ice-pwd:zFUTJmx6hNnr/JRAq2b3wOtmm88XERb3
a=rtcp-mux
a=rtcp-rsize
a=sendrecv
a=rtpmap:96 H264/90000
a=rtcp-fb:96 nack pli
a=framerate:30
a=fmtp:96 packetization-mode=1;profile-level-id=42E01F;sprop-parameter-sets=Z00AKeKQDwBE/LNwEBAaUABt3QAZv8wA8SIq,aO48gA==
a=ssrc:3776670536 msid:user3344942761@host-c94b5db webrtctransceiver11
a=ssrc:3776670536 cname:user3344942761@host-c94b5db
a=mid:video0
a=fingerprint:sha-256 AE:1C:59:19:00:7B:C2:1C:85:95:0C:6C:8C:14:E8:67:A4:7D:D0:AE:90:5D:8F:BB:D7:5B:95:49:03:6E:94:8F
m=application 0 UDP/DTLS/SCTP webrtc-datachannel
c=IN IP4 0.0.0.0
a=setup:actpass
a=ice-ufrag:lsJx+7d6hsCyL8K6m8/KbgcqMqizaZqy
a=ice-pwd:zFUTJmx6hNnr/JRAq2b3wOtmm88XERb3
a=bundle-only
a=mid:application1
a=sctp-port:5000
a=fingerprint:sha-256 AE:1C:59:19:00:7B:C2:1C:85:95:0C:6C:8C:14:E8:67:A4:7D:D0:AE:90:5D:8F:BB:D7:5B:95:49:03:6E:94:8F
```

Figure B.1: An example WebRTC SDP Offer Message

Client Hardware specification:

- CPU
  - Manufacturer: Intel
  - Architecture: 64 Bit
  - Cores: 8
  - Clock rate: 3,6 GHz
- Memory
  - Size: 16 GB
  - Clock rate: 2666 MHz
- Graphics Card:
  - Manufacturer: Nvidia
  - Name: GTX 1060
  - Memory: 3GB DDR5
  - Base clock rate: 1,5 GHz

Figure B.2: Hardware used in the benchmarks running the receiving client.

EXAMENSARBETE Integration and Evaluation of WebRTC in an Exisiting .NET Environment
STUDENTER Simon Tenggren, Martin Gottlander
HANDLEDARE Jörn W Janneck (LTH), Tore Paulsson (Axis Communications AB)
EXAMINATOR Per Andersson (LTH)

Utvärdering av WebRTC i moderna videohanteringssystem

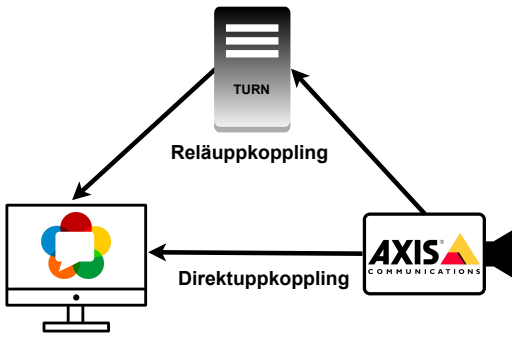
POPULÄRVETENSKAPLIG SAMMANFATTNING **Simon Tenggren, Martin Gottlander**

WebRTC utvärderas mot samtida videoströmnings-lösningar i Axis .NET miljö för att undersöka ifall de kan mäta sig mot en av Axis nuvarande lösningar, samt ifall WebRTC kan användas med de funktionerna som slutanvändarna förväntar sig.

Att hitta den optimala uppkopplingen mellan två parter som kan sitta på två väldigt olika och restriktiva nätverk har traditionellt varit ett knepigt problem, men med introduktionen av WebRTC kan dessa uppkopplingar hittas på ett dynamiskt, säkert, och standardiserat sätt! Vilket är av intresse för både utvecklare och slutanvändare.

I detta examensarbete har WebRTC utforskats som ett alternativ till en av Axis Communications videoströmnings-lösningar i deras videohanteringssystem. Genom att integrera videoströmning över WebRTC från Axis nätverskameror kunde mätvärden som har en stor påverkan på användarupplevelsen, så som fördröjning och uppkopplingshastighet. Samt värden som mäter hur mycket extra resurser som krävs vid användning av WebRTC. All mätning gjordes i den nuvarande miljön vilket gav en realistisk inblick i hur upplevelsen skulle vara hos slutkunden av videohanteringssystemen!

Två implementationer av WebRTC utvärderades. Båda under omständigheter som påverkar vilken sorts av uppkoppling som skapas mot nätverkskameran. Antingen en reläuppkoppling där en server agerar som en brevbärare mellan kameran och datorn, eller en direktuppkoppling där uppkoppling sker direkt mellan nätverkskameran och applikationen utan en tredje part. (Se bild) Resultatet visade att WebRTC kan tävla



mot Axis samtida lösning när det kom till fördröjning i videoströmmen, och slå Axis lösning i uppkopplingshastighet. Dock varierade uppkopplingshastigheten vid reläuppkopplingar markant, vilket resulterar i en skiftande upplevelse. Jämfört med Axis nuvarande lösning var WebRTC dock mer resurskrävande. De funktioner som förväntas finnas i Axis videohanteringssystem kunde användas direkt efter integrationen av WebRTC, eller kräver minimala ändringar, t.ex. i form av byte av protokoll, för att exempelvis kunna spela upp ljud som skickas över WebRTC uppkopplingar.