Evaluating Voice over IP phone implementation on a freescale Cortex A9 processor running Linux using open source SIP and WebRTC

Eric Sjögren

August 16, 2016
Master’s Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Thomas Hellström
Examiner: Henrik Björklund

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN
Abstract

Voice over IP (VoIP) is a methodology that refers to the delivery of multimedia and voice sessions over an Internet connection and it provides an alternative to regular voice calls using phone lines, usually referred to as the Public Switched Telephone Network (PSTN). Web Real-Time Communication (WebRTC) is an API definition for browser-to-browser VoIP applications; the definition acts as a foundation for applications using voice, video, chat, and P2P file sharing in a browser environment without the need of either internal or external plugins. To allow WebRTC to make calls to non-WebRTC VoIP applications, an initiation protocol (which is not included in the WebRTC implementation looked at here, i.e., the one released by Google) is needed. One such protocol is the Session Initiation Protocol (SIP), which is the standard protocol used for initialising, changing and terminating interactive sessions for multimedia today; it is particularly known for its use in VoIP applications.

In this thesis, we evaluate the possibility of the creation of a WebRTC implementation using SIP (this type of implementation is referred to as WebRTC-SIP) that runs on an ARM A9 processor architecture. The evaluation is split into two steps. The first step consists of analysing and performing tests of the Linux audio drivers on an ARM platform. The tests are used to determine how a WebRTC-SIP application could affect the audio drivers on such a platform. The second step involves implementation of a WebRTC VoIP application using SIP in a browser environment.

The measurements done on the audio drivers show that they can cope with the CPU load created by a WebRTC-SIP application. Based on this and the knowledge gained from implementing such an application for use in a browser, two theoretically possible implementation methods are presented. The first solution builds on the WebRTC-SIP application done in step two, which utilises the support of WebRTC that is built into many browsers to power the application. The second solution is a application which uses a WebRTC to SIP gateway to allow it to set up calls to non-WebRTC applications.
# Contents

1 Introduction ......................................................... 1  
   1.1 Limes Audio ..................................................... 1  
   1.2 Purpose .......................................................... 2  

2 Background .......................................................... 3  
   2.1 Related Work ..................................................... 3  
      2.1.1 WebRTC and SIP Gateways ................................ 3  
      2.1.2 WebRTC on ARM .............................................. 4  
   2.2 Voice over Internet Protocol ................................... 4  
      2.2.1 Codecs ....................................................... 5  
      2.2.2 VoIP Security .................................................. 7  
   2.3 NAT traversal ..................................................... 7  
   2.4 Session Initiation Protocol .................................... 8  
   2.5 Web Real Time Communication ................................ 13  
      2.5.1 WebRTC communication .................................... 14  
   2.6 Session Description Protocol .................................. 15  
   2.7 Real-time Transport Protocol .................................. 16  

3 Evaluation of Linux Audio Driver Impact From CPU Load ...... 17  
   3.1 Background ....................................................... 17  
      3.1.1 Frames and Samples ......................................... 18  
   3.2 CPU load effect on ALSA audio driver ......................... 19  

4 WebRTC and SIP Browser-based Implementation ................. 23  
   4.1 Implementation overview ....................................... 23  
   4.2 Used Libraries and Software .................................. 24  
   4.3 WebRTC-SIP Implementation ................................... 24  

5 Results .................................................................. 27  
   5.1 CPU Load Effect on ALSA Audio Drivers ....................... 27  
   5.2 WebRTC-SIP ARM A9 implementation evaluation ............. 27  
      5.2.1 Web-based WebRTC-SIP Implementation on ARM .......... 28
5.2.2 Native implementation on ARM ........................................ 28

6 Conclusions ................................................................. 31
  6.1 Evaluation of Linux Audio Driver Impact From CPU Load ........ 31
  6.2 WebRTC and SIP Browser-based Implementation .................. 31
  6.3 Possibility of a WebRTC-SIP implementation running on ARM A9 processor  32
  6.4 Discussion ............................................................ 32
  6.5 Future work ............................................................ 32

7 Acknowledgements ......................................................... 35

References ................................................................. 37
List of Figures

2.1 Illustration of STUN server functionality. ................................. 8
2.2 Illustration of a TURN relay server ........................................ 9
2.3 Relationships of protocols in SIP systems [8]. ............................ 10
2.4 The modules included in the Googles WebRTC package. ............... 14

3.1 The Linux audio layer illustrated. [32] .................................... 18
3.2 Analog sinus wave [3] .......................................................... 18
3.3 Digital representation of analog sinus wave. [3] .......................... 19
3.4 Visual representation of a ring buffer. .................................... 19

4.1 Illustration of a Web-based WebRTC-SIP system ....................... 24
4.2 WebRTC-SIP system blocks and communication protocols. .......... 25
4.3 web-based WebRTC-SIP Application running in Chrome in a audio-video call using SIP messages. ..................................... 26

5.1 Illustration of a Web-based WebRTC-SIP implementation .......... 28
5.2 Illustration of a Native WebRTC-SIP implementation, where the container inside the ARM device represent the modules of the system. ........ 29
List of Tables

2.1 Comparison of voice of PSTN and VoIP [34, 22]. Note that Some values in the table might have changed since the comparison was made. 5
2.2 Common codec terms [35] 6
2.3 OPUS operating modes [29] 7
2.4 SIP methods [8] 10

3.1 Times and differences between stress and non-stress measurements. Cells with '-' indicate that the times did not have valid time due to overflow. 21
Chapter 1
Introduction

In the last decades, the need for good, cheap and reliable communication has increased. Voice over IP (VoIP) and its protocols drafted by the International Engineering Task Force (IETF) and World Wide Web Consortium (W3C) have provided a solution. VoIP is a methodology that refers to the delivery of multimedia and voice sessions over an Internet connection, which provides an alternative to regular phone lines usually referred to as the Public Switched Telephone Network (PSTN). In the early years of VoIP most calls were made using VoIP phones utilising only audio as the transferred media, however in recent years the demand for video calls has been increasing as well, thus the W3C and Google drafted a new standard; WebRTC (Web Real-Time Communication) is an API definition and acts as a foundation for browser-to-browser applications for voice, video, chat, and P2P file sharing without the need of either internal or external plugins. Usually the process of building a VoIP application can take a long time and require a lot of expertise, with Google WebRTC this is done very easy in a browser environment, leading to a larger group of developer that are now able to create VoIP application. The question is if the Google WebRTC package can be used on different platforms whilst making use of the existing structures and functionality. In this thesis I examine whether it would be easier or even possible to create a VoIP application in an ARM A9 embedded environment using WebRTC. I also investigated how to combine WebRTC with the Session Initiation protocol (SIP, Standard way of setting up VoIP calls) to enable calls to be set up to already existing VoIP solutions as well as investigating how such an WebRTC-SIP application would affect the Linux audio send/receive latency in the Linux audio driver.

The project was performed at Limes Audio AB in Umeå with external supervisor Emil Lundmark and Thomas Hellström as the internal supervisor at the Department of Computer Science with Umeå University.

1.1 Limes Audio

Limes Audio AB develops software for conference phones and video conferencing equipment mainly focusing on noise and echo suppression for audio. They have several customers that require adding VoIP features to their products. This implies that the system shall function as a telephone with the ability to communicate using the SIP to other SIP based Private Branch Exchange systems (PBXs), from vendors such as Cisco, Avaya, ShoreTel, Alcatel-Lucent, Mitel etc. A PBX is a non public telephone exchange system, usually used within companies to connect an internal telephone network with the outside world to other Public switched telephone network (PSTN) and VoIP systems. Traditionally such features have been supplied by chip
manufacturers that have provided complete System-on-Chip solutions running the SIP protocol including features such as media engine. A media engine is software which handles the sound and video sent over the network, dealing with things such as compression, sampling (codecs), echo suppression and other media enhancing software (Dr. Fredric Lindström, CEO of Sales Limes Audio AB).

1.2 Purpose

The initial goal was an VoIP implementation of Google WebRTC running on the ARM processor architecture, in which the echo cancellation software embedded in the package was to be replaced with Limes Audio echo cancellation software. The application should also use SIP as the signalling layer (this VoIP application will be referred to as WebRTC-SIP in the thesis). The point of this was to create a VoIP application using cutting edge web technology from Google and utilise the benefits of the package combined with the ARM processor which is widely used around the world [5, 15]. This goal, however, proved to be too time consuming given the time span for the project, instead a goal was defined as

(a) Evaluate the possibility of a WebRTC-SIP implementation running on ARM A9 processor architecture.

The evaluation was done in two steps to arrive at a conclusion on what is required for such an implementation. The two parts are: an analysis and tests of the Linux audio drivers on an ARM platform and a implementation of a WebRTC VoIP application using SIP in a browser environment. These steps both aim to answer a sub-question which are to be answered in the thesis, more precisely:

(b) How is the kernel level ALSA capture/playback interface affected by high CPU loads on a Cortex ARM A9 processor?

(c) How can a WebRTC-SIP application be built within a browser environment?

The report is divided in three parts where each step (previous work, ALSA audio drivers evaluation and Implementation) corresponds to a chapter in the thesis; the theoretical part can be viewed in Chapter 2, the evaluation of Linux audio driver under high CPU loads can be seen in Chapter 3 and the implementation of a web-based WebRTC-SIP client can be seen in Chapter 4.
Chapter 2

Background

This chapter will provide an overview and background of the Voice over IP (VoIP) technology, as well as related work. The chapter starts with a brief section about the some related projects and solutions, followed by the background of VoIP and information about Session Initiation Protocol (SIP), WebRTC, Real-Time Transfer Protocol (RTP) and Session Description Protocol (SDP).

2.1 Related Work

To our knowledge, application using WebRTC with SIP running on a ARM architecture to create an embedded VoIP solution has not yet been attempted. However we look at areas that are related: WebRTC and SIP gateways which are covered in Section 2.1.1 and how to compile WebRTC for ARM which is covered in Section 2.1.2.

2.1.1 WebRTC and SIP Gateways

There are a lot of projects on incorporating SIP with WebRTC using a gateways which the WebRTC clients uses to communicate through, this method however separates the clients from the gateway which can be located anywhere usually hosted by a company [38].

When the application makes a call to another user, the WebRTC gateway checks whether the user is either another WebRTC application, in that case, the call is made as usual for the WebRTC application. Otherwise the gateway has to translate a couple of things, first of is the signalling layer, which there is no real standard for in WebRTC. However it is possible for the application to make use of SIP over WebSocket, in that case the gateway only repackage from WebSocket into UPD or another fitting protocol. Otherwise the gateway has to translate the signalling layer from WebRTC. Second is the fact that the WebRTC application must use the Secure Real Time Protocol (SRTP), however, some VoIP applications and phones do not implement this and if so the gateway must convert between RTP (Real Time Protocol) and SRTP. Thirdly, the gateway might have to convert between codecs, since the codecs supported by different VoIP applications might differ. Lastly is the NAT traversal issue, which is a method of getting around the fact that a lot of nodes reside behind a NAT, making it hard to retrieve the external IP address. This is automatically done in WebRTC however not in all VoIP application, this is also a feature that is performed in a gateway [38, 4].

A list of current open source variants can be seen in the list below.

- OverSIP
Kamailio
- Asterisk
- reSIProcate and repro
- WebRTC2SIP
- Janus
- FreeSWITCH
- SylkServer

The benefits from this is that WebRTC clients can communicate with any WebRTC and SIP clients without implementing SIP functionality within the application [4].

2.1.2 WebRTC on ARM

As to projects that make use of WebRTC on the ARM architecture, to our knowledge there does not exist any big or widely successful projects, there are however a couple of groups that are trying to compile WebRTC for the platform with varied results, and from reports from these groups it seems that it might work. It is reported that some programmers in a discussion group [10] have successfully managed to compile WebRTC for ARM using custom made CMake files. In the group a programmer from China has done it and provided a short guide which can be viewed and downloaded at [16]. The translation of the method given by this post is roughly: ”The basic principle is to compile the source files together with the correct parameters, so the basic approach here is to abandon GYP which is the standard compile tool for Google WebRTC and instead use CMAKE to re-write a script compiler. The steps are to use script and pick a source file and then transform it into CMAKE, include the ARM CPU-related parameters, and finally try to build. The whole process continues recursively until successfully constructed [16].” The post also provides a compiled version of the library which might be useful in future work for this thesis.

2.2 Voice over Internet Protocol

VoIP is a methodology that refers to the delivery of multimedia and voice sessions over an Internet connection, which provides an alternative to regular phone lines usually referred to as the Public Switched Telephone Network (PSTN).

Voice has been transmitted over the PSTN since about 1878 in the United States and the long distance market has grown to about $100 billion a year in business and residential demand. This has been a driving factor for companies to reduce this cost and come up with new solutions. Since the cost of package-switched networks is almost half of that of a circuit-switched network costs it was a suitable platform for VoIP applications [34].

A package-switched network is a digital networking method where the transmitted data is divided into groups of data of certain sizes called packets and then sent over a network. The package-switched network uses dynamic allocation of transmission bandwidth which means it allocates transmission resources as they are needed using statistical multiplexing or dynamic bandwidth allocation techniques, granting it more reliability, flexibility and frees up resources. When the package moves between network nodes, such as switches and routers, the packets are buffered and queued in the within each node. This buffering and queueing can result in variable
2.2. Voice over Internet Protocol

<table>
<thead>
<tr>
<th>Concept</th>
<th>Voice over PSTN</th>
<th>VoIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switching</td>
<td>Circuit-switched</td>
<td>Packet-switched</td>
</tr>
<tr>
<td>Bit rate</td>
<td>64kbs or 32kbs</td>
<td>14kbs (Depends on codec)</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>Dedicated</td>
<td>Dynamically allocated</td>
</tr>
<tr>
<td>Quality of Service</td>
<td>High (Extremely low loss)</td>
<td>Low and variable, depending on the network</td>
</tr>
<tr>
<td>Security</td>
<td>High, each line is dedicated</td>
<td>Possibility of eavesdropping</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of voice of PSTN and VoIP [34, 22]. Note that Some values in the table might have changed since the comparison was made.

Latency and throughput of the packages depending on the link capacity and the traffic load on the network and in turn affect the Quality of Service (QoS) [25]. Where QoS is the overall performance of a network such as telephone or computer networks and a high QoS shows a good performance in the network.

A circuit-switched network which is used within the PSTN uses a pre-allocated dedicated network bandwidth, meaning that each connection has a constant bit rate and latency. This prevents the congestion of the data, since there is no interference on the network giving it a high QoS [25, 22].

A comparison of voice over PSTN and VoIP are shown in table 2.1.

2.2.1 Codecs

To be able to effectively send voice data over a network without using large amounts of bandwidth there exists codecs. A codec is a method of converting an analogue signal into a digital signal called PCM (see Section 3.1.1), that can be sent over a network. A VoIP application usually implements several codecs to allow it to negotiate what kind of codec that is to be used during a call. Each codec has a specific way to package and send voice over a network see Table 2.2 for commonly used terms for codecs [35].

To ensure a baseline level of interoperability between WebRTC (see Section 2.5) endpoints, a minimum set of required codecs have been set up by the Internet Engineering Task Force (IETF). And the codecs are OPUS (RFC6716) with payload format specified in (RFC7587) and G.711 PCMA and PCMU with the payload format specified in section 4.5.14 of (RFC3551).

G.711 is a high bit rate International Telecommunications Union (ITU) standard codec, which is the standard and most used codec used in modern digital telephone networks. The codec is also know as PCM (see Section 3.1.1) and is a narrow-band audio codec that provides audio at 64 kbit/s. G.711 passes audio signals in the range of 300–3400 Hz, samples them at the rate of 8,000 samples per second and 8 bits is used to represent each sample [9].

OPUS is designed as a real-time audio codec which utilises two layers: Linear Prediction (LP) which is a mathematical operation where future values of a discrete-time signal can be estimated as a linear function of previous samples [1], and a layer based on the Modified Discrete Cosine Transform (MDCT) which is a linear discrete block transformation [14]. The idea behind using two layers is that the OPUS codec can operate over a wider range of sound [29].

The codec can scale from 6 kbit/s narrowband (NB) mono speech to 510 kbit/s fullband (FB) stereo music, with algorithmic delays ranging from 5 ms to 65.2 ms. The codec can use the LP layer or the MDCT layer, or both at the same time, granting the possibility to seamlessly switch between all of its various operating modes (See Table 2.3). This gives the codec a great deal of flexibility to adapt to various voice data and network conditions without being required to renegotiate for example a VoIP session [29][35].
<table>
<thead>
<tr>
<th>Term</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Codec Bit Rate (Kbps)</td>
<td>Based on the codec, this is the number of bits per second that need to be transmitted in order to deliver a voice call. (codec bit rate = codec sample size / codec sample interval).</td>
</tr>
<tr>
<td>Codec Sample Size (Bytes)</td>
<td>Based on the codec, this is the number of bytes captured by the Digital Signal Processor (DSP) at each codec sample interval. For example, a codec operates on sample intervals of 10 ms, which corresponds to 10 bytes (80 bits) per sample at a bit rate of 8 Kbps. (codec bit rate = codec sample size / codec sample interval)</td>
</tr>
<tr>
<td>Codec Sample Interval (ms)</td>
<td>This is the sample interval at which the codec operates. For example, a codec that operates on sample intervals of 10 ms, which corresponds to 10 bytes (80 bits) per sample at a bit rate of 8 Kbps. (codec bit rate = codec sample size / codec sample interval)</td>
</tr>
<tr>
<td>Voice Payload Size (Bytes)</td>
<td>The voice payload size represents the number of bytes (or bits) that are filled into a packet. The voice payload size must be a multiple of the codec sample size. For example, a codec can use 10, 20, 30, 40, 50, or 60 bytes of voice payload size.</td>
</tr>
<tr>
<td>Voice Payload Size (ms)</td>
<td>The voice payload size can also be represented in terms of the codec samples. For example, a codec voice payload size of 20 ms (two 10 ms codec samples) represents a voice payload of 20 bytes [ (20 \text{ bytes} \times 8) / (20 \text{ ms}) = 8 \text{ Kbps} ]</td>
</tr>
</tbody>
</table>

Table 2.2: Common codec terms [35]
2.3. NAT traversal

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Audio BandWidth</th>
<th>Sample Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>NB (narrowband)</td>
<td>4 kHz</td>
<td>8 kHz</td>
</tr>
<tr>
<td>MB (medium-band)</td>
<td>6 kHz</td>
<td>12 kHz</td>
</tr>
<tr>
<td>WB (wideband)</td>
<td>8 kHz</td>
<td>16 kHz</td>
</tr>
<tr>
<td>SWB (super-wideband)</td>
<td>12 kHz</td>
<td>24 kHz</td>
</tr>
<tr>
<td>FB (fullband)</td>
<td>≤ 20 kHz</td>
<td>48 kHz</td>
</tr>
</tbody>
</table>

Table 2.3: OPUS operating modes [29].

2.2.2 VoIP Security

As more companies look to cut their telephone expenses and turn to VoIP systems as the solution, more systems become subjects of intrusions - Some targeted directly at the systems and protocols used in VoIP and some targeted at the IP architecture. Some threats against the IP architecture is a Denial of Service (DoS) attack, which is a method that attempt to make a certain service unavailable by typically flooding the target network or resources with requests. This makes it hard for the system to process requests sent from legitimate users and as an effect rendering the service unavailable [33].

A source of vulnerability of VoIP system lies in the protocols used for call management, for example SIP (see Section 2.4). This is a weakness since SIP transmits packet headers and payload in clear text without a message integrity check (using for example digitally signed messages), which allows for intruders to terminate or redirect ongoing calls. As well as security flaws in the IP architecture and call management protocols, there exist vulnerabilities in the Real-Time Protocol (see Section 2.7) due to the absence of authentication, which can allow an attacker using the sequence numbers in the headers play back voice packages in the correct order or inject own packages. This flaw can be circumvented using Secure Real-Time Protocol (see Section 2.7) which requires additional time for encryption and decryption, making it more vulnerable to DoS attacks. These attacks are classified as four different categories which can be seen in the list below [47, 19].

**Signal protocol attack** Attacks which exploit vulnerabilities in the signaling protocol. For example, the SIP protocol contains holes in the subset related to invite messages, which is a message sent to invite a user to a call.

**Redirect attacks** A redirect attack might change a voice mail address or a call forwarding address to the address of a hacker.

**Call interception** An unauthorised person could monitor and intercept voice packets - perhaps reading and stealing or corrupting them.

**Toll fraud** An unauthorized person could monitor and intercept call setup packets, gaining sufficient information to falsely legitimate as a user and make fraudulent calls.

2.3 NAT traversal

Network address translation (NAT) is an effective technique for delaying the exhaustion of the address pool of Internet Protocol version 4 (IPv4) and is widely used in many home and corporate networks. NAT gateways work by tracking outbound requests from a network and maintaining the state of each established connection to later direct responses from the peer on the public network to the peer in the private network [26].
Chapter 2. Background

Figure 2.1: Illustration of STUN server functionality.

A large problem for VoIP solutions is the large usage of NAT routers, hiding the IP of the client from the public Internet, in order to be able to create a connection between clients behind NATs devices, WebRTC and other VoIP solutions use ICE and STUN servers [26].

Interactive Connectivity Establishment (ICE) is a technique used in many VoIP and peer-to-peer solutions to traverse Network Address Translators (NATs). The technology allows clients or "agents" to communicate even if they are located behind firewalls and routers. ICE is a method used for finding the best way of setting up connections between peers in such cases. ICE is usually used in concert with Session Traversal Utilities for NAT (STUN) and Traversal Using Relays around NAT (TURN) servers [18, 26].

A STUN server provides means for an endpoint to determine the IP address and port allocated by a NAT that corresponds to its private IP address and port. It also provides a way for an endpoint to keep a NAT binding alive [45, 26]; a basic illustration of a STUN server can be seen in Figure 2.1.

If a peer is unable to set up a connection to another peer using STUN servers, caused by for example symmetric NATs. Symmetric NATs works in the way that each request from the same internal IP address and port to a specific destination IP address and port is mapped to a unique external source IP address and port. In the case where the NAT changes port each time a new request is sent, making a peer connection impossible, a new method is used. TURN works by relaying the communication data through a server that exists on the public Internet, it works by allowing a client to obtain IP addresses and ports on that relay. This allows a peer to relay and to exchange packets with its peers using the relay instead of a direct connection, it also allows a peer to communicate with multiple peers using a single relay address. The use of TURN servers are only used as a last resort since it puts a significant load on the TURN server and the provider that owns it [46]. Illustration of how TURN servers work can be seen in Figure 2.2.

2.4 Session Initiation Protocol

The Session Initiation Protocol (SIP) is a protocol defined by the Internet Engineering Task Force (IETF) and published in RFC 3261 [6] and is today a standard for initialising, changing and terminating interactive sessions for multimedia. SIP is especially known for the possibilities
2.4. Session Initiation Protocol

The SIP protocol is message based and reminds of the HTTP protocol using requests and responses that can be sent over UDP, TCP and SCTP [6, 20].

SIP is an application layer protocol that is reminiscent of HTTP with requests and responses and can be sent over UDP, TCP and SCTP. Each transaction between SIP nodes consists of a client request that invokes a particular method on the server and at least one response is sent back to the client. SIP reuses the header fields and encoding rules of HTTP allowing for easy readability of the messages for humans [20].

The methods used in SIP can be seen in Table 2.4 and responses and response codes can be seen in the following list [6, 8]:

- **Provisional (1xx)** Request received and being processed.
- **Success (2xx)** The action was successfully received, understood, and accepted.
- **Redirection (3xx)** Further action needs to be taken to complete the request.
- **Client Error (4xx)** The request contains bad syntax or cannot be fulfilled at the server.
- **Server Error (5xx)** The server failed to fulfil an apparently valid request.
- **Global Failure (6xx)** The request cannot be fulfilled at any server.

The relationship between SIP and other protocols in a SIP system can be viewed in Figure 2.3 and for specification of the protocols see Section 2.6 for SDP and Section 2.7 for RTP.

An example of how a SIP call is set up was taken from [6] RFC 3261 and shows an exchange of request and response where Alice sends a INVITE to Bob via two proxy servers and Bob responds with a 200 OK message in return (see Listings 2.1,2.2):
Table 2.4: SIP methods [8]

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVITE (INV)</td>
<td>User Agent (UA) starts a dialog and session</td>
</tr>
<tr>
<td>Cancel (CAN)</td>
<td>Only for INVITE, and only after recipient of provisional response.</td>
</tr>
<tr>
<td>Acknowledgement (ACK)</td>
<td>From calling UA client to called UA server after a final response.</td>
</tr>
<tr>
<td>Register (REG)</td>
<td>UA transmits location and other information to front end process of location service</td>
</tr>
<tr>
<td>Goodbye/quit (BYE)</td>
<td>Always sent within a dialog to close connection, ends session and dialog.</td>
</tr>
<tr>
<td>Options (OPT)</td>
<td>Request and deliver specific about the services supported by the other party in a dialog.</td>
</tr>
<tr>
<td>Provisional ACK (PRACK)</td>
<td>Used to ACK provisional response</td>
</tr>
<tr>
<td>Update (UP)</td>
<td>Updates session information before completion of INV-OK-ACK handshakre.</td>
</tr>
<tr>
<td>Message (MSG)</td>
<td>Delivers instant messaging</td>
</tr>
<tr>
<td>Publish (PUB)</td>
<td>Makes and event state available.</td>
</tr>
<tr>
<td>Info (INFO)</td>
<td>Send application data without changing state of session.</td>
</tr>
<tr>
<td>Subscribe (SUB)</td>
<td>Allows UA to ask for notification about some change.</td>
</tr>
</tbody>
</table>

IP network layer packets (IPv4, IPv6, or both)

Data link layer frames

Physical layer connectivity

Figure 2.3: Relationships of protocols in SIP systems [8].
Listing 2.1: SIP example

```
 atlanta.com . . . biloxi.com
   .   proxy   proxy   .
```

Alice’s softphone . . . . . . . . . . . . . . Bob’s SIP Phone

INVITE F1 | INVITE F2 | 
100 Trying F3 | 100 Trying F5 | INVITE F4 |
<---------- | <---------- | <---------- |
| 180 Ringing F6 |
| 180 Ringing F7 |
<---------- | 200 OK F10 | 200 OK F11 |
<---------- | <---------- |
| 200 OK F12 |
| Media Session |
<---------- |
| BYE F13 |
<---------- |
| 200 OK F14 |
<---------- |

The message that was sent from Bob is seen in 2.2.

Listing 2.2: SIP example

```
INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bK776asdhds
Max-Forwards: 70
To: Bob <sip:bob@biloxi.com>
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq: 314159 INVITE
Contact: <sip:alice@pc33.atlanta.com>
Content-Type: application/sdp
Content-Length: 142
```
As seen in Listing 2.2 we see the following header fields example and are described in more detail in Table 2.4 and the following list [8].

**INVITE** The method name, recipient and protocol specifications.

**Via** Contains the address at which Alice will expect to receive responses from for this request as well as a branch id which identifies the transaction.

**Max-Forwards** The limit of hops the request can transit on the way to its destination.

**To** Contains a display name (Bob) and a SIP or SIPS URI (sip:bob@biloxi.com) towards which the request was originally directed.

**From** Contains a display name (Alice) and a SIP or SIPS URI (sip:alice@atlanta.com) that shows the originator of the requests as well as a tag parameter that is used for identification.

**Call-ID** Contains a globally unique identifier for this call.

**CSeq** Contains an integer and a method name, the integer is incremented for each request and serves as a sequence number.

**Contact** Contains a SIP or SIPS URI that represents a direct route to contact Alice.

**Content-Type** Contains a description of the message body.

**Content-Length** Contains the byte count of the message body.

Once Bob’s SIP phone receives the INVITE and alerts Bob to the incoming call from Alice i.e. Bob’s phone rings. Bob’s SIP phone indicates this in a 180 (Ringing) response, which is routed back through the two proxies in the reverse direction. Once Bob picks up the call, a 200 OK response message is relay back to Alice and has the following format:

```
SIP /2.0  200 OK
Via: SIP /2.0/UDP server10.biloxi.com
     ;branch=z9hG4bKnashds8;received=192.0.2.3
Via: SIP /2.0/UDP bigbox3.site3.atlanta.com
     ;branch=z9hG4bK77ef4c2312983.1;received=192.0.2.2
Via: SIP /2.0/UDP pc33.atlanta.com
     ;branch=z9hG4bK776asdhds;received=192.0.2.1
To:  Bob <sip:bob@biloxi.com>;tag=a6c85cf
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq: 314159 INVITE
Contact: <sip:bob@192.0.2.4>
Content-Type: application/sdp
Content-Length: 131
```

The first line of the response contains the response code (200) and the reason phrase (OK). The remaining lines contain header fields. The Via, To, From, Call-ID, and CSeq header fields are copied from the INVITE request. (There are three Via header field values – one added by Alice’s SIP phone, one added by the atlanta.com proxy, and one added by the biloxi.com proxy.) Bob’s SIP phone has added a tag parameter to the To header field.

Once the initiation of the call is completed the Session Description protocol (SDP) takes over and negotiates the details for the call (see Section 2.6) The session continues with a stream of media using the Real-Time Protocol (RTP) (see Section 2.7). Once the call is finished it will be
terminated by sending a `BYE` request message to the other client and will be accepted using a `200 OK` response message [13][6].

The SIP takes on a client-server approach where both clients send and receive both requests and responses similar to how HTTP works. A registrar is usually used in a SIP application and is end point that accepts REGISTER requests and places the information about the client in a location service for the domain it handles. The location service links one or more IP addresses to the SIP URI of the registering client. SIP registrars are logical elements, and are commonly co-located with SIP proxies [8].

### 2.5 Web Real Time Communication

WebRTC (Web Real-Time Communication) is an API definition drafted by the World Wide Web Consortium (W3C) and acts as a foundation for browser-to-browser applications for voice, video, chat, and P2P file sharing without the need of either internal or external plugins [36]. The API defines a method that differs from existing methods by allowing communication between browsers directly using peer-to-peer instead of passing the data through a server, hence only serving as a platform to relay connection negotiation data needed to set up the connection [36]. Since WebRTC is a multi-platform technology, different implementations of WebRTC may obey or disown RFC specification requirements [7]. This study will only focus on the version of WebRTC that is developed by Google which was released in May 2011, as an open source project for browser-based real-time communications. The WebRTC package contains all the building blocks needed to set up calls and send media between supported browsers using a JavaScript API, which makes it possible for most developers to make use of codecs and echo cancelling software without substantial knowledge in the field [43]. An image with a description of the components in the package can be seen in Figure 2.4 and in the following list. However to allow WebRTC to make calls to non-WebRTC VoIP applications, a initiation protocol is needed that is not included in the Google WebRTC implementation (see Section 2.4).

- JavaScript API.
- C++ Peer-Connection API.
- Session manager.
- Voice engine including codecs and echo cancellation.
- Video engine, which I will not utilise.
- Transport block which handles the sending and receiving of media using SRTP (Secure Real-time Transport Protocol) and NAT traversal.
- Audio capture, Video capture and Network I/O.

The *WebRTC Native Code package* is mainly created for browser developers who want to integrate WebRTC into browsers, but can also be used to create native application using the C++ API. Web application developers use the WebRTC Javascript API which is created and designed for usage inside a supported desktop browser, for example: *Chrome, Firefox, Opera, Microsoft Edge* and lately added support for some mobile browsers for iOS and Android [43].

The Google WebRTC package makes it easier for web and mobile developers to create communication applications using the built in Javascript, JAVA and C++ APIs. WebRTC allows web browsers to set up a direct peer-to-peer connection and then uses that connection to negotiate session details using SDP and then sends media over Secure Real Time Protocol (SRTP). WebRTC communication must be secured using TSL, HTTPS and SRTP [43, 36].
2.5.1 WebRTC communication

As defined by W3C: “WebRTC Communications are coordinated via a signaling channel which is provided by unspecified means, but generally by a script in the page via the server, e.g. using XMLHttpRequest or Web Sockets[36].”

XMLHttpRequest is an object which supports any text based format (including XML) that was moved over to the W3C group from the Web Hypertext Application Technology Working Groups (WHATWG) HTML effort. The object can be used to make requests on either HTTP or HTTPS and supports all requests and response standards for HTTP [48].

The Web Sockets is an API which allows web applications to maintain bidirectional communications with server-side processes [31].

Listing 2.3: The basic scenario of setting up calls in WebRTC in a browsers is done as follows (suppose there exist a peer1, peer2 and web server)

```java
1 peer1 connects to server, wait for the other.
2 peer2 connects to server.
3 server internally makes a 1:1 communication channel for peer1 & peer2
4 peer2 start ICE procedure.
5 peer2 sends ICE candidate message to server.
6 server blindingly forward this message to peer1.
7 peer1 receives ICE message and then send reply.
8 server forward this message to peer2.
9 steps repeats until peer1 & peer2 establish an p2p channel.
10 Real data transfer begins.
```
2.6 Session Description Protocol

After a call has been initiated and accepted by at least two participants, there is a requirement to negotiate media details such as; transport addresses, codecs and other session description meta data. This negotiation of media details is handled by the Session Description Protocol (SDP) defined in RFC 4566 [13]. The protocol is purely a format for session description and can be sent over any transport protocol, for example SIP and HTTP. An SDP session descriptor consists of a number of lines with the syntax:

\[
<\text{type}>=<\text{value}>
\]

The allowed character set for SDP descriptors is defined as the ISO 10646 set in the UTF-8 encoding format. The layout of such a session description message can be seen below in Figure 2.4 where types set to “*” are OPTIONAL whilst the rest are REQUIRED. The order of the descriptors must be in the order shown in Listing 2.4 to be able to be parsed, an example SDP message can be seen in Listing 2.5 [13].

Listing 2.4: SDP message layout

```
<table>
<thead>
<tr>
<th>Session description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v= (protocol version)</td>
</tr>
<tr>
<td>o= (originator and session identifier)</td>
</tr>
<tr>
<td>s= (session name)</td>
</tr>
<tr>
<td>i=* (session information)</td>
</tr>
<tr>
<td>u=* (URI of description)</td>
</tr>
<tr>
<td>e=* (email address)</td>
</tr>
<tr>
<td>p=* (phone number)</td>
</tr>
<tr>
<td>c=* (connection information — not required if included in all media)</td>
</tr>
<tr>
<td>b=* (zero or more bandwidth information lines)</td>
</tr>
<tr>
<td>One or more time descriptions (&quot;t=&quot; and &quot;r=&quot; lines; see below)</td>
</tr>
<tr>
<td>z=* (time zone adjustments)</td>
</tr>
<tr>
<td>k=* (encryption key)</td>
</tr>
<tr>
<td>a=* (zero or more session attribute lines)</td>
</tr>
<tr>
<td>Zero or more media descriptions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time description</th>
</tr>
</thead>
<tbody>
<tr>
<td>t= (time the session is active)</td>
</tr>
<tr>
<td>r=* (zero or more repeat times)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Media description, if present</th>
</tr>
</thead>
<tbody>
<tr>
<td>m= (media name and transport address)</td>
</tr>
<tr>
<td>i=* (media title)</td>
</tr>
<tr>
<td>c=* (connection information — optional if included at session level)</td>
</tr>
<tr>
<td>b=* (zero or more bandwidth information lines)</td>
</tr>
<tr>
<td>k=* (encryption key)</td>
</tr>
<tr>
<td>a=* (zero or more media attribute lines)</td>
</tr>
</tbody>
</table>
```
2.7 Real-time Transport Protocol

The open standard protocols that the public Internet is built on are not created for real time communication, especially TCP/IP [30]. To help with the emergence of real time communication over the Internet, the IETF drafted a real-time communication protocol by the name of RTP as is seen in RFC 3550 [17].

The protocol provides a real-time end-to-end service with supporting for example:

- Payload type identification
- Sequence numbering
- Time stamping
- Delivery monitoring

Applications that run RTP usually run on top of UDP to utilise multiplexing and checksum services, which also makes it easier and faster to broadcast multimedia to multiple recipients which is the primarily focus for RTP. The protocol completely relies on underlying protocols for delivery, but the sequence numbers in the protocol are used to reconstruct the messages on the recipient side, which makes it possible for codecs to process samples and place them in the right order before all the data has arrived [17].

There is also a version of RTP called SRTP which provides another layer of security on top, by providing encryption, message authentication and integrity checks of messages [17].

RTCP stands for RTP control protocol and is a way for participants to communicate feedback on the quality of the data distribution to the other participants and utilises a own UDP port. RTCP is also useful to observer and evaluate problems in the network locally or globally [17].

RTP creates separate RTP Sessions for transmission of video and audio, meaning that different RTP and RTCP packets are sent using two different UDP ports, this is done to allow the recipients to choose which mediums are used. A RTP-Session is an association between two participants in a multimedia session, there can be several sessions to different participants and media type, a RTP-Session usually consists of a RTP connection and a RTCP connection [17].
Chapter 3

Evaluation of Linux Audio Driver Impact From CPU Load

This chapter aims to answer the question:

(b) "How is the kernel level ALSA capture/playback interface affected by high CPU loads on a Cortex ARM A9 processor?"

The chapter contains the background and related details of how the ALSA audio drivers work in the Linux kernel (see Section 3.1). Here we perform some latency tests to gain a better understanding of how a VoIP application such as WebRTC could affect the delay of capture and playback interfaces of the Linux kernel drivers on the targeted ARM A9 processor. The tests are presented in Section 3.2.

3.1 Background

Advanced Linux Sound Architecture (ALSA) is a software framework that provides an API interface for programs to access audio and MIDI functionality to the Linux kernel with fully modularised sound drivers. The ALSA framework is a part of the Linux kernel and was initially released in 1998. The goal of ALSA was to be able to automatically configure the environments for different sound card hardware and to be able to handling multiple sound devices. ALSA is a project released under GNU General Public License [11].

The system that was widely used before ALSA was the Open Sound System (OSS) project which is an interface for making and capturing sounds in UNIX operating systems. ALSA has a bigger and more complex API than OSS making it somewhat harder to use; however, some versions of ALSA may be configured to support an OSS emulation layer [12, 11]. ALSA also provides a high-level interface for developers which is called “user-space-library” which can be used to have direct interaction with the kernel sound devices that is standardised over many platforms [11, 44, 12]. The overview of the Linux audio layers can be seen in Figure 3.1.

ALSA arranges audio devices into a hierarchy of cards, devices and sub devices. Each ALSA card corresponds directly to a hardware sound card which can be identified by an ID or a numerical index typically indexes from 0 to 7 (eight cards in total). In each card, there exists a set of devices also numbered starting from zero; these devices determine over which connector or set of connectors audio signals will be sent. At the lowest level there are sub-devices e.g. channels...
also numbered starting with zero. Usually it is enough to specify the card and the device to play a sound [12, 11].

ALSA consists almost entirely of plugins that performs the work on different ALSA devices (different from the hardware devices mentioned before) which are wrappers for the plugins. The most important and most frequently used plugin is the “hw” plugin which accesses the hardware driver but does not do any processing itself; it can be accessed by hw:0,0 which is demonstrated in the experiments [27, 12, 11].

### 3.1.1 Frames and Samples

The key component of digital audio processing is Pulse-code modulation (PCM), which is a method to digitally represent real analogue audio signals. In a PCM audio stream, the amplitude of the analog audio is sampled at regular intervals to represent the audio (see Figure 3.2 and 3.3). The individual measurements are called samples and a set of samples is called a frame. The number of samples in a frame can vary from one sample to several thousands depending on the implementation used. Some frame sizes as they are reported in [2] can be seen below.

1 frame of a Stereo 48khz 16 bit PCM stream is 4 bytes.

1 frame of a 5.1 48khz 16 bit PCM stream is 12 bytes.

Generally: 1 frame = (num_channels) * (1 sample in bytes)

To handle incoming audio to the device from external sources, ALSA implements a ring-buffer which temporarily holds frames of a PCM stream that are captured by for example a microphone or USB device. A ring-buffer is a buffer which is wrapped, which means that the last index of the buffer points to the first index (see Figure 3.4).
3.2 CPU load effect on ALSA audio driver

To fetch the frames collected in the ring-buffer, ALSA sends interrupts to the CPU using certain intervals depending on the implementation. The interrupts halt other processes and allowing ALSA to pick up the frame(s) from the ring-buffer for further processing. The time between these interrupts is determined by the size of a period which is expressed in a number of frames which is can be set dependent of the implementation [2]; an example from [2] can be seen below.

If we set 16-bit stereo @ 44.1Khz, and the period size to 4410 frames

$$\Rightarrow$$ for 16-bit stereo @ 44.1Khz, 1 frame equals 4 bytes – so 4410 frames equal $4410 \times 4 = 17640$ bytes

$$\Rightarrow$$ an interrupt will be generated each 17640 bytes – that is, each 100 ms.

Correspondingly, buffer size should be at least $2 \times$ period size =

$2 \times 4410 = 8820$ frames (or $8820 \times 4 = 35280$ bytes).

3.2 CPU load effect on ALSA audio driver

The goal of the tests presented in this section is to examine whether a WebRTC-SIP application running a default priority could affect the performance in the ALSA audio driver. And with a focus on how the latency between the capture and playback interfaces are affected within the PCM ring-buffer as an effect of how the CPU-interrupts would be delayed from a high CPU load.

The tests were performed using a latency test program provided by the ALSA project written in C. The experiments were performed 40 times for each buffer-sizes, half of the tests were performed whilst the CPU load was at 100% to see if it would take longer for the interrupts to occur [23].

The test were performed on the following platform:

Hardware
The program was somewhat modified in order to measure the wall time of the write and read latencies of a certain number of frames between the capture and playback interfaces for a set of frames from the ring buffer; they were measured by using the difference between when the playback and capture is started.

Example:
```
> ./latency -P hw:0,0 -C hw:0,0 -r 44100 -m 128 -M 128 -p
```

The period used in the experiments are set as:
```
period_size = bufffersize / 2;
```

The buffer size is set as the value for the flags for “m” and “M” and the number of frames measured in each buffer size is:
```
frames_measured = buffersize * 100
```

The buffer sizes used in the experiments can be seen on the left hand side in Table 3.1.

To stress the CPU we use two methods; To create a steady load of about 25% on one core on the system the command `cat /dev/zero >> /dev/null` is used. To create some random peaks in CPU usage we make use of a custom made program that peaks the CPU to 100% for random amounts of ms with random sleep between the spikes. The test program uses scheduling priority `SCHED_OTHER`; the standard round-robin time-sharing policy, giving it the same priority as other processes. Since the VoIP application would be run using this priority it was decided that the tests running with higher priority would not be needed, to allow for a more realistic setting for this thesis.

The results given in ms times displayed from the test derive from times retrieved in nanoseconds from the `<time.h>` library using the `timespec struct` counting the nanoseconds, this value was then divided to display the values shown in milliseconds, hence some data are lost in the process. To make sure that when the long containing the nanoseconds overflowed due to long test times, we also compared other timers with a lower sensitivity to verify the correctness. The values where this long has overflowed is not displayed. The differences in ms between non-stress runs and stress runs can be seen in table 3.1 and the mean calculations for each buffersize is calculated using this formula:

\[
v = mean(time_{buffersize\ stress}) - mean(time_{buffersize\ no\ stress})
\]  
(3.1)

And for the min and max diff values, the values produced by formula 3.1 is extracted for min and max see formulas 3.2 and 3.2:

\[
max\ diff = max(v)
\]  
(3.2)
### 3.2. CPU load effect on ALSA audio driver

<table>
<thead>
<tr>
<th>Buffersize</th>
<th>mean non-stress(ms)</th>
<th>mean stress(ms)</th>
<th>max diff(ms)</th>
<th>min diff(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>72.6537</td>
<td>72.7167</td>
<td>0.0926</td>
<td>-0.0260</td>
</tr>
<tr>
<td>128</td>
<td>145.1770</td>
<td>145.2366</td>
<td>0.0893</td>
<td>0.0149</td>
</tr>
<tr>
<td>256</td>
<td>290.3439</td>
<td>290.4057</td>
<td>0.0966</td>
<td>-0.0379</td>
</tr>
<tr>
<td>512</td>
<td>580.5523</td>
<td>580.6138</td>
<td>0.0863</td>
<td>0.0136</td>
</tr>
<tr>
<td>1024</td>
<td>1161.1110</td>
<td>1161.1579</td>
<td>0.0873</td>
<td>0.0103</td>
</tr>
<tr>
<td>2048</td>
<td>-</td>
<td>-</td>
<td>-0.0033</td>
<td>-0.1023</td>
</tr>
<tr>
<td>4096</td>
<td>-</td>
<td>-</td>
<td>0.0790</td>
<td>-0.028</td>
</tr>
<tr>
<td>8192</td>
<td>-</td>
<td>-</td>
<td>0.0546</td>
<td>0.0043</td>
</tr>
<tr>
<td>16384</td>
<td>-</td>
<td>-</td>
<td>0.1423</td>
<td>-0.0069</td>
</tr>
<tr>
<td><strong>Means</strong></td>
<td>-</td>
<td>-</td>
<td><strong>0.08055</strong></td>
<td><strong>-0.0175</strong></td>
</tr>
</tbody>
</table>

Table 3.1: Times and differences between stress and non-stress measurements. Cells with '-' indicate that the times did not have valid time due to overflow.

\[
min_{diff} = min(v) \tag{3.3}
\]

The min/max differences can take either negative or positive values, times that are negative mean that the experiments with high CPU load performed faster overall than the experiments with the same buffer size but no additional CPU load. The results of the tests can be seen in Section 5.1 and the conclusion can be viewed in Section 6.1.
Chapter 4

WebRTC and SIP Browser-based Implementation

This chapter aims to answer the question:

(c) "How can a WebRTC-SIP application be built within a browser environment?"

To be able to evaluate the technology of WebRTC and investigate how to incorporate SIP, a web-based implementation was constructed using the built-in WebRTC stack and the JavaScript API in the browser. The benefits from this is that we were able to make use of the entire WebRTC stack without the need to create modules for example audio and video capture which is handled within the browser itself.

4.1 Implementation overview

Here we show two ways of implementing a VoIP client which uses WebRTC and SIP; the first way is to make use of a WebRTC to SIP gateway (See Section 2.1.1) which uses an external server which translates WebRTC layers into SIP – this approach is also a solution which could be implemented in a native application on the ARM processor. The other way is to use a SIP signal layer on top of WebRTC JavaScript API which is a way of creating a more compact system removing the need for external servers [37].

To create a Web-based implementation, a couple of things are required for the application to work. WebRTC requires a secure HTTPS connection to the web-server which hosts the files to allow for the browser to grab the audio and video via the getUserMedia API. The API is related to WebRTC in browser since it is the gateway to the audio and video APIs in WebRTC and provides the means to access the user’s local camera/microphone streams. To create a HTTPS service, it is possible to buy certificates or to create self-signed certificates generated by the OpenSSL library for Linux [24].

The PBX that is to be used as the "telephone switch" in the system must be able to handle both UDP and WebSocket connections for the system to work. Most open source PBXs support UDP but not WSS, which is necessary in the case of WebRTC that uses peer connections over WebSockets to communicate with the PBX. WebRTC, however, requires the use of Secure WebSockets (WSS) to send data, this is solved by creating self-signed certificates using the OpenSSL library for Linux. The PBX works a as central for the client, which they use to register their
ID, IP and other meta data necessary for the establishment of a peer connection between the clients [36].

A simple illustration of how the system setup looks and how the communications between different modules works can be seen in Figure 4.1

### 4.2 Used Libraries and Software

In the development of the application I have used some open source solutions, these libraries and software are presented and shown in this section.

SIP.js is a JavaScript library that helps developers add a full SIP signalling stack to their WebRTC applications. SIP.js is originally a fork of the open project JsSIP and developed as an open source library by OnSip which creates applications for multimedia communication. The JavaScript library is fully SIP compliant according to the company [37].

Freeswitch PBX is a free and open source fully functional communications software for the creation voice and message sessions, including VoIP calls using SIP, it also supports Websockets which are essential to the web-based WebRTC-SIP application. It is licensed under the Mozilla Public License (MPL) and built on a library called libfreeswitch [39].

Node.js is an open-source, cross-platform runtime environment used mainly for creating server-side Web applications. This is what I used to create the HTTPS server that hosts the web-based WebRTC-SIP application [40].

OpenSSL was used to create the self-signed certificates that are used in the Node.js implemented HTTPS server as well as the certificates used in the FreeSwitch PBX to be able to construct a WSS connection [42].

NW.js (previously known as node-webkit) is a platform which enables developers to write native application using web technologies. The platform is built using node.js and chromium and the applications are run in a browser environment but allows for node.js modules and API directly from the DOM (Document Object Model) [41].

### 4.3 WebRTC-SIP Implementation

A fully functioning closed system was constructed, which allowed the web-based WebRTC-SIP implementation to call various open source VoIP SIP phones. The closed system simulates a
4.3. WebRTC-SIP Implementation

Figure 4.2: WebRTC-SIP system blocks and communication protocols.

realistic environment for the tests which in return could be verified throughout all the steps [28].

The system allows for several VoIP phones to connect to the PBX using both WSS for the WebRTC client and UDP for the other open source VoIP phones which was used as a test to confirm that it works with existing VoIP application, which was one of the goals. The system consists of two servers running on a *Ubuntu 14.04 LTS* machine: a Node.js HTTPS web server which provides the static files to the browser and a FreeSwitch PBX acting as the switch which connects clients together.

The actual web-based WebRTC-SIP application is built upon the existing JavaScript API provided by WebRTC in the browser, on top of that a library called SIP.js which acts as a signalling layer was used for the application and in turn replaces the signalling layer used within WebRTC with SIP. This is the key for the application since it makes it possible to communicate with the PBX and VoIP phones [28].

Another application was constructed which uses the same code and the same SIP.js library as the web-based WebRTC-SIP implementation with the difference that it uses a software called Node-Webkit which uses Node.js and a chromium trunk. This implementation allows for web-applications to be run as native desktop applications. The only big difference between the web-based WebRTC-SIP implementation and the Node-Webkit implementation is that makes use of regular Websockets (WS instead of WSS) to connect to the PBX. The reason for creating such an application was to test whether it could be possible to port this particular application directly to an ARM environment.

The complete closed system can be seen in Figure 4.2.

An example of how the web-based WebRTC-SIP client starts and calls a open source SIP based VoIP application is shown below (for details see Section 2.4 for SIP and Section 2.6 for SDP):
Chapter 4. WebRTC and SIP Browser-based Implementation

Figure 4.3: web-based WebRTC-SIP Application running in Chrome in a audio-video call using SIP messages.

*Start
- Fetch static HTML files from Node.js server
- Request audio and video streams from user
- Set up a connection with the FreeSwitch PBX using WSS
- Register client with the PBX

*Make a call
- Make a request for call to a user (VoIP phone also registered)
- Set up peer connection to VoIP peer
- Send INVITE message to peer
- Negotiate media details
- Start session and transfer streams

*End call
- Send BYE message to peer
- Connection closed

The web-based WebRTC-SIP application was tested between two *Ubuntu 14.04 LTS* machines running *Google Chrome* web browser with version *40.0.2564.82* (64-bit) and with two open source SIP VoIP phones: X-Lite (*http://www.counterpath.com/x-lite/*) for Windows and PJSUA (*http://www.pjsip.org/pjsua.htm*) for Linux. The application allows for calls to be done using only audio (between WebRTC-SIP to PJSUA and X-Lite) or with audio and video (between two web-based WebRTC-SIP clients). It also allows for messages to be sent using SIP between the web-based WebRTC-SIP client, see Figure 4.3 for example of a audio-video call using SIP. (see Section 2.4 for details on how the call is set up using SIP).
Chapter 5

Results

This chapter presents the result as two parts: The first part (see Section 5.2) shows a theoretical evaluation of the possibility of a WebRTC-SIP implementation on a ARM A9 processor. The second part (see Section 5.1) presents the results of the tests made in Section 3.2.

5.1 CPU Load Effect on ALSA Audio Drivers

The measurements from the tests in 3.2 show a very small difference between the runs made with a very low load (≤ 2%) and the runs made with a high load (≥ 99%). Since the measurements are done around a write-read loop in the program, the small differences most likely depend on the loop and function calls and not a delay in the run time of the ALSA-kernel drivers.

The tests show that a high CPU load of programs with round robin process priority will not affect the performance of kernel-level drivers using the default settings. It also shows that a VoIP application which might utilise a high CPU load will not affect the sound capture in the ring buffer in the ALSA driver, at least not in a noticeable manner since the maximum difference discovered in the tests was 0.0966ms which is so small it’s redundant in this context since the delay over a network is more prominent [21].

5.2 WebRTC-SIP ARM A9 implementation evaluation

From my research I have found two ways to approach the implementation of a WebRTC-SIP application on ARM architecture. The easiest and fastest way is to utilise the Javascript API which exists within browsers or a webkit application and create a Javascript application. This allows the ARM device to host the necessary files to a browser which the user will use to make calls to SIP phones without straining the ARM device (see 4.3 for example).

The other way is to create a native application using the C++ API coupled with a few modules which complement the package such as audio/video capture as well as a middle-ware translator between the signalling layer of WebRTC and SIP. This approach has more potential since the usage expands from the browsers to a range of embedded devices. The problem with this however is the WebRTC package does not support ARM cross-compilation, and if successful is reported to be unstable.
5.2.1 Web-based WebRTC-SIP Implementation on ARM

This approach is pretty straightforward, since it only requires the application to be hosted on the card as a web service and connect to an external PBX. The benefits from this is that the WebRTC core that is running the VoIP application is run in the users browsers and can be constantly be updated by Google and the need for support is minimised. The drawbacks of this approach is the same as the benefits; there is no way to change the WebRTC stack that is running in the browser without creating a own browser, and therefore loosing the possibility for changing some aspects, for example the noise cancellation software or the codecs.

Since this approach only requires the application to be hosted on the card as a web service and connect to an external PBX all the computation and calls will be performed on the client side in a browser, which reduces the strain on the actual ARM device. The benefits from this is that since WebRTC is not actually running on the ARM device but on the users browsers, there is no need to update code on the device. Instead the WebRTC package which resides within the browser will be constantly updated automatically by the browser developer, hence the need for support is minimised, only changes in the web-service is required.

The drawbacks of this approach is the same as the benefits; there is no way to change the WebRTC stack that is running in the browser without creating an application from a browser trunk which can be complex. This limits therefore the possibility of the application and makes it harder to create a embedded device as a conference phone for example without the need for a client browser connected to the device. It also makes it more difficult to change the behavior of the package, for example changing the noise cancellation software or the codecs. Simple illustration of such an implementation with a computer using a browser and a VoIP SIP phone connected to the same PBX can be seen in Figure 5.1.

5.2.2 Native implementation on ARM

The creation of an native application is a lot more time consuming than the web-based implementation since it requires the creation of two modules within the arm architecture to work. As standard the WebRTC package grabs the video and audio from the browser API namely getUserMedia which is supported by the browsers, this means that a module which can grab the audio and video streams from the card has to be created to allow for the WebRTC package to receive it.

The second module that needs to be created is the middle-ware between WebRTC and SIP which should translate the signalling layer of WebRTC to SIP, the easiest way to do this would
be to have it be a WSS server which receives the requests and then translate using an existing SIP library then forward the information to a PBX or a VoIP phone.

Currently the biggest problem is that the WebRTC package does not officially support cross-compilation to the ARM architecture. There have been some success in this area by some discussion threads, what I have learned from these is that it can work but the performance is unstable and unreliable. There has been some success in compiling an older version of WebRTC from 2014 to ARM which can be seen in Section 2.1.2.

The benefits from this approach is the ability to adjust and customise every aspect of the software, allowing for tweaks in the WebRTC stack such as changing the media engine and allow for another range of codecs and echo cancelling software. It also allows for the implementation to be applied in a large range of applications such as VoIP phones and sound cards. The main drawback of this approach is that it requires more time to get a stable software in the state of WebRTC at the moment. Another drawback is that it will require more from the developer to maintain since updates from Google to the package will be harder to patch than the web-based approach where it happens automatically. An illustration of how a such an implementation could be designed can be seen in Figure 5.2.

Figure 5.2: Illustration of a Native WebRTC-SIP implementation, where the container inside the ARM device represent the modules of the system.
Chapter 6

Conclusions

This chapter contains the conclusions of Chapter 3 (See Section 6.1) and conclusions from the implementation in Chapter 4 (See Section 6.2). The conclusions related too the thesis goal

(a) Evaluate the possibility of a WebRTC-SIP implementation running on ARM A9 processor architecture.

are in detail described in Section 5.2 (See Section 6.3).

6.1 Evaluation of Linux Audio Driver Impact From CPU Load

As shown in the results presented in Section 5.1 it was found that the delays from a high CPU load may have an effect on the latency from the CPU interrupts on the ring-buffer is about 0.0966 ms. With consideration that the measurements might be caused by other factors than actual delay in the drivers (see Section 3.2), we conclude that high CPU loads will not have an significant affect the ALSA audio drivers on a ARM A9 device, thereby answering the question stated in Section 1.2:

(b) How is the kernel level ALSA capture/playback interface affected by high CPU loads on a Cortex ARM A9 processor?

The conclusion is that the kernel level ALSA capture and playback interfaces is not at all, or insignificantly, affected by high CPU loads.

6.2 WebRTC and SIP Browser-based Implementation

As shown in Chapter 4, I have constructed a functioning system for which I have implemented a web-based WebRTC-SIP implementation that works in a closed system and can make VoIP SIP phone calls. The question in Section 1.2,

(c) How can a WebRTC-SIP application be built within a browser environment?

is answered in full in Section 4, The conclusion is that it can be done in two major ways, either using a WebRTC to SIP gateway (see Section 2.1.1) or the way that I proceeded in Section 4 which build on using a library which acts as the signalling layer of WebRTC but utilised SIP messages instead.
6.3 Possibility of a WebRTC-SIP implementation running on ARM A9 processor

The thesis has presented theory, an evaluation of the effects of high CPU load for kernel level ALSA audio drivers as well as a web-based WebRTC-SIP implementation. These individual parts gives us a good idea of how a WebRTC-SIP application could work on a ARM A9 processor. And the answer is that there is ways to be able to develop such an application, with some things in consideration: The WebRTC package may or may not work particular good on the platform and is time consuming and potentially hard to succeed with. A WebRTC to SIP middleware may be constructed using a WebRTC to SIP gateway.

6.4 Discussion

During the project several obstacles have encountered, which has forced the focus of the of the project in different directions.

It was believed that a WebRTC-SIP application running on a ARM processor with Limes audio echo cancellation software should be a reasonable span for the project. This however turned out to be way to time consuming. Also a focus on an ARM native WebRTC-SIP application without the echo cancellation software also got dismissed after some more research, realising that porting WebRTC to ARM would probably take the majority of the time for the project, mainly because the package does not officially support cross-compilation for the platform. And the fact WebRTC is a really big project with many different modules and the lack of detailed documentation was also a factor into why we did not pursue a WebRTC-SIP implementation.

The major obstacle in the project was the shear size of the task, in fact these kinds of systems takes a lot of time and resources to complete.

Instead an effort was put into an evaluation of such an software solutions, this was preformed in two steps, that are described in the thesis. Where the first step was decided to incorporate the targeted platform and evaluate how such an device could be affected when working with audio applications. The results showed that the ARM platform is a fitting candidate for VoIP solutions since the audio drivers did not experience any large effects from stress on the device CPU load.

The second step was decided to construct a WebRTC-SIP client in the browser to gain more experience with VoIP systems in practice and to learn about how a VoIP system is built. During this, the requirements to the solution was figured out, tests with other VoIP applications was used to verify that a WebRTC-SIP application could work on an ARM device.

By performing the two steps we show two approaches to such an ARM based WebRTC-SIP application, which could be used in the future to construct systems like this and open up a new target for web based VoIP solutions – in an embedded environment such as sound cards and VoIP conference phones.

6.5 Future work

When implementing a web based WebRTC-SIP solution, it should be quite easy to implement on a ARM device using a simple web server with HTTPS which should yield an application with full support for calling SIP phones. To do this, some more work has to be done to the current Javascript implementation to support multi-client calls, adding a dial-pad and ability to connect to different PBXs.
The way to proceed from the research of creating a embedded implementation of a WebRTC-SIP client would be to first successfully cross-compile the WebRTC package for ARM. Then create a middle-ware software that translates the WebRTC signals and forwards them (see Section 5.2.2 for more about this).

Another area of improvement is to replace the current echo cancellation software in WebRTC with *Limes audios TrueVoice*. 
Chapter 7

Acknowledgements

First of I want to thank Thomas Hellström my supervisor at Umeå university which has been a great help in the process. I want to thank the people at Limes Audio for all their help and support during the duration of my thesis. Especially I give a big thanks to Emil Lundmark and Magnus Berggren which has been a great support and helping me every step of the way. Thank you Joel Viklund for the support and being a sounding board and helping me to see things from another perspective.
References


REFERENCES


