Synchronization of streamed audio between multiple playback devices over an unmanaged IP network

Christoffer Lauri
`hcj.lauri@gmail.com`
Johan Malmgren
`malmgren.joh@gmail.com`

Department of Electrical and Information Technology
Lund University

Advisor: Jens A. Andersson, Stefan Höst
Examiner: Maria Kihl

October 7, 2015

Printed in Sweden
E-huset, Lund, 2015

Abstract

When designing and implementing a prototype supporting inter-destination media synchronization – synchronized playback between multiple devices receiving the same stream – there are a lot of aspects that need to be considered, especially when working with unmanaged networks. Not only is a proper streaming protocol essential, but also a way to obtain and maintain the synchronization of the clocks of the devices.

The thesis had a few constraints, namely that the server producing the stream should be written for the .NET-platform and that the clients receiving it should be using the media framework GStreamer. This framework provides methods for both achieving synchronization as well as resynchronization. As the provided resynchronization methods introduced distortions in the audio, an alternative method was implemented. This method focused on minimizing the distortions, thus maintaining a smooth playback.

After the prototype had been implemented, it was tested to see how well it performed under the influence of packet loss and delay. The accuracy of the synchronization was also tested under optimal conditions using two different time synchronization protocols. What could be concluded from this was that a good synchronization could be maintained on unloaded networks using the proposed method, but when introducing delay the prototype struggled more. This was mainly due to the usage of the Network Time Protocol (NTP), which is known to perform badly on networks with asymmetric paths.

Popular Science Article

Synchronized playback between multiple speakers over any network

When working with synchronized playback it is not enough just obtaining it – it also needs to be maintained. Implementing a prototype thus involves many parts ranging from choosing a proper streaming protocol, to handling glitch free resynchronization of audio.

Synchronization between multiple speakers has a wide area of application, ranging from home entertainment solutions to big malls where announcements should appear synchronized over the entire perimeter.

In order to achieve this, two main parts are involved: the streaming of the audio, and the actual synchronization. The streaming itself poses problems mostly since the prototype should not only work on dedicated networks, but rather on all kinds, such as the Internet. As the information over these networks are transmitted in packets, and the path from source to destination crosses many sub networks, the packets may be delayed or even lost. This may create an audible distortion in the playback.

The next part is the synchronization. This is most easily achieved by putting a time on each packet stating when in the future it should be played out. If then all receivers play it back at the specified time, synchronization is achieved. This however requires that all the receivers share the idea of when a specific time is – the clocks at all the receivers must be synchronized. By using existing software and hardware solutions, such as the Network Time Protocol (NTP) or the Precision Time Protocol (PTP), this can be accomplished. The accuracy of the synchronization is therefore partly dependent on how well these solutions work. Another valid aspect is how accurate the synchronization must be for the sound to be perceived as synchronized by humans. This is usually in the range of a few tens of milliseconds to five milliseconds depending on the sound.

When a global time has been distributed to all receivers, matters get more complicated as there is more than one clock to consider at each receiver. Apart from the previously mentioned clock, now called the 'system clock', there is also an audio clock, which is a hardware clock positioned on the sound card. This audio clock decides the rate at which media is played out. Altering the system clock to synchronize it to a common time is one thing, but altering the audio clock while media is being played will inevitably mean a jump in the playback, and thus a distortion. Although an initial synchronization can be achieved, the two clocks will over time tick in slightly different pace, thus drifting away from each other. This creates a need for the audio clock to continuously correct itself to follow the system clock.

In the media framework GStreamer, used for handling the media at the receivers, two alternatives to solve the correction problem were available. Quick evaluations of these two methods however showed that either audible glitches or 'oscillations' occurred in the sound, when the clocks were corrected. A new method, which basically combines the two existing, was therefore implemented. With this method the audio clock is continuously corrected, but in a smaller and less aggressive way. Listening tests revealed much smaller, often not audible, distortions, while the synchronization performance was at par with the existing methods.

More thorough testing showed that the synchronization over networks with light traffic was in the microsecond-range, thus far below the threshold of what will appear as synchronized. During worse conditions – simulated hostile environments – the synchronization quickly reached unacceptable levels though. This was due to the previously mentioned NTP, and not the implemented method on the other hand.

Acknowledgements

Firstly, we would like to thank our supervisor Jens Andersson at the Department of Electrical and Information Technology, LTH, for always helping and pointing us in the right direction, and not least for always having a nice anecdote or two ready to tell. We also want to send our thanks to our main supervisor, Stefan Höst, and our examiner, Maria Kihl, both also belonging to the Department of Electrical and Information Technology at LTH, for comments and feedback that have greatly improved the thesis.

The thesis project itself has been carried out at Axis Communications, where we would like to thank our two supervisors Olle Blomgren and Anders Persson, for always being up for a discussion or two on what the next step should be, over a nice cup of coffee. Yet another vital person that we could not have finished the thesis without, is of course Anders Hansson – our very own personal supplier of an endless amount of network cables. A great thank you from us to the rest of the New Business team as well, where people always have been eager to help and advise. Also, a great thank you to all other great Axis people we have had the pleasure to meet and discuss with.

Finally we also want to thank our families and friends, who have supported and been there for us during the whole project, believing in us, and cheering us up when needed.

Table of Contents

List of Figures

List of Tables

Abbreviations

AoIP	Audio over IP
API	Application Programming Interface
BC	Boundary Clock (PTP)
DCCP	Datagram Congestion Control Protocol
IETF	Internet Engineering Task Force
LAN	Local Area Network
MU	Media Units
NAT	Network Address Translation
NTP	Network Time Protocol
OC	Ordinary Clock (PTP)
PLC	Packet Loss Concealment
PTP	Precision Time Protocol
RR	Receiver Report (RTCP)
RTP	Real-time Transport Protocol
RTCP	Real-Time Control Protocol
RTT	Round-Trip Time
SCTP	Stream Control Transport Protocol
SR	Sender Report (RTCP)
TC	Transparent Clock (PTP)
TCP	Transmission Control Protocol
TLCC	TCP-like Congestion Control
TFRC	TCP-friendly Rate Control
UDP	User Datagram Protocol
VoIP	Voice over IP
WAN	Wide Area Network

Chapter 1

Introduction

Streaming media over Internet Protocol-based (IP) networks has been performed since the 1990s [1], when the widespread usage of the Internet began. Ever since new streaming services have evolved and been improved, and today products such as Spotify, used for music streaming, and Netflix, used for video streaming, can be taken for granted by many people. This development has of course been possible thanks to the extension of Internet connections, and the continuous increase in Internet speeds. Davidsson does for example show in [2] a 127% increase in the average Internet download speeds in Sweden between 2008 and 2013.

While the capacity and availability increases, the packet based approach itself for exchanging data used on the Internet and on a majority of local area networks (LAN), does nonetheless pose a number of problems. These problems could for example be loss of packets or delay which in turn may lead to a interruption of the stream. Moreover, when streaming media synchronized to multiple clients – simultaneous playback across all the receving devices – packet loss and especially delay will pose an even greater problem. The synchronization aspect also adds even further challenges, such as how to actually achieve the synchronization, as well as how to maintain it. Furthermore, maintaining the synchronization will be the hardest part focusing on high quality audio playback, as distortions easily are introduced when performing resynchronization. One commercial solution trying to solve these problems is Sonos [3].

The aim of this thesis is to look at what protocols may be used for media streaming, but also what suitable ways there are to synchronize playback between multiple embedded devices given the constraints found under Section 1.2.1. An illustrative overview of this server-client relationship can be found in Figure 1.1. The thesis also aims to look at different resynchronization methods, focusing on high quality audio playback. This means that audible glitches in the stream should be kept at a minimum.

All these investigations will then result in the implementation of a prototype, which in turn will be evaluated with respect to the aspects stated under Section 1.2. Finally it can be noted that the primary goal of the thesis is to look at the synchronization and resynchronization part. This means that although an analysis of protocols is made, this will not be as exhaustive as the one made for synchronization and resynchronization.

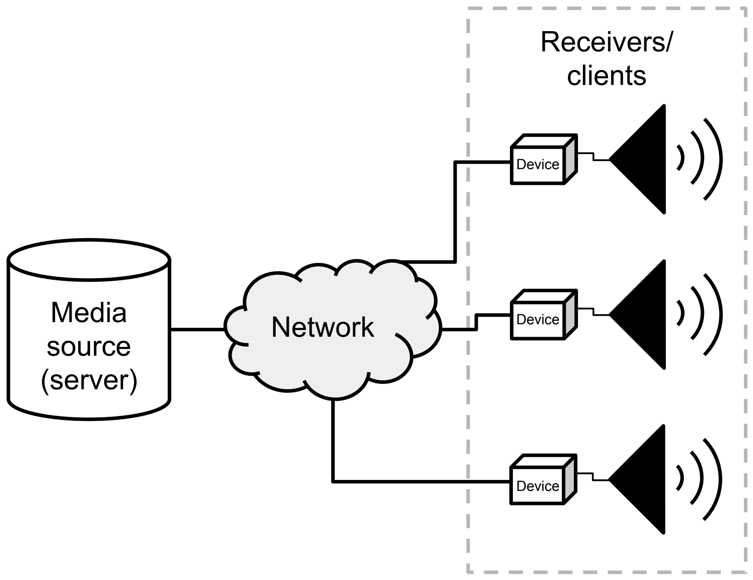


Figure 1.1: A simple overview of the structure of the system the thesis intends to investigate and prototype.

1.1 Background

1.1.1 Audio streaming

Streaming does in contrast to simply downloading a file allow playback to begin before the whole file has been fetched. This removes the waiting time to a great extent for the user, as only a part of the file needs to be buffered before the first part of the content can be enjoyed. One example of this is as previously mentioned Spotify, which is a music streaming service where the user is expected to be able to listen to the audio clip shortly after pressing play. Another example is a telephone call made over IP using for example Skype, so-called Voice over IP (VoIP), although this case involves live streaming, and not the streaming of stored content, as in the previous example. In the VoIP-case the audio quality can be seen as being of lower priority – as the primary focus is voice, the signal can be more compressed. Latency, delay in processing and transmitting the audio, is of utter importance to keep low though, in order to be able to have a satisfying conversation. The case with streaming music is very similar, yet slightly different – while low latency is good to have here, the audio quality is especially crucial, in contrast to VoIP. The term for this focus on music streaming, instead of voice, is Audio over IP (AoIP).

As AoIP focuses on music streaming, being able to provide uninterrupted streaming to the end user is one aspect to consider in order to maintain an acceptable audio quality. Other aspects may be the actual encoding and decoding process to minimize the degradation of the audio source. This in contrast to when working with voice, VoIP, where interruptions and a more aggressive compression probably would be more acceptable, as long as the spoken words still are audible.

The thesis will consider both AoIP and VoIP, although VoIP only will be a smaller part of it. This is because the prototype in addition to music also should handle messages, why VoIP is relevant, but the main focus will nonetheless be on the music streaming part and thus AoIP.

1.1.2 Streaming protocols

In the process of streaming media a suitable protocol has to be selected first of all. There are two transport protocols most commonly used on the Internet today – the Transmission Control Protocol (TCP) [4] and the User Datagram Protocol (UDP) [5]. TCP is a reliable protocol in delivering the data, by for example numbering the packets in order for them to be processed in the correct order as well as retransmitting lost ones. UDP is, in contrast to TCP, unreliable in the sense that it does not support numbering of packets and simply provides a best-effort approach – given a packet is lost, it is simply ignored [6]. TCP is therefore suitable in applications such as everyday web browsing, as some delay due to retransmission is fully acceptable and probably unnoticeable, while a corrupted file due to missing packets would not be tolerable. The same is true for streaming of stored content, where the video or audio quality is of higher importance than waiting times resulting from buffering, such as when using Spotify.

Focusing on live streaming instead, TCP may become problematic in this case, as the media content should be available for consumption by the receivers shortly after it has been produced. Otherwise it can be debated whether it is a livestream. As TCP requires buffering to make sure that lost packets will have the chance to get retransmitted before an interruption of the stream occurs, delays will be imposed when using TCP.

Another approach is therefore to use the barebone UDP and accept its nonexistent congestion control and reliability. This can then to some extent be compensated for by introducing higher-level protocols which in turn provides more functionality or possibilities – and primarily more flexibility. One such protocol could be the Real-Time Transport Protocol (RTP) [7], which introduces for example sequence numbers as well as detection of when a packet has been lost. How the packet loss should be handled is then decided by protocols higher up in the stack, or the user application. As the protocol can be used in conjunction with both UDP and TCP, it complements UDP by the introduction of sequence numbers, while slightly overlapping with TCP in this particular case. It however adds some media specific extensions that are not available in either of the two protocols.

Although TCP and UDP are common choices, they are however not the only available alternatives – the Datagram Congestion Control Protocol (DCCP) [8] is also a protocol on the same level, primarily targeted at real-time media. It provides mechanisms for congestion control, while still being unreliable like UDP when it comes to the dataflow itself.

1.1.3 Synchronization

In the process of streaming media, synchronization is always a vital part. First of all the transported data should be played in the correct order to achieve a well-functioning stream, which also is a kind of synchronization. But there may also be multiple streams that must be played simultaneously with a certain offset in order to deliver a satisfactory user experience. A typical case is when streaming a movie, which may contain one video stream and one stream for the audio, which of course need to be synchronized.

In order to differentiate between cases of synchronization, such as the two previously mentioned examples, synchronization can be categorized into three types, as explained by Boronat et al. in [9]:

- intra-stream synchronization
- inter-stream synchronization
- inter-destination, or group, synchronization

Intra-stream synchronization concerns the synchronization within a media stream, that is, the temporal relations between the media units (MU) in it. What this means is that there simply should be no gaps or overlaps of the frames.

Inter-stream in turn refers to the synchronization of different, but related, media streams. An example of this is lip synchronization, where the video and the audio need to be synchronized in order to play a voice in sync with the lip movements.

The third, inter-destination, involves different receivers where the media needs to be played out simultaneously and at the same point in time at all the receivers [9]. Inter-destination synchronization is important in many situations. For example in the case of streaming audio to speakers, where a few seconds delay at one speaker would completely ruin the music experience if the speakers are within audible distance from each other.

As this thesis concerns audio streaming to multiple devices, intra-stream and inter-destination synchronization are the types that will be concerned.

When synchronization has been achieved, the next step is to maintain this synchronization. As clocks in the hardware and software are not perfect, these may tick at slightly different pace, and thus result in the synchronization getting worse over time – although an initial, accurate, synchronization was achieved.

This problem of maintaining the accuracy of the synchronization over time poses follow-up problems, such as how a synchronized playback should be restored, while minimizing audible artifacts resulting from the process. To solve this issue there are different possible approaches, some more naive and others more advanced. The more advanced solutions may try to achieve a better result through algorithms, gradually correcting the clock, whereas the naive solutions simply will set the clock to the expected value, and ignore the thereby introduced artifacts.

1.2 Problems and Method

As much research has been carried out regarding audio streaming and synchronization, literature and other current work was first consulted and evaluated in order to establish to what extent this previous research could be used in the intended context of the thesis. Interviews with people who have knowledge in the area, and have been involved in similar systems, were also held.

When a basic insight had been gained in the subject, the implementation of a prototype started in parallel with carrying out more research when it was needed. The focus when implementing the prototype was how to handle the resynchronization that is needed due to the effect of clock drift. Since third-party software was used as explained in Section 1.2.1, a large part of the thesis has also thus been learning and understanding the required software and not solely reading about media streaming and synchronization techniques in general.

Finally when a working prototype had been created, tests and evaluation of advantages as well as disadvantages of the choices made were carried out. The focus of this evaluation was how well the audio was synchronized, but also how well performance metrics like packet loss and delay were handled.

1.2.1 Scope

This thesis aims to first examine and discuss different protocols and synchronization concepts. This in order to create a prototype which should achieve synchronized playback to multiple devices, where the audio should have a minimal amount of distortions. To do so, the thesis will evaluate the currently available options for streaming, synchronization, and resynchronization, using the required software.

Furthermore, the focus of the thesis is not to achieve the best possible synchronization. Instead it will focus on providing a smooth playback, as long as the audio is perceived as synchronized by the listener.

When doing the examination and the final choice of protocols and techniques a few limitations are given however, which need to be considered. These are:

- The receiving clients are embedded devices[1], and thus have limited hardware.
- The receiving clients will use GStreamer [10] to playback the received input.
- The sending server will be implemented using the .NET platform.
- The network is IP based, and no assumption about control of the network can be made, for example quality of service settings cannot be assumed.
- The audio codec used should be Opus [25].

Moreover there are other complexities involved when synchronizing audio which will not be covered in this thesis. One such complexity is echo effects which may be present due to the listener position or movement between the different speakers and not because of that the speakers are not in sync. Another complexity it will

[1]Raspberry Pi 2 Model B with HiFiBerry DAC+.

not focus on is building a complete audio system where it should be possible to setup stereo speakers or a home theatre. Instead the same stream will be sent to all receivers, and thus speakers.

1.3 Thesis structure

The thesis begins with a presentation of the background and problem description. This is then followed by a chapter with the theory for the thesis, as well as current work done in the field. As GStreamer will be used, a short chapter introducing this framework follows. The next chapter evaluates the theory presented in the previous chapter, and discusses and explains the final design choices. An implementation chapter then follows, explaining the prototype in more detail. This prototype is then evaluated in the next chapter, which contains both test cases and the corresponding results. A chapter briefly discussing the theory and implementation then follows. Finally a chapter with conclusions and suggestions for future work is presented.

Chapter 2

Relevant Work

This chapter will present a theoretical background of streaming and synchronization, focusing on parts relevant for the thesis. Also, excerpts from current work within these fields will be covered.

2.1 Streaming

In the upcoming sections issues and protocols related to the streaming of media will be described.

2.1.1 Challenges in streaming media

Network address translation and firewalls

When working with unmanaged IP networks the concept of network address translation (NAT) [11] needs to be explained to understand what problems that may be introduced by this process to transport layer protocols.

A device supporting the process of network address translation, a NAT device, can act as a middle-man between different IP address spaces. This thereby allows to have a private network on one side, and connect this to a public network on the other one. The private network can then use IP addresses from any of the three reserved pools of addresses for private networks, such as 192.168.0.0 - 192.168.255.255. These addresses can be freely used, as they are only unique within a private network, not globally. This enables the mapping of a set of IP addresses, globally unique and visible on the Internet, to a set of addresses that are only visible on the private network. In order to still let the devices on the private network access the public network, such as the Internet, the NAT device replaces the source IP address in the IP packets. What this simply means is that the private source address will be replaced with the global IP address, before the packet is forwarded to the next router on the public network. For transport layer protocols containing source ports, these might also be altered to a port that is unused on the NAT device. This process is illustrated in Figure 2.1.

All the mentioned packet changes, as well as destination IP and destination port, are finally stored in a translation table. If a reply is received, the NAT device can simply look up the IP and port combination in the translation table. By then replacing the destination IP address and destination port with the values from the

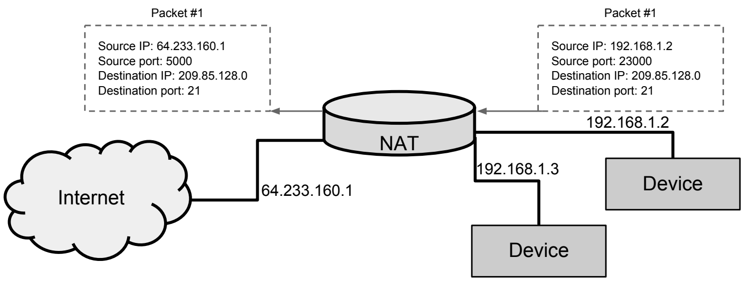


Figure 2.1: An illustration of network address translation (NAT).

table, the packet can reach the correct device on the private network. A received packet that does not match any entry in the table can simply be discarded.

An issue with NAT devices is that they must actively support the chosen transport layer protocol, as they need to change header values, such as ports and recomputed checksums. New transport layer protocols therefore face problems, as the support for them needs to be implemented in the NAT devices before successful forwarding can occur.

2.1.2 Protocols

When deciding upon which protocols to be considered, it is obvious that the two most widespread transport protocols on the Internet should be candidates. These are, as commonly known, UDP [5] and TCP [4]. The reason these protocols cannot be neglected, is that a focus of the thesis has been to implement a prototype with as much widespread support as possible, thus working in most environments.

Apart from these two protocols, other alternatives exist as well, but few of them have gained any larger recognition, as it may be hard to enter a field where already well-deployed alternatives exist. In the end the spread of a protocol depends to a great extent on its ease of deployment, as argued by Stewart and Amer in [12], which might also be the greatest barrier for protocols such as the Stream Control Transmission Protocol (SCTP) [13]. It is true that these types of more unknown protocols in comparison to TCP and UDP may introduce important advantages, since both TCP and UDP have their shortcomings, but as only a few alternatives may be investigated due to the scope of the thesis, SCTP will not be further discussed.

An alternative of a lesser known protocol that will be investigated however is the Datagram Congestion Control Protocol (DCCP). This protocol specifically targets real-time media streaming and thus, although it is not very widespread, is an interesting candidate to consider. In the upcoming sections there will therefore be an in-depth introduction to DCCP, followed by other related aspects to the challenges and alternatives when streaming media.

2.1.3 DCCP

An alternative protocol to UDP and TCP is the Datagram Congestion Control Protocol (DCCP) [8]. This protocol aims to provide congestion control while maintaining a focus on data with timing constraints, such as streaming media. Apart from congestion control it also provides reliable connection setup and teardown, mechanisms to acknowledge packets, and support for reliably negotiating connection options. At the same time it has an unreliable datagram flow, such as no guaranteed in-order delivery or retransmission of packets. DCCP is therefore a way to relieve upper-layer protocols and applications of the need to implement congestion control, and also support multiple congestion-mechanisms, which is negotiated during the connection setup. The mechanisms specified by the DCCP RFC [8] for use as congestion control are the TCP-like Congestion Control (TLCC) [15] and the TCP-friendly Rate Control (TFRC) [16]. They are also known as CCID2 and CCID3, where the latter one tries to provide a solution aimed at media streaming.

TCP-like Congestion Control (TLCC)

TLCC is, as the name implies, much inspired of the congestion control mechanisms used in TCP. What this means is that abrupt changes in the data rate is to be expected, and thus an application must be able to handle this. This means that it may not be optimal in a streaming application, where sudden changes in the rate may lead to choppy playback. On the other hand it better allows the sender to utilize the available bandwidth. It is therefore suitable for applications that are not sensitive to – or can buffer enough data to not be affected by – sudden major changes in the data rates. These abrupt changes are partly due to the handling of the congestion window, which specifies how many data packets that are allowed on the network at the same time, as this number is halved if there is a congestion event.

TCP-friendly Rate Control (TFRC)

The alternative mechanism, TRFC, focuses unlike TLCC on a smooth data sending rate, by responding slower to events and changes on the connection. The maximum data sending rate is computed by calculating a loss event rate. This loss event rate is a ratio between the number of loss events and the total number of packets transmitted. In addition to the loss event rate, the maximum data sending rate also takes into consideration the average packet size and an estimate of the round-trip time. TRFC is thereby more suitable for applications that can accept not to use all the available bandwidth, but still needs a smooth data rate. A reason for this need might be that the application is unable to have large enough buffers, or is greatly affected by abrupt changes in the rate. This alternative may thus be better used in a streaming application, but it all depends on whether buffering may be allowed.

Evaluations

From the experiments performed by Chowdhury et al. in [17] the authors could conclude that CCID3 – TRFC – did maintain a smooth rate, with few abrupt changes in the bit rate. It also showed far smoother rates compared to CCID2, TLCC. TLCC did on the other hand present a high throughput, one close to that of TCP, while TRFC – as expected in order to minimize changes – did not utilize the bandwidth equally good. Also Azad et al. in [18] observed that the throughput of CCID3 was lower than that of CCID2, although not much lower. CCID3 however showed considerably higher jitter, at times twice as high as CCID2, while it at the same time had less than half the delay. The reason for this is that CCID2 had problems with dropped packets during congestion conditions.

Packet types and header

Looking away from the congestion control mechanisms, the DCCP specification also states several packet types to be used during the communication. A few examples are the DCCP-Data to send data from upper layers, DCCP-Ack to send only acknowledgements, and DCCP-Request to begin a connection. In contrast to TCP DCCP specifies more packet types instead of utilizing flags. There are for example a DCCP-Sync-type to resynchronize sequence numbers, a DCCP-Close to close the connection, a DCCP-Reset to terminate the connection, and so on. This does for example avoid the problem of an unexpected combination of flags.

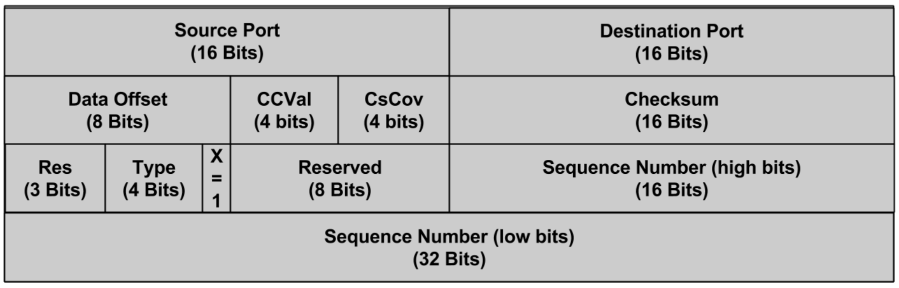


Figure 2.2: The DCCP header with the X-field set to one [8].

In Figure 2.2 the generic DCCP header can be seen, which is followed by the upper-layer data in a packet. As DCCP is a transport layer protocol it defines source and destination ports just like UDP and TCP do. In order to be able to determine packet loss and handle acknowledgements the header therefore contains fields for sequence numbering. Depending on the value in the X-field either 24-bit or 48-bit long sequence numbers are used, where the 48-bit alternative adds further protection against attacks such as blindly injecting reset packets. Using 48 bits also decreases the risk of sequence numbers wrapping around for high-rate connections. As the value in the X-field in Figure 2.2 is one, an extra field of 32 bits (low bits) is added to the header allowing the longer sequence numbers. Given that the X-field would be zero instead, the 'Reserved'-field would be removed, and be used for the sequence number instead as can be seen in Figure 2.3.

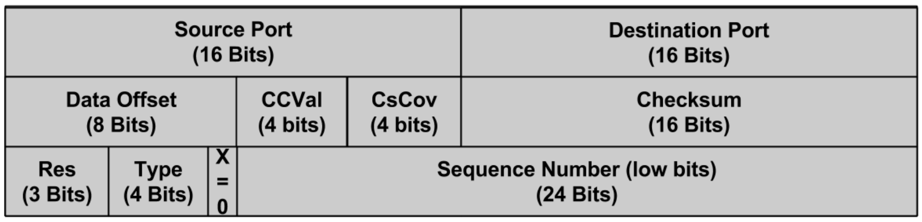


Figure 2.3: The DCCP header with the X-field set to zero [8].

Issues

As discussed in Section 2.1.1 the NAT process poses a problem for non-widespread transport layer protocols, which is also the case for DCCP. Although a specification for NAT behaviour for DCCP connections has been produced [19], the support for DCCP continues to be a problem network-wise [20].

The issue is, as previously mentioned, that the DCCP header contains a source port as well as a destination port. In addition to this, the header includes a checksum, which takes into account a pseudoheader containing header information from the IP layer. All these parts may need to be changed by a NAT device, not least the pseudoheader, as this might contain IP addresses which most likely will be altered from the private to the public address space. This in turn will invalidate the DCCP checksum value, which is why DCCP needs to be actively supported by the NAT device in order for the packet to not be dropped along the way. For UDP and TCP packets to correctly pass through a NAT device these need to be supported by the device as well. But as TCP and UDP are the most common protocols on the Internet they also face a considerably higher support in NAT devices, compared to DCCP.

Due to the common support of UDP in NAT devices, a workaround to these NAT issues has been introduced in a RFC, which tries to address the problem by encapsulating DCCP packets in UDP packets [21]. While this allows NAT traversal it introduces some overhead, where source and destination port fields will occur twice in the packet, as well as a length and checksum field. As an UDP header is eight bytes long, this is added to each DCCP packet sent, and whether this is an acceptable overhead must be evaluated with respect to the advantages provided by DCCP.

While the NAT issue can be overcome by using the UDP approach, another issue is the support in operating systems [20]. While the Linux-kernel has provided DCCP support since version 2.6.14 [22], there is no support implemented in any of the available versions of Microsoft Windows-operating systems at the time of the writing of this thesis, to the best knowledge of the authors.

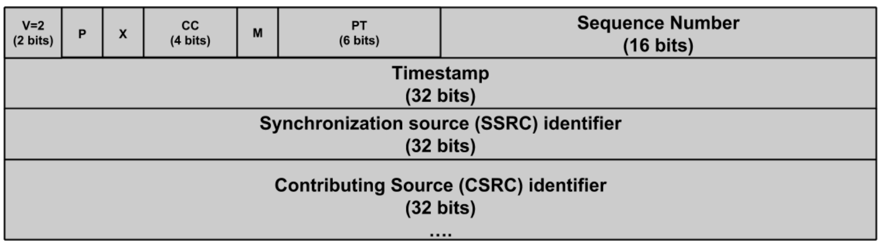
2.1.4 The RTP-family

To address the use cases of real-time media streaming the Real-Time Transport Protocol (RTP) and corresponding helper protocols were introduced and specified in 1996 [26] by the Audio-Video Transport Working Group of the Internet Engineering Task Force (IETF). A revision of the standard was published in 2003 [7].

RTP is unlike UDP, TCP, and DCCP, not a transport layer protocol. Instead it is used in conjunction with these protocols in order to add useful features when streaming media. It operates one level higher up, and thus utilizes transport layer protocols to send and receive data, where UDP is a common choice by applications. It can work on top of TCP as well, and the corresponding advantages and disadvantages is discussed in Section 4.1.

The RTP-family consists primarily of two parts, working side-by-side: RTP, carrying application data, and the Real-Time Control Protocol (RTCP), providing statistics and control information, that is, focusing on the quality of the transmission.

RTP



V=2 (2 bits)
P
X
CC (4 bits)
M
PT (6 bits)
Sequence Number (16 bits)
Timestamp (32 bits)
Synchronization source (SSRC) identifier (32 bits)
Contributing Source (CSRC) identifier (32 bits)
....

Figure 2.4: The RTP header [7].

In Figure 2.4 the header for RTP packets can be seen. It reveals that RTP provides support for both sequence numbering, timestamping, as well as specifying the kind of application data it is carrying. To start with, the sequence number is a randomly initialized number, which is to be incremented by one for each packet. This allows the receiver to both ensure correct order of the received packets, as well as detect packet loss. The timestamp in turn is related to the media being carried, that is, the number will increment depending on the sampling of the media. This field does thus correspond to how much media the packet is carrying. The timestamp is then used to maintain the correct temporal relation between frames, that is, intra-stream synchronization. What it cannot be used for is synchronizing multiple streams against each other, inter-stream synchronization. The reason is that the timestamps of different streams often have a random initial offset as well as increment with different values, if they contain media of differing types. To achieve inter-stream synchronization RTCP also needs to be involved, which will be discussed in the next section. Finally the payload type field (PT) tells the receiver what type of media the packet is carrying, by either specifying a default profile from for example [27], or a dynamic one outside of RTP.

RTCP

RTCP is, as stated earlier, part of RTP sessions in order to distribute feedback about the session back to the sender. It may also provide the receivers with transmission statistics and information for synchronizing streams. To achieve this each RTP stream has a corresponding RTCP stream, which defines five different types of messages: sender reports (SR), receiver reports (RR), source descriptions (SDES), end of participation (BYE), and application-specific messages (APP). In addition to the five types RTCP allows for extensions, where RFC3611 [28] defines a standardized Extended Report (XR) type. In the use cases relevant for this thesis, the RTCP type of interest is the sender report.

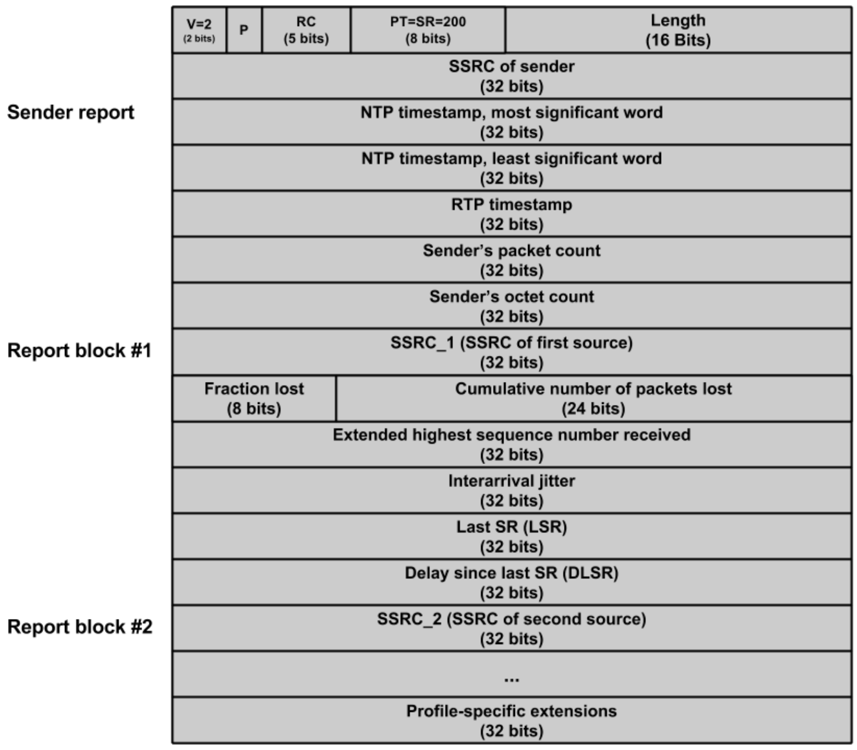

V=2 (2 bits)
P
RC (5 bits)
PT=SR=200 (8 bits)
Length (16 Bits)
SSRC of sender (32 bits)
Sender report
NTP timestamp, most significant word (32 bits)
NTP timestamp, least significant word (32 bits)
RTP timestamp (32 bits)
Sender's packet count (32 bits)
Sender's octet count (32 bits)
Report block #1
SSRC_1 (SSRC of first source) (32 bits)
Fraction lost (8 bits)
Cumulative number of packets lost (24 bits)
Extended highest sequence number received (32 bits)
Interarrival jitter (32 bits)
Last SR (LSR) (32 bits)
Delay since last SR (DLSR) (32 bits)
Report block #2
SSRC_2 (SSRC of second source) (32 bits)
...
Profile-specific extensions (32 bits)

Figure 2.5: The RTCP Sender Report [7].

In Figure 2.5 the packet format for sender reports can be seen. The most interesting fields are the NTP timestamp, RTP timestamp, and the report block(s). The NTP timestamp represents 'wall clock time', absolute date and time, in the same timestamp format as the Network Time Protocol (NTP). Worth noting however is that the way of determining the absolute time is up to the sender, and does not need to be a result of having the system clock being synchronized by this protocol.

Connected to the NTP timestamp is also the RTP timestamp, which in turn is directly connected to the RTP timestamps in the RTP data packets. This means that they are in the same units as well as have the same initial offset. The RTP timestamp is not necessarily the same as in any nearby RTP data packet, as it is just calculated from the NTP timestamp. The idea is to establish a clear relationship between the 'real time', and the 'media time'. As this 'real time' will be shared among all the RTP streams from the same sender, a receiver can use the two timestamps to synchronize multiple streams against each other. Therefore, inter-stream synchronization – such as lip synchronization – can be achieved. Finally each report block corresponds to a synchronization source, which simply is a RTP packet sending source. The report blocks contain reception statistics, such as an estimate of the fraction of lost packets from the specific source.

2.2 Introduction to synchronization

As explained in the background synchronization can be divided into three different types, and the synchronization part of this thesis will focus on inter-destination synchronization. Although intra-stream synchronization is vital for the playback to work, this is handled by the media software, and will thus not be considered in this section.

In order to achieve the inter-destination synchronization there are several aspects that need to be taken into consideration, as explained in the upcoming sections.

2.2.1 Thresholds for perceived synchronization

The first question might be what actually is required for two sounds to be perceived as one – that is, appear synchronized. Basically two separate sound sources would appear as synchronized, that is the listener cannot differentiate the sources, if the difference in actual playout time between the playback devices is low enough. In [35] Wallach et al. examined sound localization by the human ears, investigating when sounds appear as two distinct sources and when they seem fused into only one, single, sound. They concluded that in order for two short click sounds to appear as one sound the difference in time between them could not exceed approximately five milliseconds. In the case of more complex sounds, for example orchestral music, the limit is much higher, maybe as high as 40 milliseconds. When these limits are exceeded echo effects might be perceived, or the sounds may simply appear as two distinct sounds.

These numbers are of course approximations, as there are many variables affecting the perceived synchronization. One example of such a variable is whether the output device is a pair of expensive headphones, or a cheap speaker. Another variable may be the room environment, where acoustics of the room itself may introduce echo effects which also may affect the perceived synchronization.

2.2.2 Achieving synchronization

An approach to actually achieve synchronization would be to ensure that each audio sample is played back at the same point in time, according to some shared, global clock across the playback devices. Therefore, if each audio packet is timestamped with a time somewhere in the future, and the clock at each playback device is synchronized with the clocks at all the other playback devices, synchronized playback can be achieved. This clock synchronization can be accomplished by using some protocol to distribute a global clock, as will be elaborated in the upcoming sections. However, as also will be discussed, it is not just a matter of once synchronizing the clocks, but there is also a need of maintaining a synchronized audio experience over time.

Finally it can be said that there are many parameters affecting synchronization, such as network load, device load, and temperature variations. These all play an important role when working with synchronization as they may greatly affect the result. Not all of them, such as temperature variations, will be considered on the other hand due to the scope of this thesis.

2.2.3 Timestamping

As previously discussed timestamping plays[1] an important role in the process of synchronizing. In order to achieve as high precision as possible for these timestamps, they should be computed and written to the packets as close to the hardware as possible. This is much due to the fact that packets may be delayed in queues or processing times in routers, switches and alike. To compensate for these software-induced delays timestamping of the packets need to occur on a hardware level, which results in more precise calculations. Software-induced inaccuracies may generally occur in the whole chain from recording the media, to sending it to the receiver, and finally playing it out. However, this will not be the culprit of not achieving an accurate synchronization since many other factors will play an even greater part, but it should be noted that this also could be a potential problem working with synchronization.

2.3 Time synchronization protocols

In order to synchronize the time over multiple devices, it is important to know that every computer has a clock which is used to tell what the time is. Depending on the computer a real-time – hardware – clock may be available, but as the Raspberry Pi:s used in this thesis have none, they therefore use a software approach instead.

During runtime a real-time clock does not need to be involved, instead the 'system clock' takes over, which is only software and increments based on timer interrupts from the hardware. These timer interrupts in turn are often a result of oscillating crystals, which can differ in quality and thus accuracy. A way to continuously adjust the system time to a global time is therefore needed, in order to achieve synchronization. Two such examples of protocols are the Network Time Protocol (NTP) and the Precision Time Protocol (PTP).

When discussing the need for clock synchronization, it is worth noting that how well the receiver clocks conforms to the 'real' time is of less importance than the spread of time differences. What this means is that there are no extra requirements

[1]No pun intended.

on how correct the clocks are with respect to what the time actually is, as it is the difference between the clocks on the receivers that define the level of achieved synchronization.

2.3.1 NTP

NTP has iterated a number of versions since its introduction in the 1980s, and it has now reached version number 4 with a corresponding RFC [37]. The protocol consists of both specifications of how to exchange time information, as well as algorithms for how to adjust clocks and choose the time that probably is the most accurate.

How it works

The basic structure is a master-slave architecture, involving different so-called 'stratums'. In the stratum hierarchy a lower number means a more accurate server or clock. A stratum 1 server is for example considered to be a very good NTP server. These servers are connected to so called 'reference clocks', also known as stratum 0-devices, where the best of them often are atomic clocks. Moving down the hierarchy then means less trusted clocks and servers, and thus higher stratum numbers. A computer or server that thus synchronizes its time against a server of stratum n will itself have a stratum of $n+1$. This number therefore indicates how far from a reference clock a computer is distanced, where the upper limit is 15. A stratum of 16 indicates an unsynchronized computer. The stratum numbering also helps in avoiding cyclic synchronization dependencies.

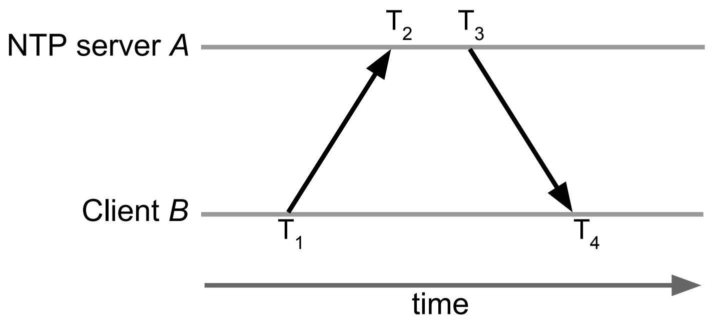


Figure 2.6: The exchange of packets between client and server in NTP.

The process of exchanging time information in order to synchronize a client B's clock to an NTP server A involves four timestamps, as well as four packets. The case of unicast-mode can be seen in Figure 2.6, where B initiates the process by sending a packet to A and storing the sending origin timestamp T_1. A receives the packet at time T_2, and sends a response back to B at time T_3. This packet then arrives at B at time T_4. As T_1, T_2, and T_3 are all part of the packets, and B

itself knows T_4, the client now have all four timestamps that it needs to compute the round-trip time δ and the relative clock offset θ to A:

$$\delta = (T_4 - T_1) - (T_3 - T_2) \tag{2.1}$$

$$\theta = \frac{(T_2 - T_1) + (T_3 - T_4)}{2} \tag{2.2}$$

These values are then evaluated through certain algorithms and filters. Several exchanges of packets are required before the resulting time is trusted though, and all received packets must have successfully satisfied the protocol specification. This number of correct exchanges needed before a server is trusted is about five times [38]. When one, or multiple, servers are trusted, the client's clock begins being adjusted according to the round-trip time and clock offset. This is either done in a step or gradually, depending on the magnitude of the offset. The frequency at which these packet exchanges are done can be configured, but usually starts at about one minute. This then increases to about 20 minutes, in order to minimize network overhead.

An issue in the determination of clock offset is asymmetric delay times, that is, the delay is different for the forward and backward travelling times. The reason is that these cannot be differentiated from just the offset. If the conditions are really bad, such as a landline connection in one direction and a satellite in the other, the errors could reach a magnitude of 100 ms or more. These errors can only be avoided by on beforehand having knowledge about the network paths. In contrast to these errors a possible accuracy of 100 μs on lightly loaded Ethernet-networks are achievable, and in the tens of milliseconds over intercontinental Internet paths [39].

An aspect that also affects the accuracy of NTP is the fact that it is purely software based. This means that in the process of timestamping delay caused by for example the operating system or other processes are present. Improvements can however be made, as discussed in [36], where Microsoft took an approach to design an IP sound system using NTP as the time synchronization protocol. These improvements resulted in a 50 μs granularity.

Availability

NTP is widely available and supported by systems, and the NTP project offers a complete implementation for Unix-like systems [40]. Microsoft also ships a NTP service with later versions of their Windows operating system.

2.3.2 PTP

PTP was first specified as a standard in 2002, but was revised in 2008 in IEEE 1588-2008 [41], which was named as PTP Version 2 and lacked backward compatibility with the first version. Only version 2 will be considered in this thesis.

While PTP shares the obvious goal with NTP of synchronizing clocks, they differ on several aspects. To begin with PTP defines time in its packets in seconds and nanoseconds, clearly showing PTP's focus on high-precision environments. As a result PTP also puts requirements on the environment it is to operate in, and the specification explicitly mentions Ethernet as an ideal environment, where longer network distances are preferably avoided. Although PTP is possible to run in pure software mode, it needs supported hardware in order to achieve the best possible precision. Hardware might include devices such as network interface controllers, routers, and switches.

How it works

The structure of a PTP clock distribution system is as in the case of NTP a master-slave architecture, where a master provides the time to a number of slaves. To achieve the high-precision synchronization that PTP can provide, the hardware timestamping is important. With hardware support the latency errors can be in the nanoseconds range, while at application level they are in the range of hundred of microseconds to milliseconds [41].

While NTP uses stratum numbers to select time sources, PTP utilizes a so-called 'best master' algorithm. This basically means that a node either enters slave mode or master mode, depending on the PTP network. If there are no masters available, or the masters available performs worse than the node itself, it becomes a grandmaster. If however there are better performing masters available, it automatically becomes a slave. This allows a node to become a new grandmaster, if the current grandmaster for any reason disappears or suddenly loses accuracy. Such a reason could be losing its GPS connection, if that would be the time source.

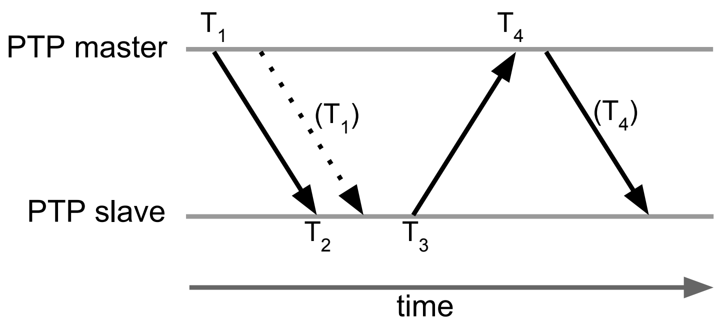


Figure 2.7: The exchange of packets between slave and master in PTP.

When calculating the clock offset and round-trip time basically the same four timestamps used in NTP are present as well. Equations (2.1) and (2.2) are also used to compute the values needed to synchronize the clocks. The difference from the already mentioned NTP case is that PTP is preferably run in multicast mode, although unicast mode also is supported. This slightly redefines the origin of the timestamps, as illustrated in Figure 2.7. As can be seen in the figure the master begins with sending a *Sync*-message at time T_1, which is received at time T_2 by the slave. To measure the round-trip time, the slave then sends a *Delay_ Req*-message at time T_3 to the master, which it receives at time T_4. The master finally sends a

Delay_ Resp-message to the slave containing T_4. The master may send a so-called *Follow_ Up*-message after the *Sync*-message, containing T_1, if its hardware does not support to accurately timestamp the *Sync*-message when it is sent. As the computations are the same as for NTP, PTP is also vulnerable to asymmetric network paths, and will thus not be able to differentiate it from the clock offset.

The frequency at which these *Sync*-messages, and all other messages, should be sent are of course configurable in PTP, and the frequency is implementation specific. In the case of PTPd [42], an open source PTP-implementation for Unix-like systems, the *Sync*-message interval is for example one second [43].

Evaluations

In [44] PTP-solutions from four manufactures were tested by Novick et al., in order to see how well these may perform under three conditions: master and slave on the same subnet in the same room, five network elements between the master and slave, and finally the public Internet between the master and the slave. As these were complete solutions, they may have involved PTP-aware hardware as well. In the first test, same subnet, the average time offset for all solutions was below 60 ns, while peak-to-peak variations ranged from 45 ns up to more than 170 ns between the solutions.

With five network elements involved the results worsened: three of the solutions managed to uphold an average time offset of below 10 μs, but the fourth had an average offset of 22.9 μs. The fourth solution had on the other hand only a peak-to-peak difference of 2.0 μs, a value the other three exceeded.

In the last test, over the Internet, only three of the solutions were available for testing. One of the solutions managed an average offset of just under 10 ms, while the other two showed an average slightly over 10 ms. The solution with an average just under 10 ms also showed a relatively small peak-to-peak variation of 56 μs, with the other two varying in hundreds of microseconds. As a comparison NTP was also tested in the same conditions over the Internet. Interestingly enough this resulted in an average offset below 3 ms, however with a peak-to-peak variation of more than 2 ms.

Another examination of PTP's precision was carried out by Kovácsházy and Ferencz in [45], where the previously mentioned PTPd was used. Apart from the examination itself a minor modification of PTPd was also made in order to decrease the settling time, while no negative impact on the stability could be observed. The results showed a mean time offset of 0.95 μs for an unloaded software only environment, with a standard deviation of 1.16 μs. When putting on load, the offset increased to around 30 μs, and the standard deviation slightly higher than so. The second environment consisted of mixed software and hardware. This means that the clients – end nodes – had PTP aware hardware, but no other devices on the network were PTP aware. In the mixed environment the mean offset ranged between 47 ns to 66 ns, depending on the load, and the standard deviation between 38 ns to 15 μs.

Availability

As previously mentioned there is a free PTP implementation available for Unix-like systems, PTPd. If running a Linux-distribution LinuxPTP [46] can also be used, which supports both running as only software and with hardware timestamping. Neither of these do however seem to support unicast mode with multiple slaves. For Microsoft Windows there are no open source implementations of PTP to the authors' best knowledge, but there are a number of commercial alternatives available.

2.4 Clock drift and clock skew

An inevitable problem when working with synchronization is the one of clock drift, which just as the name implies is the drift of a clock often compared to a nominal perfect reference clock. No matter the accuracy of the clock, all clocks will drift sooner or later. Therefore it needs to be considered especially when working with synchronization.

Clock skew is another important phenomenon that needs to be considered. When talking about clock skew in general it often refers to the fact that not all components in an electrical circuit receive the clock signal at exactly the same time. It may also refer to the difference in readings of clocks at different nodes on a network, such as the Internet. Simply speaking this means that there may be a time difference between two clocks at two different units on the network since the clocks are not synchronized. This is of course also of importance in the context of the thesis. Moreover the term can for example also be used when talking about the time difference between two local clocks on a computer. Thus networks do not necessarily need to be involved, although this is common.

The two terms are closely related. This as given that the problem with clock skew is solved by for example using NTP to synchronize the clocks, the clock drift will once again lead to clock skew which once again needs to be corrected, and so on. Although closely related, when looking at media synchronization, clock skew is however what actually determines how exact the synchronization may be. The explanation being that the timestamps of a packet is used to determine when it should be played back. Clock drift on the other hand introduces the problem of resynchronization, which a major part of this thesis concerns.

Finally it can be argued that the distinction of what can be classified as an actual clock drift and just short-time network jitter is not always an easy one to make [23]. As the thesis specifically concerns streaming over IP networks, this problem may be present. The consequences of correcting such a 'false' drift may lead to the correction being counterproductive. This is because the drift may correct itself until the next packet arrives, given that it is caused by jitter. What this in turn leads to is that yet another correction needs to be carried out, back to the initial value. The problem can partly be solved by allowing a certain amount of drift before trying to correct it, however determining a proper value may prove not to be trivial. If the value is set too high, unnecessary drift is allowed, making the synchronization less accurate, whereas a lower value may introduce false corrections.

2.4.1 Clock drift

The clock skew can be said to be almost constant if it would not be for the clock drift. Furthermore solutions for clock skew have been discussed, such as synchronizing the clocks using NTP or PTP as explained in Section 2.3. The upcoming sections will discuss how and to what extent this drift occurs.

2.4.2 Reasons for clock drift

As mentioned previously, an oscillating crystal is typically used as a clock in computers. Readings of the number of oscillations will then determine how much the clock has ticked. As can be expected, all crystals may not oscillate at the same pace, which introduces the problem of drifting clocks.

2.4.3 The magnitude of clock drift

The next question is how much it will occur, which also typically differs depending on the crystal used. The accuracy itself is measured in parts per million (ppm), or some times parts per billion (ppb), which states how many seconds per second the clock may drift in general. For example an accuracy of 1 ppm, which can be said to be typical for a normal computer grade crystal [48], would result in a drift of

$$1 \times 10^{-6} \times 60 \times 60 \times 24 = 0.0864$$

seconds per day. This value does of course differ between crystals, and this drift is the reason why an algorithm such as NTP or PTP is needed in order to keep the clocks tightly in sync.

Looking more specifically at the hardware used in this thesis, namely Raspberry Pi:s, it can be seen that an accuracy of around 41 ppm can be expected which would lead to a

$$41 \times 10^{-6} \times 60 \times 60 \times 24 = 3.542$$

seconds drift per day [49], which is considerably higher. Although the Raspberry Pi:s used in this thesis use updated hardware which may affect the accuracy, this should give a hint of what the accuracy could be. In order to improve the accuracy a more accurate clock could be introduced, such as a GPS or a real-time clock. No matter the actual drift it needs to be corrected nonetheless, since it will be present sooner or later.

2.4.4 Drift correction

This finally leads to the question what the implications are to correct such a drift. The implications may usually not be of high importance, since it often does not matter if a correction of a clock is being made. This simply because it would not be noticed in everyday usage. However, looking specifically at the system clock and the audio clock on a client, this correction must be carried out with extra care.

There are different alternatives for correcting such a drift, as can be seen in Section 2.5. What can be said is that strictly slaving the clocks to each other may introduce glitches or artifacts in the stream. Thus although the clock drift itself is possible to correct, doing so may lead to an unsatisfactory audio experience if not done in a proper way.

2.5 Resynchronization

As explained in Section 2.4 a drift between two clocks is inevitable. Resynchronization of them therefore needs to be carried out sooner or later if the synchronization should be kept. This can be seen as a trivial problem at first and partly is, if no requirements on how it can be done are present. However, when taking a closer look focusing on minimizing the audible artifacts in the audio which usually are introduced when working with resynchronization, the problem is far from trivial.

In the context of the thesis two clocks are especially focused on when working with resynchronization, namely the system clock and the audio clock. It is true that the system clocks across the devices must be corrected as well, as these also will drift. Performing corrections on these will not affect the playback of the media however, as it is the audio clock that is used to determine when the media should be played back. Therefore it is not as important how the correction of the system clocks is carried out, as how to make the audio clock follow the system clock, and thus the global time.

Focusing on the correction of the drift between the audio clock and the system clock then, it is not enough that the audio clock is modified to decrease the drift between the two clocks. This is at least the case when focus on high quality audio playback. The reason is that it would introduce audible artifacts. To understand why a simple example may be considered. The media stream sent over the network will be divided into different segments, each having its timestamps of when it should be played back. These timestamps will not contain any overlaps or gaps between them, instead the stream of segments will of course be continuous. If the audio clock then suddenly is modified without modifying the timestamps of the segment, the segment will be played back at a slightly different time. Thus it may overlap or introduce a gap with the previous segment.

Therefore the segments need to be modified when the resynchronization occurs, where the most common methods are:

- introducing silence
- jump ahead in the playback (by skipping samples)
- altering the playback speed to either catch up or slow down, this itself is done by resampling

The two first approaches are more naive since no advanced algorithm is required in order to implement them. Introducing silence could for example be done by the client farthest ahead in the playback by simply pausing the playback until the other clients have caught up, whereas skipping samples could be carried out by the clients in the back, until they have caught up to the one farthest ahead.

Nonetheless, both of the approaches will of course introduce audible artifacts, be it just silent gaps, or actually glitches in the playback since samples are being thrown away.

It is clear that the two more naive approaches have their problems in regards to audible artifacts introduced, and can be said not to be acceptable if high quality audio playback is of importance.

Left is the third alternative, altering the playback speed by resampling. There are both more trivial approaches, such as duplicating samples, to more advanced algorithms, which might consider both the previous and the upcoming sample.

2.6 Related work

Design, development and evaluation of an adaptive and standardized RTP/RTCP-based IDMS-solution

Looking at the possibilities for achieving inter-destination media synchronization using a RTP/RTCP-approach, the standard implementation does not support this. Instead it supports achieving lip synchronization. This means that it only concerns synchronizing an audio and video stream, and not streams to multiple receivers. A solution to this problem was proposed in 2015, where an extension of RTCP was designed and developed. This proposition made it possible to achieve inter-destination media synchronization as well by mainly using RTP/RTCP [53]. This was created as part of a PhD by Climent in [54]. The RFC however states that the actual method of synchronizing the time across the devices is left to the implementer.

In addition to the proposed standard, an implementation based on GStreamer was created, which achieved excellent results with regard to inter-destination synchronization. As the focus of the thesis is to achieve the same type of synchronization using GStreamer, this of course can be seen to have been of major interest. However, the implemented solution, as well as the PhD, focus on achieving inter-destination media synchronization for mainly video. The PhD does touch upon audio as well, but regarding the resynchronization part which has been a major part of this thesis only a few examples how it may be solved are introduced. This in comparison with video where an exhaustive discussion is held. Thus if this would have been used in this thesis, the core problem of resynchronization would still be present. Moreover, it would be needed to implement the standard as an extension in GStreamer, since the existing implemented solution has not been made available when this thesis was written.

Chapter 3

GStreamer

GStreamer is an open-source multimedia framework which provides a large number of libraries and plugins that enables developers to create advanced media processing applications. In the context of this thesis GStreamer does for example provide plugins for handling UDP packets, RTP packets, as well as encoding and decoding audio. It does even provide functionality to achieve inter-destination media synchronization between the clients, but as will be seen later the solutions provided by GStreamer to achieve the synchronization have their problems concerning the audio quality.

The core framework is written in C, but there are also bindings for different languages such as Python and Java, making it possible to use in a great set of applications. Moreover, there are bindings which makes it possible to use with for example .NET in Windows. However, as these only are bindings, although official, all functionality in the core framework cannot be guaranteed to have been mapped.

3.1 GStreamer introduction

3.1.1 GStreamer elements

In order to understand how GStreamer works it is important to understand the concept of elements. This is the base in GStreamer, where each element is part of a plugin, which can provide some decided upon functionality. Each element has one or multiple 'sinks', where it receives its incoming data to process. In the same way it may have one or multiple sources, on which it provides its processed data, to make it available for other elements. In order for these terms to be more understandable, all elements are conceptually chained together, which means that between each two consecutive elements there is a connection. This connection has a 'source', where the first element provides its data, and then a 'sink', where the data is made available for the second element. The terms should be seen from the point of view of the data flow, as illustrated in Figure 3.1.

Each element can also expose an arbitrary number of configurable parameters, and internally store and process the available data as it sees fit. This means that an element can for example buffer data during a certain time period, as well as have an internal clock that is based on the sound card in the computers, as is the case for the AlsaSink element.

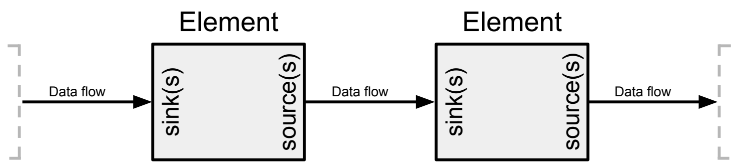


Figure 3.1: The flow of data between elements in GStreamer.

3.1.2 The pipeline

Since an element only provides a certain type of functionality, for example processing UDP packets, or RTP packets, many elements often need to be used in combination in order to achieve more advanced functionality. One such example could be streaming media over a network. In order to link all of these elements together a 'pipeline' is used in GStreamer. To this pipeline an arbitrary number of elements can be added, and through it all elements can retrieve access to global information, such as a global clock and state.

3.2 Clocks and synchronization

To provide an understanding of what functionality GStreamer offers, a complete example for playing a received UDP and RTP stream, where the audio is encoded with Opus, can be considered. This corresponding pipeline can be conceptualized as in Figure 3.2, and will be relevant for the actual implementation as well. A brief explanation of each element in the figure follows:

UdpSrc The UdpSrc element can receive UDP packets on a certain port, and unpack these. Supports multicasting as well.

RtpBin The RtpBin element provides a jitterbuffer, and support for handling the processing and sending of RTCP packets. It does also provide options in the process of synchronizing the incoming data.

RtpOpusDepay The RtpOpusDepay element unpacks RTP packets into Opus frames.

OpusDec The OpusDec element decodes Opus frames into raw audio frames.

AlsaSink The AlsaSink element receives an audio stream and communicates with the ALSA layer in the operating system. The ALSA layer is the audio layer in Linux, which then sends the audio on to the hardware for actual playback. The element does also provide methods for resynchronization due to drifting clocks, as explained below.

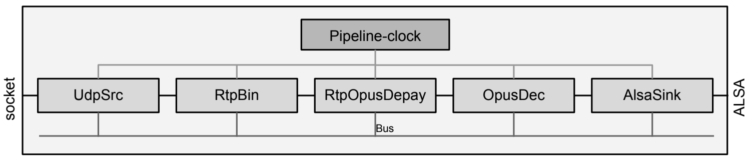


Figure 3.2: Illustration of a complete GStreamer pipeline.

As explained in previous sections clocks play an important role in synchronization, and in the pipeline there are primarily two clocks that are of interest in this particular case. These are the global pipeline clock, which will be a 'GstSystemClock', and the one in the AlsaSink element, which will be a 'GstAudioClock'. The former uses the built in system clock as the name implies, whereas the latter typically uses the internal clock of the sound card.

As the GstSystemClock is directly connected to the system clock of the computer, and thus fetches its time by polling the operating system, it will be affected by changes made to the system clock of the computer as well. This is why using for example NTP to synchronize the operating system's clock also affects the GStreamer pipeline. What this means in turn is that GStreamer is decoupled from the time synchronization protocol in use, as it will only look at the system clock. Therefore NTP and PTP can be used interchangeably in GStreamer, without any additional configuration.

The GstAudioClock ticks independently of the system clock, and is computed by looking at the number of audio samples consumed by the sound card. This makes it tightly connected to the actual ticking of the hardware clock of the sound card. Also, it is here a difference between the clocks may occur, namely between the system clock – the GstSystemClock – and the actual consumption rate – the GstAudioClock. How the clocks are resynchronized to each other is explained in Section 3.3.

3.2.1 Synchronization elements

The inter-destination synchronization itself in GStreamer is achieved by different elements working together. To understand this it is important to remember that in order for multiple clients to stay synchronized to each other, the system clock needs to be the same across all the devices. What also is vital is that the audio clock needs to be slaved to the system clock, so that the actual playback happens at the correct time.

AlsaSink element

An important element is the AlsaSink, as it is responsible for the actual playback of the media. Moreover, it is the element which uses the GstAudioClock, which needs to be tightly synchronized to the GstSystemClock. The reason for this is that it will not matter if the system clocks between the clients are synchronized to each other if the audio clock, which in the end determines when the data should be consumed, will tick without regard to the system clock.

One concept that can be clarified is the one of segments and samples. When the data arrives to the AlsaSink element, it arrives in so-called media units (MU), also called segments, which each includes a certain number of audio samples as can be seen in Figure 3.3. To then make the samples available for consumption by the sound card they are written one by one to the ring buffer of the ALSA layer. This layer is then responsible for playing them back later on the output.

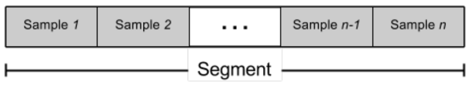


Figure 3.3: The relationship between a segment and a sample.

Problems may arise if the segments are not aligned properly, that is, they overlap or have gaps between them. This in turn will introduce glitches in the audio being played back.

3.3 Resynchronization methods

How the clock is corrected, when it is corrected, and how this affects the playback is determined by the 'slave-method' parameter of the AlsaSink element. The different alternatives currently provided by GStreamer and a short description of what they are trying to achieve is found below:

Resample Slaving The 'Resample Slaving' method tries to resample a segment in order to put two consecutive segments together without any glitches between them. This is done by simply duplicating samples if the segment needs to be longer, or delete samples if the segment needs to be shorter. The method also uses linear interpolation in order to try to interpolate the audio clock to the system clock.

Skew Slaving The 'Skew Slaving' method will instead of avoiding a glitch between two consecutive segments the gap, introduced by the clock correction, leave this unaltered. This will force a jump in the playback since a few samples will be skipped in order to slave the audio clock back to the system clock.

None As the name implies this alternative does not synchronize the audio clock to the system clock, and will thus be irrelevant in the context of this thesis.

3.3.1 Resample Slaving

Brief introduction

In the 'Resample Slaving' method the idea is to tightly slave the audio clock to the pipeline clock, which will be the system clock, by continuously altering the time and rate of the slaved clock. This is done by setting the pipeline clock as the master clock for the audio clock. What this in turn will result in is that a method sampling the master's and the slave's clock times will be called, at default at an interval of ten times a second.

The sampling method called will do a linear regression, using the least squares fitting technique, in order to determine a suitable rate and correct time for the slaved clock, given that there are enough samples. The slave clock is then calibrated with these linearly regressed values. Finally the rate and the other calibration values of the slaved audio clock are fetched and used when computing the start and stop times for the segments. This will affect the lengths, and thus to what degree they are extended or shortened.

The 'Resample Slaving' method will thereby never introduce gaps between segments or samples. Instead it will try to affect the playback rate – in the primitive way of deleting or duplicating samples – in order to follow the master clock.

Evaluation

When examining the 'Resample Slaving' method in GStreamer the problem is partly that it is too aggressive when it performs the resampling, meaning that it is altering the frequency too much which will distort the audio to a great extent. This is due to the fact that it only uses one segment to carry out the correction needed instead of trying to correct it over a longer period of a time.

An example of why this is a problem can be illustrated by having two consecutive segments each containing 100 samples and having a gap corresponding to 25 samples in difference between them. One of the segments will then add these 25 extra samples missing to it. This will of course alter the sample rate drastically, distorting the audio.

Although the example may be extreme and the gap between two consecutive segments typically is less, over a longer period of time there is a risk of a larger gap between the segments. This may be due to temporary network jitter, or other temporary instability of the playback, such as resynchronization of the system clock caused by for example NTP when it has drifted too far. Hence the algorithm clearly has its problems, since these drastic rate changes are allowed.

The other problem of the method is the currently used algorithm for resampling the actual audio. This algorithm simply duplicates or removes samples from the segment that is arriving. An illustration of the problem can be found in Figure 3.4. Here a wave is oscillating at the same frequency continuously. When the information is streamed over the network a number of sample points symbolize the waveform, as the analogue signal must be stored in a digital wave some way. The sample points can also be seen in the figure as small stars.

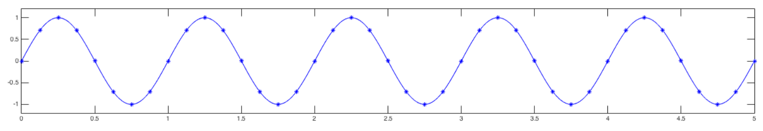


Figure 3.4: A sine wave with corresponding sample points, where no sample points have been duplicated.

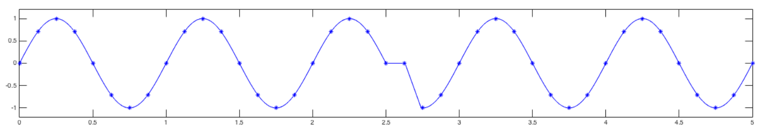


Figure 3.5: A sine wave with corresponding sample points, where one sample point has been duplicated.

When the analogue signal then should be reproduced, the digital sample points are used to make an approximation of the original waveform. Thus if one of the sample points is duplicated, and the waveform at that point is descending, the duplicated sample will make the waveform stop its descent for a short moment. This is illustrated in Figure 3.5. What this in turn does is of course introducing an artifact as the waveform will be malformed. If even more samples are duplicated next to each other, the distortions will get even more noticeable.

Finally, using linear interpolation to slave two clocks to each other can also be said to be a rather primitive algorithm and thus also having its flaws. An example of such a flaw is that it is making it possible for the clocks to change faster than might be necessary, potentially introducing larger gaps between the segments.

3.3.2 Skew Slaving

The skew slaving method will look at the different clocks involved when trying to find a drift between the clocks. These clocks are the pipeline clock, which in the case of the thesis will be the system clock, and the audio clock. The method will just as the 'Resample Slaving' one always try to slave the audio clock to the system clock. However, 'Resample Slaving' will use interpolation techniques to do this, whereas 'Skew Slaving' simply will set the time hard by calling a method on the audio clock. This is to make sure that the time of the audio clock corresponds to the one of the system clock at all times. Thus jumps may constantly be present since the corrections are carried out at each new segment received.

To avoid all of the arising glitches in the playback as the time of the audio clock is being modified, the key is that up to a certain threshold the segments will still be aligned. This means that the timestamps of the segments also will be modified to make sure that they follow the corrections of the audio clock.

What this in turn means is that even though the audio clock may do a hard jump, a corresponding jump will be done in the timestamp of the segment, which will ensure a smooth playback. However, this will of course not correct the drift between the system clock and the audio clock, as the same modifications are done both to the audio clock and the timestamps of the segments. GStreamer solves this by saving the total drift between the system clock and audio clock.

As the method would of course be useless if it never would correct the actual drift, a correction will be carried out when the threshold of what an acceptable drift is has been reached. More specifically when this threshold has been reached, GStreamer will simply not modify the timestamps of the next segment. This in turn will impose a glitch between the segments, and therefore the clocks will be synchronized to each other again.

The reason for actually having a threshold is to avoid unnecessary glitches in the playback, due to temporary jitter or errors in calculations.

Drift-tolerance parameter

The question is then what decides when the segments should not be aligned and thus when the glitch should occur. This can actually be set as a parameter in the AlsaSink element, called the 'drift-tolerance', which simply determines how much drift given in microseconds is allowed.

It is not obvious what value this parameter should have, but given that a tight synchronization is wanted, the lower, the better of course. At the same time having a low value may be counterproductive since jitter is present on the network and the audio clock may be slaved in the wrong direction. This may in turn mean that a correction back to its original position is needed when the next segment arrives. Therefore it may not be advantageous at all times to have the lowest value possible, due to problems which are connected with the resynchronization.

The final choice of the value depends on the application. The standard value used is 40 ms which as discussed in Section 2.2 can be said to be a reasonable value.

Evaluation

The problem of this method when focusing on smooth playback is that it allows a glitch between two consecutive segments. Simply put this will introduce a spike in the sound, which may more or less always be audible as a 'click' or 'pop' sound depending on what kind of audio is playing. An example where this also can be seen is in a spectrogram, which can be found in Figure 3.6. The circled spike at around the 15 second mark is due to the resynchronization part of the algorithm.

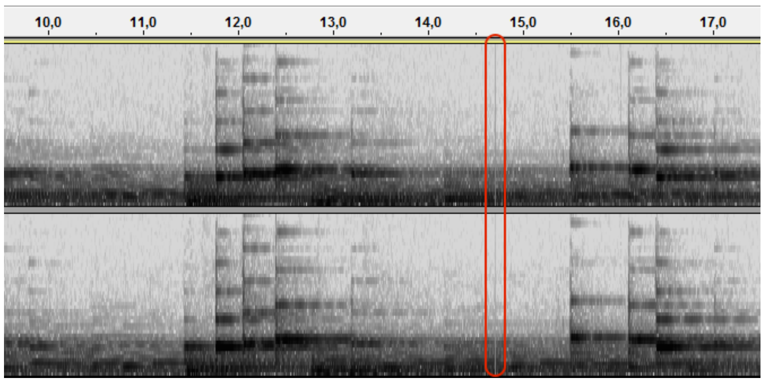


Figure 3.6: A spectrogram showing a circled spike shortly before the 15 second mark which are introduced when the clocks are resynchronized. A drift-tolerance of 5000 μs was used.

Given that these spikes would not occur frequently they could have been somewhat acceptable. But despite using a drift tolerance of 40 ms pragmatic tests carried out still showed that spikes due to resynchronization would be present multiple times per hour. An even higher drift tolerance would not be possible as the sound most likely will be perceived as out of sync, as argued by Wallach et al. in [35].

It is true that with the help of the drift-tolerance parameter the size of the glitch can be modified. In the previous figure a drift tolerance of 5000 μs was used, and thus the spike will have a higher intensity compared to if the size would have been smaller. However, even when using almost the lowest drift-tolerance possible spikes are present as can be seen in Figure 3.7, although they will have less intensity.

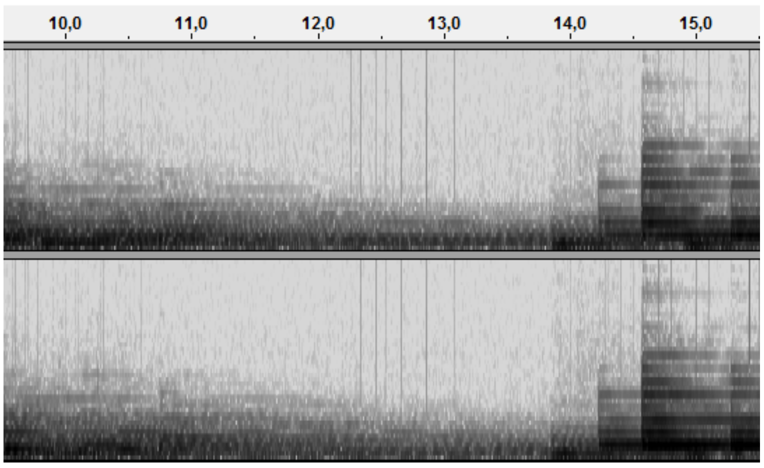


Figure 3.7: A spectrogram showing reoccurring spikes, for example between the 12 and 13 mark, which is introduced when the clocks are resynchronized. A drift-tolerance of 5 μs was used.

The glitches with lower intensity are still audible, although this partly depends on the audio. Furthermore, even if GStreamer would have allowed a glitch of only one sample, this would also introduce visible, and more importantly audible, spikes in the sound. This can be seen in Figure 3.8 at the 0.11 second mark and the 0.30 one, where just one sample has been silenced each. Thus no matter the size of the gap, corrections may always be audible

3.3.3 Conclusion of the methods

When evaulating the two different methods for resynchronization in GStreamer it can be seen that from a synchronization point of view, the algorithms should be sufficient in GStreamer. Thus the problem is not that they do not provide a tight sync, but rather the actual resynchronization when a drift has occurred.

The aim of this thesis has however been trying to find a resynchronization algorithm which ensures as smooth playback as possible. Therefore it can be said that the provided methods have their problems when focusing on this aspect, which also is why a proposed improvement will be introduced. But seen from a

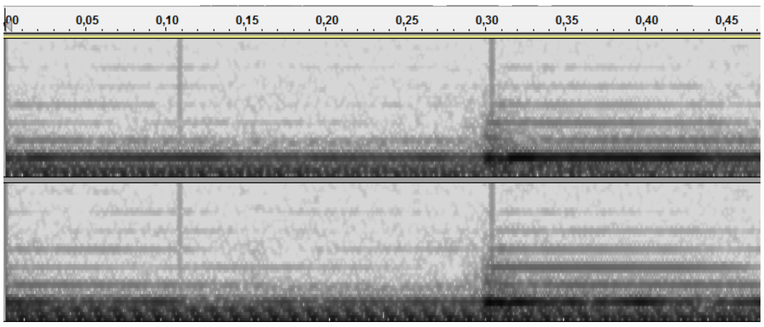


Figure 3.8: A spectrogram showing spikes present around the 0.11 and the 0.30 mark due to just having silenced one sample at a time, each only 20 μs long.

synchronization point of view, these methods are sufficient.

Chapter 4

Selecting Streaming and Time Synchronization Protocols

The following sections will cover comparisons which were carried out in order to see what choice of streaming and time synchronization protocol would be the most suitable one. After this a short discussion about the final choices made follows.

4.1 Streaming protocol: TCP, UDP and DCCP comparison

Each of the protocols considered have their advantages and disadvantages depending on the context they are used in. Below a comparison looking at the aspects important specifically for this thesis is found. The protocols thus may have more strengths and weaknesses than mentioned.

4.1.1 Widespread support

First of all the support on the Internet is crucial since a requirement is that the prototype should work over an unmanaged network. As commonly known both UDP and TCP are widely used on the Internet, and are therefore supported in routers and NAT devices across the globe. DCCP on the other hand lacks support in NAT devices to a large extent [20], which poses a critical problem. It is true that this partly can be solved by modifying DCCP, but this would on the other hand impose modifications [21], which would be time consuming. Moreover a lack of support is present on the Windows platform as well, clearly making it the less supported protocol of the three. This as both TCP and UDP are supported.

As a final small note RTP over UDP packets may have higher priorities in the routers throughout the Internet, compared to TCP [23].

4.1.2 Delay versus error correction

Delay, latency, and retransmission

Since the thesis primarily concerns music streaming, it can be argued that a delay is allowed in comparison to having glitches in the playback due to lost packets. Looking at the different alternatives TCP may thus be the better choice since it will guarantee a glitch free playback, because lost or corrupted packets will be resent.

In contrast UDP will not be able to recover lost or corrupted packets. However, this statement requires qualification if UDP is combined with RTP, which it most likely will be in streaming applications just because of its shortcomings for this type of application. This means that when then RTP is used in conjunction with UDP, retransmission of lost packets may be possible [30], when extensions to RTP are used. Of course this would require a buffer introducing a delay just as in the TCP case, but the aspect of handling lost packets with respect to delay can be said to be a tie between the two protocols given that RTP is used.

Looking at DCCP instead the standard definition does not offer support for retransmission of packets. However, this may be solved in the same way as in the case with UDP, just by simply using RTP in conjunction with DCCP [24].

What can be said is that using TCP inevitable will result in a delay due to its buffers, independently of whether packets are lost. This in comparison to UDP which will not buffer its content and thus keep delays at a minimum. Moreover, even more delay may be introduced by the mechanism present in TCP to not provide packets with a sequence number higher than any lost packets to upper layers. Therefore, given that the network is congested, resulting in many lost packets, a problem may of course arise here if the resent packets in turn also are lost. This problem is especially present given that only a message should be streamed, since some glitches due to lost packets may be allowed as long as the announcement still is audible. Therefore latency in this case may be highly unwanted since the announcement should reach the listener as quickly as possible. Low latency is also especially crucial if the announcer is in the same room as the announcement being made and will be able to hear him- or herself. Although this is a minor focus of the thesis, it is also considered. In this case UDP may be the better choice, as this at least gives an allowance for a few dropped packets with low latency compared to TCP, where this is not an option at all.

On the other hand, it can be debated how much packet loss actually is present on unmanaged networks, and thus how much of an extra delay this would impose in the end. Unfortunately an estimation of how common it is that packets are subject to loss on the Internet has not been found, which makes it hard to debate to what extent this is a present problem. What can be said is that TCP will always introduce a slightly higher delay in comparison to UDP [18], why UDP is the better choice when working with applications where a low latency is of utter importance.

Codec concealment

To complicate matters even further the need of retransmission can also be debated. First off depending on what codec is used, mechanisms for adaptive playback as well as concealment of lost packets may be present. Thus given that a packet is lost, the codec may be able to conceal the loss quite well. Moreover, the codec may also provide mechanisms for adapting to congestion on the network. This means that given that the network becomes congested, the bitrate can automatically be lowered and thus requiring less bandwidth. One example of such a codec providing these mechanisms is Opus [25], which also was the required codec to use in this thesis. Therefore these mechanisms should be taken into account when discussing the protocols. However, even when packet loss concealment is available it is obvious packet loss may still be audible, but the result may affect the user experience less negatively in comparison to a glitch.

Referring back to the debate on how much packet loss actually is present on the networks, packet loss may not be a great problem in the end. Thus the extra delay needed to recover from losses may not be needed, given that a proper codec with error correction by a concealment algorithm is used.

4.1.3 Congestion control

Having in mind that the clients can be placed across the globe, the risk for congestion on the network is present, meaning that congestion control can be advantageous. Both TCP and DCCP provide mechanisms for this, where DCCP even provides an alternative focusing on real-time applications such as streaming. In this case the send rate is tried to be kept smooth in comparison with TCP where abrupt changes are allowed. UDP on the other hand does not have any support for congestion control at all, which may pose a problem.

There are disadvantages of imposing congestion control though. For example using the congestion control mechanisms of DCCP increases the delay and jitter in comparison with just using UDP [18]. On the other hand the packet loss will be less, and a more efficient throughput will be present given that DCCP is used. TCP in turn offers an even better throughput with less packet loss in comparison with DCCP, but the cost of this will be a higher delay and more delay variation. Although the throughput of DCCP is lower than that of TCP, the TRFC mechanism does provide a far smoother data rate, as discussed in the DCCP evaluation. This smoother data rate is advantageous in streaming, as the risk for buffers underflowing or overflowing is decreased.

4.2 Time synchronization protocol: NTP and PTP comparison

4.2.1 Performance

Apart from a proper streaming protocol, a proper time synchronization protocol also is needed in order to ensure the synchronized playback.

Looking at the precision of the two protocols considered in this thesis, namely NTP and PTP, PTP clearly outruns NTP, at least under good conditions. Two aspects that must be considered however are the actual precision needed, and how well NTP and PTP performs under non-optimal conditions. This as a perfect LAN environment cannot be assumed.

Regarding the first aspect some studies show that the precision needed is in the range of milliseconds, depending on the sound. Both protocols achieve this with a good margin under good conditions. However, under WAN conditions NTP quickly approaches the limit for what is acceptable. Interestingly enough Novick et al. [44] noticed even worse performance from PTP under these conditions, although NTP had a greater variance. Thus it may not be certain that despite

PTP being more accurate during good conditions, that it would outperform NTP in all cases. In addition one more aspect should be remembered when analyzing the results – most likely PTP-aware hardware was used, which cannot be assumed for this thesis. An even worse performance when using PTP during WAN conditions should therefore be considered as fully possible in the context of this thesis.

Extensive investigations of how NTP actually performs over current WANs are rare, at least publicly available, which makes it hard to do a theoretical evaluation.

Finally focusing on the application of this thesis, neither protocol is perfect. NTP focuses on synchronizing personal computer clocks over the Internet, with minimal overhead, while PTP shines on LANs where a high precision is much more important than whether some extra overhead is introduced. The rationale behind the choice of NTP and PTP was that they are common choices in the field of time synchronization protocols.

4.2.2 Support and availability

Due to the many years NTP has been in existence as well as due to its popularity, a wide range of open and free clients are available, both for Unix-like systems as well as for Windows systems. PTP on the other hand is still a protocol much under development, in the sense that the availability of open and free full implementations of the protocol are few. Although alternatives such as PTPd and LinuxPTP supports much of the PTP specification, a feature such as proper unicast support is missing. For Windows the situation is worse – clients are available, but mostly commercial versions. If the full potential of PTP should be utilized special hardware is needed as well, further complicating the matter, but as it is possible to run in a software only mode, it can still be considered a viable alternative.

4.3 Final choices made and discussion

4.3.1 Streaming protocol

In the discussion of a proper streaming protocol there were three candidates: TCP, UDP, or DCCP. While DCCP should be the perfect choice seen from the protocol's target application, it suffers from major support issues. For example Microsoft's Windows platform lacks support for it, and the support in NAT devices and firewalls is very scarce. Just implementing a Windows version according to the specification could probably be a thesis project on its own, why it would not fit within this thesis' time constraints. This albeit the advantages the protocol itself may bring. Further evaluations regarding the performance of DCCP are needed, but on the whole the focus of the protocol, flexibility, and functionality speak for it. If it would start to gain some more reputation as well as mature, it might become a very good candidate for applications like this thesis, but will due to lack of support be discarded from use in this project.

In the comparison between TCP and UDP, the greater flexibility of UDP – although lack of reliability – is a great advantage. It is true that mechanisms for congestion control are missing, which may lead to higher packet loss with no chance of recovery. Even more prominent, the order of the packets cannot be guaranteed. This as sequence numbering is not an option, which is a must if a well-functioning stream should be achieved. The latter problem will however be solved by introducing RTP, and introducing RTCP even statistics can be achieved on the network. This in conjunction with the risk of TCP's possible abrupt changes in sending rates in case of network congestion, leads to that UDP will be used in the prototype, in conjunction with RTP to provide orderly playback. This as abrupt changes in sending rates are not suitable for real-time media streaming.

The discussion in itself can be said to be less interesting however since GStreamer will be used as streaming software, which supports both TCP and UDP where they easily can be interchanged if needed. Thus the thesis will focus on using UDP, but the prototype will also allow usage of TCP instead with a small addition, if the advantages of TCP can be seen as more advantageous in a specific situation.

4.3.2 Time synchronization protocol

In order to synchronize the system clocks across all receiving devices the choice stands between NTP and PTP. While NTP's theoretical precision is far below the one of PTP, unless modifications are done to it, it has the great advantages of being widespread, open, and putting few requirements on the network. During many circumstances NTP's precision is enough to fulfill the limits for what is perceived as synchronized, although PTP may actually accomplish far better values.

Although PTP v2 supports WANs, its performance is questionable in comparison with NTP according Novick et al. in [44]. Moreover, it can be run in a software only mode, but the number of available solutions are limited, particularly to Windows. But as the solution of how to synchronize the system clocks is completely decoupled from the GStreamer solution which will be used in the implementation, both alternatives may be tested and evaluated. However, due to time constraints the tests will typically be carried out favouring NTP, where a corresponding PTP variant of the test will be made where possible. This is due to the fact that NTP can be said to be more widespread. Moreover, in a realistic situation where the server will be running .NET on Windows, which has scarce support for PTP, it may be hard to include a PTP server on the server in a final solution. Thus the focus will be NTP since this can be guaranteed in a final solution, whereas PTP cannot.

4.3.3 Resynchronization

Clock drift will always be present to a certain extent, and thus the need of resynchronization is inevitable. The alternatives of skipping samples, or playing and pausing the playback for the clients, in order to get back in sync with each other, is not a viable option. This as the focus of the thesis is to provide as high quality playback streaming as possible, while still keeping it synchronized. No matter how the resynchronization is done using these methods, artifacts will be introduced. Moreover, they will be audible, no matter the number of samples skipped or the time paused. It is true that this might be just a minor artifact, however since the artifact always will be present given that this mechanism is chosen, the method of using resampling has been chosen instead. It is true that GStreamer partly supports this natively, but problems with this implementation are present, and these will then be tried to be solved.

Chapter 5

Implementation

5.1 The server: choices and languages

The server uses .NET, since this was a requirement from the start. However, in order to provide functionality for sending RTP and RTCP packets, and in general streaming media, the official GStreamer-Sharp binding [55] was used. This made it possible to utilize GStreamer on the server side as well. The advantages of this are many. Not only does this solve the problem of a lack of a native support library for sending RTP packets in .NET, but this also allows for doing almost all media processing with the help of GStreamer. This in contrast to switching between different plugins.

GStreamer also allows for easy switching between using RTP over UDP or RTP over TCP since it supports both. Moreover, as the clients will use GStreamer it is a good match to have GStreamer on the server side as well, since all information sent can easily be handled by the clients.

Streaming of 'live' media is also possible in the implemented prototype, meaning that one can stream the content being played out on the sound card of the computer, so-called loopback. The Wasapi API in Windows provides functionality for loopback recording, but as it is exposed in C++, the wrapper library NAudio [56] was used, as it makes it easy to access Wasapi's features through C#.

5.1.1 GStreamer

The server implementation supports multiple sound sources, why the first pipeline element differs depending on the selected input source. Table 5.1 shows each of these elements and their corresponding use case. The succeeding sections will then shortly describe the elements that can be used as sources of audio, as well as the rest of the elements in the pipeline.

The source elements

Both the FileSrc and the AutoAudioSrc elements are simple source elements. This means that they either only need the location of the file, in case of the FileSrc element, or simply detects an audio source by itself, as in the case of the AutoAudioSrc. AppSrc on the other hand is a more complicated element, but also necessary, if the input should be in raw bytes. As the actual source in the loopback

Input	Element	Parameters
File	FileSrc	location
Loopback	AppSrc	stream-type do-timestamp caps
Microphone/line-in	AutoAudioSrc	

Table 5.1: The source elements and corresponding configuration parameters used in the server prototype.

case is NAudio which produces a stream of bytes, these needs to be stamped. This stamping involves for example times and media properties. After the process of stamping the stream of bytes they are made available to the next element, which then may interpret the raw audio.

The rest of the pipeline

After the source elements the rest of the pipeline follows, briefly described below:

Queue The Queue element acts as a buffer between two elements, where the data rate may not be smooth.

DecodeBin The DecodeBin element automatically creates a suitable decoding pipeline based on the incoming media.

AudioConvert The AudioConvert element supports conversion of raw audio buffers between different formats, such as the conversion of integer data to float data.

AudioResample The AudioResample element supports resampling of raw audio buffers between different sampling rates, such as 44.1 kHz to 48 kHz.

OpusEnc The OpusEnc element encodes raw audio buffers into Opus audio frames.

RtpOpusPay The RtpOpusPay element creates RTP packets from Opus audio frames.

UdpSink The UdpSink element puts the received data into UDP packets and send them to a configured destination. Supports multicasting.

Apart from the above elements, a 'RtpBin' is also used in conjunction with an additional 'UdpSrc' and 'UdpSink', in order to send and receive RTCP packets.

5.2 The clients: choices and languages

The clients use only GStreamer and GLib-related [57] functionality, meaning that the code on the client side is written in C. GStreamer is used both for processing received packets as well as achieving synchronization between the different clients.

However, the implementation has been slightly modified as explained under Section 5.3 in order to achieve a better resynchronization than the ones provided currently.

5.2.1 GStreamer

The elements and any relevant corresponding configuration parameters used in the client prototype can be found in Table 5.2. In addition to these elements 'RtpBin' uses a 'UdpSink' and an additional 'UdpSrc' element too, in order to be able to send and receive RTCP packets.

Element	**Parameters**
UdpSrc	
RtpBin	buffer-mode ntp-sync
RtpOpusDepay	
OpusDec	plc
AlsaSink	drift-tolerance sync slave-method

Table 5.2: The elements and corresponding configuration parameters used in the client prototype.

RtpBin

The RtpBin element offers a few different buffer modes for how timestamps on the incoming RTP packets should be handled. The actual difference between the individual receivers' time of playout and the shared global time only can be computed shortly before the actual playout. This means that the timestamps should not be smoothed or adjusted by the RtpBin. The reason is the delays that might be introduced by the elements themselves as the audio sample travels along the pipeline. Hence the skew corrections should be computed as late as possible, when the actual total delay is known and can be accounted for – which is not in the RtpBin element. As 'None' is the only mode that basically leaves the timestamps untouched – they are just converted to local GStreamer timestamps – this was the mode chosen for the application.

As global clock synchronization is carried out by a protocol outside of GStreamer, NTP or PTP, the 'ntp-sync' parameter is set to true as well.

OpusDec

The 'plc' parameter, standing for Packet Loss Concealment, tries to compensate for lost packets, and minimize the audible distortions from such a case.

AlsaSink

As will be discussed in the upcoming sections modifications to the available slave methods have been done, in order to achieve a better result in resynchronization. The 'drift-tolerance' is not relevant in this proposed improvement either.

The 'sync' parameter however is relevant. This as it states whether the AlsaSink should playout a sample as soon as possible, or if the corresponding timestamp should be used for determining the playout time. As samples need to be played out at the correct point in time in order to achieve inter-destination synchronization, this configuration is enabled. If samples would be played out as soon as possible, they could for example be outputted too early, if they would arrive at the sink quicker than the expected latency along the pipeline.

5.3 Proposed solution for resynchronizing

As have been discussed in Section 3.3 there are problems present with both the 'Skew slaving' and the 'Resample slaving' methods built into GStreamer, why an improved solution is proposed. This proposed solution tries to combine the existing two alternatives and take the advantages of each of them.

What this means is that first of all interpolation for slaving the audio clock to the system clock is not used. Instead the approach used in the 'Skew Slaving' algorithm is chosen which continuously will slave the clocks to each other. Also, it will, just as the original algorithm, align the segments after the clock slaving to make sure that no glitches between two segments are present.

The difference lies within the resynchronization step. Here resampling is used instead of allowing the glitch to occur which 'Skew Slaving' usually would have carried out to get back in sync. But in contrast to the 'Resample Slaving' method the correction is carried out over multiple segments instead of just one, which makes it less audible that resampling actually is occurring. Thus when the drift has become big enough, segments will start to be partly modified in order to get back in sync. At the same time the altered segment will be aligned to the next one to ensure that no glitches ever are present. This is carried out until enough modifications of the segments have been made for the clocks to be back in sync with each other. A parameter, much like the drift-tolerance parameter, is used to determine the upper threshold for the drift.

5.3.1 Problems and possible improvements

The resample algorithm

Although the proposed solution solves a few of the problems with the two other alternatives, problems are still present and improvements can be made. One of the most serious problems is that the resample algorithm built into GStreamer is still used, which simply adds or deletes samples as explained earlier. This will introduce spikes or other artifacts just as it does when the 'Resample Slaving' method of GStreamer uses it.

In contrast to 'Resample Slaving', the proposed solution does try to improve this by having a few unmodified segments between the modified ones in order to make the effects of the resampling as small as possible. It also tries to modify the segments as little as possible to ensure that the problems with too aggressive resampling will not occur. This typically means that a segment only will have a maximum of three samples added or deleted to it, where each segment will contain 960 samples in the setup used in the thesis. Thus there will only be a $3/960 = 0.3\%$ change of a segment at most. Although this is slightly audible at times when focusing on trying to find artifacts in the sound, it is still a lot better compared to the standard 'Resample Slaving' method. But as stated, the actual resampling algorithm is still a problem and should be improved in order for even higher quality playback.

Risk of increasing drift

Another problem is that there is a risk that the drift is not corrected, but grows larger over time given that the drift is increasing too rapidly. The longest test that has been carried out during the thesis was running around 60 hours, where it was able to maintain the synchronization. However, this was under good conditions on an unloaded network, and as soon as load and delay will be present the drift typically may be worse leading to that the algorithm may not correct the drift as fast as it occurs.

Moreover, what should not be forgotten is that there still is a mechanism that will force a resynchronization given that the drift has grown large enough. This is a feature in the original implementation of GStreamer, which simply will do a hard jump to the correct playback time when a certain threshold has been reached. The threshold is determined by a parameter much like the 'drift-tolerance' parameter. Thus a safeguard for a fast increasing drift is partly present, but as can be expected it is highly unwanted to use such a mechanism. The reason is that a hard jump for certain will introduce an audible glitch in the playback. However, for safety reasons it should still be available since unexpected scenarios may occur with a high load on the network.

Soundstage alteration

Yet another problem which is not solely a problem for this method, is the fact that as soon as a drift which is large enough starts to get corrected, and if this is done fast enough, the sound stage will be heard to be altered. This is of course not wanted given that the speakers stand right next to each other. Solving this problem is harder on the other hand, since if the drift is not allowed to be large before it is corrected, the correction will occur often, leading to audible artifacts due to the resampling algorithm used. Furthermore, as argued in Section 3.3, correcting it too often may be counterproductive since it just may be temporary, correcting itself with the next segment. This means that a balance needs to be found between how much drift that is tolerated before it needs to be corrected, and how fast the correction should be made in that case. As this can be seen to be a matter of personal choice, the tweaking of parameters is left to the user.

Resynchronize during silence

Looking at a possible improvement instead, one such example could be to have a more aggressive resynchronization taking place during periods of silence. Even a hard jump may be possible in this case since it will not be heard given that it will not jump straight into the actual audio of the stream.

This of course introduces a requirement on the stream however, namely that silence must be present. If this is not the case continuous modifications or resampling must be done in order to keep it at 40 ms, which itself is on the verge of being interpreted as being out of sync.

Given that silence is present, using this approach may of course prove to be advantageous. However, if a better resample algorithm is introduced, continuous resampling might not be audible, and thus nothing is gained resynchronizing during silence. This as the resynchronization will not be heard, in either case.

Choice of elements

Finally it can also be mentioned that the choice of elements and more importantly, the parameters used in the elements in GStreamer, may affect the synchronization and the drift. Therefore it cannot be certain that the chosen elements and parameters are the optimal ones, although research has been carried out to try to find the best ones possible.

Chapter 6

Experiments

6.1 General test setup

A number of tests have been carried out in order to identify weaknesses of the current prototype, as well as on a whole identify possible risks in such a system. Below the two main use cases and the corresponding focuses for the system are defined. The term 'delay' in the use cases refers to the time between the input of audio on the server – from which the streaming is controlled – until the audio is output on the audio jack on the receivers, to which speakers can be connected. Both use cases have synchronized playback as a primary factor.

Music Streaming In the case of music streaming the main focus is to achieve high quality audio without interruption, where the delay should be kept down, but is of lower priority.

Live speech in the same room In the case of live speech, where the person speaking can hear the audio output and thus his or her own voice, the main focus is to have as low delay as possible. The sound quality is of lower priority.

The general equipment used in the test cases is listed below. Equipment specific for a test case will be specified under that test case as a complement to the general equipment.

6.1.1 Equipment

Hardware

- 1 x Windows 7-PC (Media server)
- 1 x Raspberry Pi Model B+ (RPi#1)
- 2 x Raspberry Pi 2 Model B with HiFiBerry DAC+ (RPi#2, RPi#3)
- 1 x NetEm-PC (NetEm [58])
- 1 x ProCurve Switch 1800-8G (Switch)
- 1 x LeCroy waveRunner 44 MXi 400 Mhz Oscilloscope 5 GS/s

Software

Windows server

- The prototype written in .NET
- The time service (W32Time), which synchronizes the computer over NTP, will be disabled during the tests. It is run right before the start of them though, in order to have the correct time on the computer. The reason for this is that the W32Time service resulted in instability in the synchronization at the clients. This might have been due to that the server was locked to an NTP server different from the one of the clients.

Raspberry Pi:s

- Debian 7.8, Linux version 3.18.11-v7+
- GStreamer 1.4.5
- NTP: ntpd 4.2.6p5 (when applicable)
- PTP: ptpd 2.1.0-debian-1-2 (when applicable)

6.1.2 Network

The general network setup can be found in Figure 6.1. If not otherwise stated, NetEm will not modify any packets.

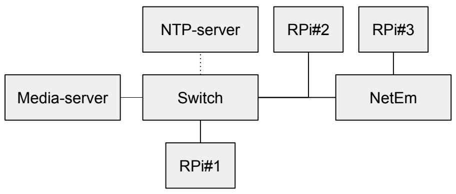


Figure 6.1: The general network setup used in the test cases.

6.2 Summary of test cases

A brief summary explaining the different test cases which will be carried out can be found in Table 6.1. For a more detailed description, please refer to the corresponding section.

6.3 1. Synchronization on unloaded networks

6.3.1 Objective

The objective is to determine the actual difference in playback time between two clients, on an unloaded network. This can be done by measuring on the soundcard outputs, and thus determine the extent of successful synchronized playback.

ID	Title	Explanation
1.1-1.3	Synchronization on unloaded networks	These tests determines the actual difference in playback time between two clients, on an unloaded network. Tests using both NTP and PTP will be carried out.
2.1-2.3	Synchronization on loaded networks	These tests introduce asymmetric delays on the network. This in order to examine the synchronization under loaded network conditions.
3.1-3.5	Packet loss on networks	These tests aim to determine the the effect of packet loss in terms of audio distortions, stream recovering, and how well they can be concealed by Opus.
4.1-4.6	Delay - total time between time of capturing and time of playback	These tests will determine the actual delay from the time of recording the audio, to the time of actual playback.
5.1-5.2	Drift between system clocks	These tests look at the drift behaviour when two system clocks can drift freely, as well as when one of them can drift freely while the other one is controlled by NTP.

Table 6.1: The different test cases which will be considered, with a brief explanation of each category of them.

6.3.2 Setup

The setup can be seen in Figure 6.2.

6.3.3 Execution

By measuring on the output phone jacks of the Raspberry Pi:s with an oscilloscope, the actual difference in synchronization between the two devices can be measured. Any differences in delay caused by connected speakers are thus disregarded. The difference can be determined by first letting the Raspberry Pi:s playback an audio signal, where the period T of the signal should be twice as large as the maximum difference that should be possible to determine. Then the time difference between the corresponding peaks of the waves from the two devices can easily be measured. In the tests carried out a simple square wave form will be used.

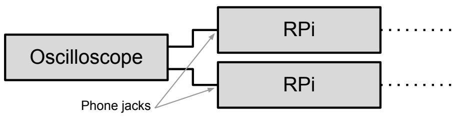


Figure 6.2: The setup for measuring the difference in synchronization between two devices.

6.3.4 Variations

The synchronization may depend on values on certain parameters in GStreamer. A few examples of such are the drift-tolerance parameter, as well as whether the system clocks are synchronized using NTP or PTP. Thus a few different tests will be carried out in order to determine the achievable synchronization, on an unloaded network. Both the proposed algorithm, and the algorithm provided by GStreamer will be tested to make sure that the proposed algorithm does not affect the synchronization in a negative way. Below a short description of each of these test cases can be found:

ID 1.1: Synchronization during no network load with NTP, using proposed solution

The following environment will be used:

- NTP service enabled.
- PTP service disabled.
- The devices use an NTP server available on the network.
- GStreamer: slave-method is the proposed solution.

ID 1.2: Synchronization during no network load with PTP, using proposed solution

The following environment will be used:

- NTP service disabled.
- PTP service enabled.
- The devices use RPi#1 as PTP server.
- GStreamer: slave-method is the proposed solution.

ID 1.3: Synchronization during no network load with PTP, using 'Skew Slaving'

The following environment will be used:

- NTP service disabled.
- PTP service enabled.

- The devices use RPi#1 as PTP server.
- GStreamer: drift-tolerance parameter set to 1 μs.
- GStreamer: slave-method is 'Skew Slaving'.

6.3.5 Results

The results can be found in Table 6.2. Please note that both the average difference and the maximum difference can be negative values as well. The values can therefore appear a bit misleading, as a perfect oscillation between -2 μs and 2 μs would result in an average difference of zero. Using an absolute value of the average difference is more common, why extra care should be taken when evaluating these results. This may explain any unexpected values of the standard deviation.

All the tests were run for approximately 18 hours.

ID	1.1	1.2	1.3
Avg. diff.	31.21 μs	2.04 μs	3.56 μs
Max. diff.	1550.13 μs	143.13 μs	909.23 μs
Std. dev.	515.61 μs	18.53 μs	58.21 μs

Table 6.2: The average difference in synchronization, the largest difference measured, and the corresponding standard deviation for the three test cases. Please note that both the average difference and the maximum difference can be negative values as well.

6.4 2. Synchronization on loaded networks

6.4.1 Objective

The objective is to introduce asymmetric delays on the network, in order to examine the synchronization under loaded network conditions.

Delay on the network can affect both the playback and the synchronization. This is the reason there is a jitterbuffer present in the prototype, which will hide and remove any present jitter up to a configurable threshold. As long as a packet is not delayed more than the threshold, the successive elements will not experience the delay. If however the delay is greater than the threshold, it will simply be discarded by the jitterbuffer, and thus handled as any lost packet would be. These tests will therefore focus on examining how the synchronization is affected, as NTP is vulnerable to asymmetric delays. The delays will therefore not be constant, but randomized in such a way that the forward and backward network paths will differ in delay. Asymmetric delays may be more realistic in a real environment.

6.4.2 Execution

To simulate a connection with packet delay NetEm will be used. This is done by using a computer with two network interfaces, where the traffic is bridged between them, and NetEm affects the bridged traffic. One of the Raspberry Pi:s will thus be connected to one of the interfaces, while the other interface is connected to the switch. The other Raspberry Pi will receive an unaltered, complete, stream. The setup can be seen in Figure 6.1.

NetEm's delay feature accepts three arguments, namely a delay, a variation, and a correlation value. This allows a simulation of for example a delay of 100 ms $\pm$ 10 ms with a 25% correlation value, as in the case of packet loss. For each packet NetEm's random number generator will then randomize a delay in the interval of $[90, 110]$ ms, with a 25% dependency on the previous outcome.

The test will thereby compare the affect on the synchronization when one of the Raspberry Pi:s receive a randomly delayed stream of packets, while the other receives an unaltered stream, as seen in Figure 6.1. The evaluation of how well synchronized the two devices are will be carried out in the same way as previously described.

6.4.3 Variations

How well the prototype will handle delay is in contrast to packet loss, much dependent on the buffer size at the client. This as no buffer can protect against a lost packet, if no retransmission mechanism is in use, which is the case for the prototype.

The prototype uses a 1000 ms jitterbuffer size at the receiver side, which should help handle minor to medium delay on the network fine. As said previously these tests will focus on exposing the time protocol to asymmetric delays, why NTP will be enabled in all the cases. Due to time constraints corresponding PTP tests will not be possible to carry out.

The delays which have been chosen can be found in Table 6.3. A correlation value of 25% was used in all of the cases.

ID	**Delay (ms)**	**Variation (ms)**
2.1	50	± 5
2.2	100	± 10
2.3	500	± 50

Table 6.3: The different test cases which will be considered with their delay and variation.

6.4.4 Results

The results can be found in Table 6.4. Please note that both the average difference and the maximum difference can be negative values as well. The values can therefore appear a bit misleading, as a perfect oscillation between -2 μs and 2 μs would result in an average difference of zero. Using an absolute value of the average difference is more common, why extra care should be taken when evaluating these results. This may explain any unexpected values of the standard deviation.

All the tests were run for approximately 18 hours.

ID	2.1	2.2	2.3
Avg. diff.	28.00 ms	47.94 ms	232.30 ms
Max. diff.	50.80 ms	60.69 ms	487.30 ms
Std. dev.	7.99 ms	8.14 ms	33.46 ms

Table 6.4: The average difference in synchronization, the largest difference measured, and the corresponding standard deviation for the three test cases. Please note that both the average difference and the maximum difference can be negative values as well.

6.5 3. Packet loss on networks

6.5.1 Objective

The objective is to introduce packet loss in the stream and test

- how noticeable the resulting distortions are
- if the stream can recover after major packet losses
- if Opus' packet loss concealment (PLC) feature makes any difference

The tests are thereby to a great extent an examination of the stability of GStreamer and its ability to recover from lossy connections.

6.5.2 Execution

The execution will be the same as in test case 2.1 to 2.3, but instead utilizing NetEm's packet loss feature.

This feature for packet loss in NetEm accepts a value a for how many percent of the packets passing through that randomly should be dropped. As a second argument it accepts a correlation value c. This defines how much each probability $P(n)$ depends on the previous outcome $P(n-1)$, as in

$$P(n) = c \times P(n-1) + (1-c) \times rand(a) \tag{6.1}$$

where $rand(a)$ is NetEm's random number generator. It thereby allows simulation of bursts of packet losses, which is more realistic on the Internet. A reasonable value for this correlation value when simulating WANs could be 25%.

Finally the output, a 440 Hz test signal, from the Raspberry Pi that is exposed to the packet loss will be recorded in order to study the artifacts that may be introduced. The check of maintained synchronization will be carried out as in previous synchronization tests.

6.5.3 Variations

A few different cases will be tested in order to examine the behaviour during packet loss. For all cases the output audio will be recorded, synchronization will be checked, and the devices will have the NTP daemon enabled, if not otherwise stated. The different cases can be found in Table 6.5.

ID	Loss (%)	PLC	Comment
3.1	5	On	-
3.2	5	Off	To be able to compare the difference PLC may or may not do.
3.3	7.5	On	-
3.4	10	On	-
3.5	90	On	To examine if the stream can recover and regain the synchronization, when exposed to extreme packet loss.

Table 6.5: The different test cases which will be considered with their input values.

6.5.4 Results

Examples of how the distortions introduced by the losses looked can be seen in Figure 6.3, which shows one packet being lost, while PLC is on. Figure 6.4 on the other hand shows the case of multiple lost packets, also with PLC on. Finally Figure 6.5 shows a packet being lost while the PLC feature was turned off.

In the 'extreme loss' test, where 90% of the packets were dropped, the playback stopped, but the stream was still up during the whole test. When the packet drop was disabled again, without touching the stream, GStreamer managed after just a short while to once again play the stream. Moreover, it did so synchronized with the other Raspberry Pi which received an unaltered stream.

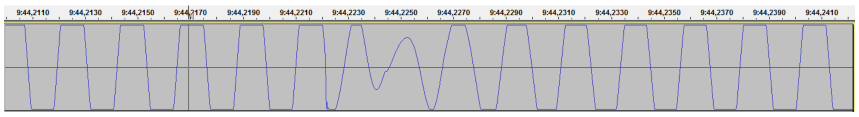


Figure 6.3: The distortion caused by one lost packet, with PLC on.

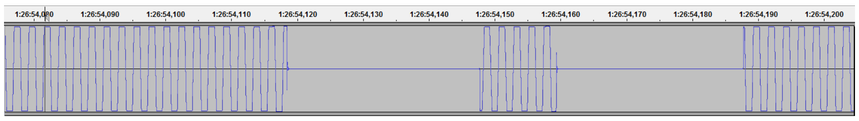


Figure 6.4: The distortion caused by multiple lost packets, with PLC on.

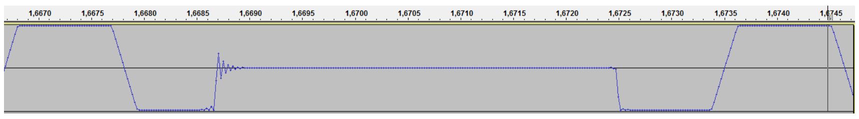


Figure 6.5: The distortion caused by one lost packet, with PLC off.

6.6 4. Delay - total time between time of capturing and time of playback

6.6.1 Objective

The objective is to determine the actual delay from the time of recording, catching the sound on the server, to the time of playback, the output from sound card on the client. This will include processing times at sender and receiver, as well as network propagation time.

6.6.2 Equipment

- 1 x Audio producing device/microphone

6.6.3 Setup

The setup can be seen in Figure 6.6.

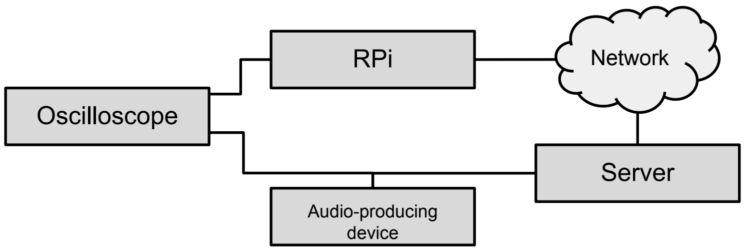


Figure 6.6: The setup used for measuring the total delay from recording to playback.

6.6.4 Execution

By measuring on the output of the audio producing device and the output phone jack of the Raspberry Pi with an oscilloscope, the actual difference in time between the two devices can be measured. Any differences in delay caused by the audio producing device and speakers are thus disregarded. The delay can be determined by first letting the audio producing device output an audio signal, where the period T of the signal should be twice as large as the maximum delay that should be possible to determine. Then the delay time difference between the corresponding peaks of the waves from the audio producing device and the Raspberry Pi can easily be measured. In the tests carried out a simple square wave form will be used.

6.6.5 Variations

The total delay is partly dependent on the buffer sizes used, since a buffer will introduce a corresponding delay. The two buffer sizes which will be modified in order to decrease the delay are in the RtpBin and AlsaSink element. This as they have two rather large buffers – 200 ms each – as a standard value, and the focus is to minimize the delay. This test case therefore relates to the 'live speech' use case, where a large delay is unwanted. The reason for this is that a delay may lead to an annoying echo effect, which may make it harder to give an announcement for example. In the case of just streaming music this is not as important, since a delay can be said to be acceptable as long as the audio itself is synchronized.

It is true that other buffers may be involved as well. However, these will not be considered in this test case since they can be said to be smaller, and thus not affecting the total delay to the same extent.

The case will be divided into two types – microphone and loopback. The microphone cases mean that a signal was produced by a second computer, which was connected to the microphone-input on the server, and thus simulating a microphone announcement. When using the loopback interface instead, an audio signal was played on the server. The played sound file containing the square wave was then captured by the server prototype software, as previously described in Section 5.1. The reason for testing both of the input devices is that the internal buffers of the two elements may differ, and thus result in different delays.

The different cases which will be considered can be found in Table 6.6.

ID	Input	RtpBin buffer (ms)	AlsaSink buffer (ms)
4.1	Mic	200 (default)	200 (default)
4.2	Mic	150	150
4.3	Mic	100	100
4.4	Loopback	200 (default)	200 (default)
4.5	Loopback	150	150
4.6	Loopback	100	100

Table 6.6: The different test cases which will be considered with their corresponding input and buffer sizes.

6.6.6 Results

The experiments were only carried out once due to time constraints. Thus they should be seen as more of a indication of what results may be achievable, although variation may exist. Despite the lack of multiple measurements it should not be forgotten that the results are sensible given the buffer sizes used. This is because more buffers are involved than the two focused on.

The results using the microphone can be found in Table 6.7, and the results using the loopback-interface can be found in Table 6.8.

ID	4.1	4.2	4.3
Tot. delay	643 ms	548 ms	430 ms

Table 6.7: The measured total delay in three cases when using microphone input.

ID	4.4	4.5	4.6
Tot. delay	480 ms	345 ms	250 ms

Table 6.8: The measured total delay in three cases when using the loopback interface.

6.7 5. Drift between system clocks

6.7.1 Objective

The objective is to briefly examine the drift behaviour among the playback devices. This to investigate whether the system clocks drift in the same direction when uncontrolled, and how much they drift from the correct time. This can give indications on whether the drift is continuous and stable, or differs over time and between devices, that have the same hardware and software.

6.7.2 Execution

By measuring on the output phone jacks of the Raspberry Pi:s with an oscilloscope, the actual difference in synchronization between the two devices can be measured. The difference can be determined by first letting the Raspberry Pi:s playback an audio signal where the period T of the signal should be twice as large as the maximum difference that should be possible to determine. Then the time difference between the corresponding peaks of the waves from the two devices can be measured. In the tests carried out a square wave form will be used as the signal.

By observing the start difference and the final difference, a total drift can be determined. In order to determine the difference between the system clocks, the 'Skew Slaving' method in GStreamer is used, with a drift tolerance of 1 μs. As the audio clock should follow the system clock as closely as possible, the drift tolerance should be as low as possible as well. This in contrast to using the proposed solution, as this one focuses on correcting a drift in a way that minimizes audible distortions. What this in turn means is that it may result in a larger difference between the system and audio clock than the 'Skew Slaving' method.

6.7.3 Variations

A short description of the two test cases can be found below.

ID 5.1: Drift of system clocks during no network load, no NTP/PTP

This test will focus on how two system clocks behave when they are free to drift in any direction, during a longer time period of about 18 hours. As the clocks should be able to drift freely, all time protocol daemons on both Raspberry Pi:s are disabled.

ID 5.2: Drift of system clocks during no network load, one NTP-synchronization

Both system clocks are free to drift in any direction, given that neither is controlled by any time synchronization protocol. This is the case in test case 5.1. They might therefore drift in the same way, and thus show a smaller drift than what each clock individually actually has experienced compared to the 'real' time. In this test the NTP daemon on one of the Raspberry Pi:s will therefore be disabled, while it is enabled on the other. This in turn allows a measurement of the drift from the 'real' time. The run time for this test will also be 18 hours.

6.7.4 Results

The experiments were only carried out once due to time constraints. Thus they should be seen as more of a indication of what results may be achievable, although variation may exist.

The results can be found in Table 6.9.

ID	5.1	5.2
Start diff.	0.44 ms	0.17 ms
End diff.	17.78 ms	46.48 ms

Table 6.9: The start difference and end difference between the devices.

Chapter 7

Discussion of Test Results

7.1 Synchronization on unloaded networks

First considering the result for test case 1.1, a low average value is revealed, if compared to the accuracy stated for NTP in the literature. This low average should be taken with a grain of salt however, as this is not an absolute value as shortly explained in the results of the tests. Therefore, oscillations between -20 ms and 20 ms would for example result in an average difference of 0 ms, why the average difference cannot solely be used to determine whether an acceptable synchronization was achieved. The average differences of all the tests can thus be seen as somewhat optimistic values, and thereby a bit misleading.

As oscillations are possible, this may also explain the rather high standard deviation of the tests. The maximum difference also verifies this. But even if the values are being optimistic, it can be said that they still are well below the threshold for what is perceived as synchronized though. This as the standard deviation still is well below the upper threshold of what may be acceptable.

Comparing the results of NTP with the results of PTP, test case 1.2, the average difference is only a tenth of the one for test case 1.1. As the results are in the tens of microseconds range, this matters less, as they are both below the thresholds for perceived synchronization.

Moreover, the standard deviation as well as the maximum difference are much lower, and clearly indicates a much more stable synchronization, compared to NTP. One aspect that might play a role in this better stability, is the fact that PTP per default has a more frequent process of packet exchanges. This might thereby more tightly control the clocks.

The third test, test case 1.3, also involved PTP, but used GStreamer's built-in 'Skew Slaving' resynchronization method. This test was designed to check that the proposed solution would not have affected the synchronization negatively. It was also because of this reason the lowest possible drift tolerance was chosen, as the goal was to compare the proposed algorithm to the best possible synchronization GStreamer can offer. The audio experience was thus not considered in this test.

As then can be seen from the results, the difference in synchronization between test case 1.3 and 1.2, where the proposed solution was used, is negligible. The average difference, as well as both the maximum value and the standard deviation, are all higher compared to test case 1.2. Especially a higher maximum difference was achieved, which probably was what affected the standard deviation.

7.2 Synchronization on loaded networks

While both NTP and PTP performed quite well on unloaded networks, as discussed in Section 7.1, it was interesting to test the performance in a more hostile environment. WANs would be an example of a such a hostile environment, where the previously mentioned asymmetric network paths may occur. These asymmetric paths are, as explained in Section 2.3.1, not possible to differentiate from the clock offset. The accuracy of NTP can therefore quickly degrade in these environments.

As explained in Section 7.1, the average difference may be misleading, which is also true in these test cases.

In the first test case, 2.1, the average difference of 28 ms does not exceed the upper threshold of 40 ms, as specified in Section 2.2.1, but it is not far from. Furthermore, with a maximum difference of about 50 ms and a standard deviation of about eight milliseconds, there will be many cases of the audio not being perceived as synchronized during this test.

Naturally, both the second and third test case, 2.2 and 2.3, also show a maximum difference that exceeds the upper threshold for synchronization. But now not even the average differences are within in the acceptable range. There are therefore very few use cases or environments where these differences would not be seen as completely unacceptable.

From these results it can be seen that the synchronization quickly degrades when load is put on the network and thus asymmetric delays are introduced. Although the values used are somewhat artificial, they give an indication of the impact variation in delay may have on NTP. This is a reason that it may be important to ensure that the receiving clients in a system, like the prototype, have a stable connection to the NTP server. The system might otherwise render useless, seen from a synchronization point of view.

7.3 Packet loss on networks

As can be seen from the results, Opus' PLC feature seems to be able to conceal the lost packets quite well, as long as not too many consecutive packets are lost. In the case of multiple consecutive loss of packets a significant distortion thus occurs.

It can also be seen that a larger distortion is present in the playback although only one packet is lost, when the PLC feature is off. The PLC feature can therefore be said to make a clear difference.

Finally it could be seen that the playback, but also the synchronization, could be regained even after a 90% packet loss. This means that GStreamer most likely always will be able to restore the playback and regain synchronization. Looking at the cases with less packet loss, it can thus be said that the synchronization most likely would be regained in these cases as well. Due to time constraints this could not be verified however.

7.4 Delay - total time between time of capturing and time of playback

The results show a major problem for the 'live speech' use case, as the total delay is considerably higher than the sum of the two major buffers, if the microphone input is used. This indicates that the element used, AutoAudioSrc, introduces a large buffer in contrast to the loopback source, where the total delay is considerably lower. Neither AutoAudioSrc nor the loopback source provide any public parameter to affect the buffer sizes, thus the delay cannot be further decreased by modifying the buffers of these elements.

The delay of the loopback source is also too high to make the 'live speech' use case feasible though. Pragmatic tests have shown that an upper limit of 50 ms can be tolerated, and even when using small buffer values this limit is exceeded significantly. Even lower values of the buffers cannot be used either, as a certain amount of buffering is needed for the elements to work properly. Instability in the pipeline was present already when having a size of 100 ms of the AlsaSink buffer. Thus it would prove impossible to achieve the upper limit of 50 ms, at least by using the current setup. Therefore no further test cases with more variations of buffer sizes were carried out.

7.5 Drift between system clocks

The fact that the total drift is twice as high in the second test case compared to the first one may indicate that the system clocks of the two devices drift in the same direction. This should not be seen as surprising, as the same hardware is used.

As test case 5.2 aims to determine the total drift of the system clocks, it is also interesting to see how the result compares to typical values discussed in Section 2.4. According to this test result the Raspberry Pi with cheap hardware would perform better than a typical computer crystal. Moreover, it is far off from the value determined by Bergsma in [49], where Raspberry Pi:s also were used. The reason for this may be that Bergsma performed much more thorough testing and evaluation, but also comparing it to a better clock source. In comparison, test case 5.2 did not include a proper clock source to compare to, such as a GPS-based clock. Instead the two Raspberry Pi:s were compared against either, which may be the reason that this result was achieved. These test results should therefore rather be used as an indication of how the end result output wise could be, under the given circumstances.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

8.1.1 Evaluation phase

There are a number of available streaming protocols, where newcomers – compared to UDP and TCP – such as DCCP provides features specifically targeted at the thesis' context. Although such a protocol in theory would be a good choice, maybe even a better one than UDP or TCP, it can be concluded that it is not mature or widespread enough to actually be a viable option.

It was also seen that GStreamer was not the culprit of how accurate synchronization could be obtained. Instead this was dependent on the hardware of the clients, in conjunction with the accuracy of the time protocol of choice, such as NTP or PTP.

As all clocks will drift sooner or later, the core of the thesis has been how to handle the resynchronization of them in an audio context. This means that as few audible artifacts as possible should be introduced. The usage of GStreamer partly simplified the resynchronization part since two different alternatives already were implemented. However, these two each had issues, either by introducing glitches, or altering the pitch in an aggressive manner.

After having examined the existing alternatives and pinpointed the shortcomings of each, a proposed algorithm was implemented. This tried to combine the two alternatives by using the strengths of each one, and eliminate the issues. The result was an algorithm that resamples the audio by duplicating or removing samples from it, but tries to do so as smoothly and slowly as possible, in order not to introduce more artifacts than necessary. Examinations of the proposed algorithm showed considerably smaller and less audible distortions, compared to GStreamer's built-in alternatives.

8.1.2 Test phase

During the test phase multiple tests were carried out in order to determine how well the prototype worked. The first considered aspect was the tightness of the synchronization on unloaded networks, using both NTP and PTP. The results showed a synchronization well below any threshold for what is perceived as synchronized, although PTP made it possible for the clocks to be more accurately synchronized compared to using NTP, leading to a tighter synchronized playback. Also, comparing the proposed algorithm with one of the existing ones in GStreamer revealed that the improvements had not degraded the synchronization, while still providing a better audio experience.

When asymmetric delays were introduced on the network on the other hand, NTP struggled to maintain an acceptable synchronization. This result was partly expected as it is known that NTP does not work well on asymmetric networks, that is, the forward and backward networks paths differ in delay. Thus it may be necessary to ensure a stable connection between the clients and the NTP server, in order to maintain a synchronized playback.

Examining the effects of packet loss instead it could be seen that by using Opus' Packet Loss Concealment (PLC) mechanism, the packet losses could partly be concealed. However, when enough number of packets were lost consecutively, audible gaps were introduced. The playback was always resumed by GStreamer though, even at packet loss rates of 90%.

8.2 Future work

The thesis has involved many different parts stretching from the general problem of finding a proper streaming protocol, to a more fine-grained level actually implementing a resynchronization algorithm specific to GStreamer. There is thus naturally a lot that can be done in the field in order to improve the proposed prototype. The upcoming list is therefore far from exhaustive, and the interested reader may find even more areas on how the work can be improved.

8.2.1 Implement DCCP stack on Windows

An open-source implementation of DCCP on Windows may help to spread the protocol so that it gains more recognition. Moreover, if an extension to GStreamer would be written as well, supporting DCCP, it may be interesting to see how much better it will perform on networks where it is supported.

8.2.2 Implement a better resample algorithm in GStreamer

A major problem with the current solution of the resynchronization method is that it will only duplicate or remove samples when it tries to resample in order to resynchronize. Since this is a naive method it will introduce artifacts as previously explained, and the method could be improved considerably if a proper resample algorithm would be introduced. What should be kept in mind however, is that since it should run on embedded devices with limited computing power it needs to be efficient as well. Nonetheless, finding an improvement of the current algorithm which is naive should be fully possible.

8.2.3 Introduce same improvements to NTP as proposed by Microsoft

Microsoft was able to achieve a much better accuracy with the time protocol. This was done by for example polling more often, and calculating a mean of multiple

RTTs instead of only one, as done in the original implementation of NTP. It could be interesting to see whether the proposed prototype in the thesis would maintain an even more accurate synchronization if this was introduced, and if so, how much better.

8.2.4 More exhaustive testing of NTP

Scientific tests on how well NTP performs on for example asymmetric networks are scarce to this date, at least publicly available. It would therefore be interesting to see how well the test results in this thesis conforms to more studies.

8.2.5 More exhaustive testing of PTP

Due to time constraints tests to see how well PTP handled delay on asymmetric networks were not carried out. It could thus be interesting to see how much better, or maybe worse as some studies show [44], the results would have been in comparison with NTP. Also more general tests of the behaviour over WANs might be interesting.

8.2.6 Further examination of IDMS using RTP/RTCP

Since it is newly proposed it may be interesting to look at the advantages of using the proposed standard of using extensions of RTCP to achieve inter-destination media synchronization as introduced in [53]. Evaluations of it has partly been made, and even integration with GStreamer with excellent results focusing on video has been implemented. But even further evaluations of how it performs in a more general context could be interesting to see.

References

[1] Rolling Stones Live on Internet: Both a Big Deal and a Little Deal [Online]. Available: http://www.nytimes.com/1994/11/22/arts/rolling-stones-live-on-internet-both-a-big-deal-and-a-little-deal.html (accessed: 2015-08-22)

[2] P. Davidsson. (2014, April). Bredbandskollen – Surfhastighet i Sverige 2008-2013. The Internet Foundation in Sweden. Stockholm, Sweden. [Online]. Available: https://www.iis.se/docs/Bredbandskollen_surfhastighet_i_Sverige_2008-2013.pdf (accessed 2015-07-15)

[3] Sonos. http://www.sonos.com (accessed 2015-07-12)

[4] [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981.

[5] [RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980.

[6] B. A. Forouzan, *Data Communications and Networking.* New York, NY: McGraw-Hill, 2007.

[7] [RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, DOI 10.17487/RFC3550, July 2003.

[8] [RFC4340] Kohler, E., Handley, M., and S. Floyd, "Datagram Congestion Control Protocol (DCCP)", RFC 4340, DOI 10.17487/RFC4340, March 2006.

[9] F. Boronat, J. Lloret, M. García, "Multimedia group and inter-stream synchronization techniques: A comparative study", *Information Systems*, vol. 34, pp. 108-131, March 2009.

[10] GStreamer Open Source Multimedia Framework [Computer Software]. (2015). Retrieved from http://gstreamer.freedesktop.org (accessed 2015-07-05)

[11] [RFC2663] Srisuresh, P. and M. Holdrege, "IP Network Address Translator (NAT) Terminology and Considerations", RFC 2663, DOI 10.17487/RFC2663, August 1999.

[12] R. Stewart, P. D. Amer. (2007, September). Why is SCTP needed given TCP and UDP are widely available?. Internet Society. Virginia, USA. [Online]. Available: https://www.isoc.org/briefings/017/ (accessed 2015-08-15)

[13] [RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007

[14] Yuan-Cheng Lai, , Chang-Li Yao , Performance comparison between TCP Reno and TCP Vegas, Computer Communications Volume 25, Issue 18, 1 December 2002, Pages 1765–1773.

[15] [RFC4341] Floyd, S. and E. Kohler, "Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 2: TCP-like Congestion Control", RFC 4341, DOI 10.17487/RFC4341, March 2006.

[16] [RFC4342] Floyd, S., Kohler, E., and J. Padhye, "Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC)", RFC 4342, DOI 10.17487/RFC4342, March 2006.

[17] I. S. Chowdhury, J. Lahiry, S. F. Hasan, "Performance analysis of Datagram Congestion Control Protocol (DCCP)", in *12th International Conference on Computers and Information Technology*, Dhaka, Bangladesh, 2009, pp. 454-459.

[18] M.A. Azad, R. Mahmood, T. Mehmood, "A comparative analysis of DCCP variants (CCID2, CCID3), TCP and UDP for MPEG4 video applications", in *Information and Communication Technologies. ICICT '09. International Conference on*, 2009, pp. 40-45.

[19] [RFC5597] Denis-Courmont, R., "Network Address Translation (NAT) Behavioral Requirements for the Datagram Congestion Control Protocol", BCP 150, RFC 5597, DOI 10.17487/RFC5597, September 2009.

[20] M. Schier, M. Welzl, "Using DCCP: Issues and Improvements", in *20th IEEE International Conference on Network Protocols*, Austin, TX, 2012, pp. 1-9.

[21] [RFC6773] Phelan, T., Fairhurst, G., and C. Perkins, "DCCP-UDP: A Datagram Congestion Control Protocol UDP Encapsulation for NAT Traversal", RFC 6773, DOI 10.17487/RFC6773, November 2012.

[22] L. Melo de Sales, H. Oliveira, A. Perkusich, A. Carvalho de Melo, "Step two in DCCP adoption: The Libraries", in *2009 Linux Symposium*, Montreal, QC, 2009, pp. 239-250.

[23] Jonsson, Lars, Jonsson, Mathias, "Streaming audio contributions over IP". EBU Technical Review, 2008.

[24] [RFC5762] Perkins, C., "RTP and the Datagram Congestion Control Protocol (DCCP)", RFC 5762, DOI 10.17487/RFC5762, April 2010.

[25] [RFC6716] Valin, JM., Vos, K., and T. Terriberry, "Definition of the Opus Audio Codec", RFC 6716, DOI 10.17487/RFC6716, September 2012.

[26] [RFC1889] Audio-Video Transport Working Group, Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", RFC 1889, DOI 10.17487/RFC1889, January 1996.

[27] [RFC3551] Schulzrinne, H. and S. Casner, "RTP Profile for Audio and Video Conferences with Minimal Control", STD 65, RFC 3551, DOI 10.17487/RFC3551, July 2003.

[28] [RFC3611] Friedman, T., Ed., Caceres, R., Ed., and A. Clark, Ed., "RTP Control Protocol Extended Reports (RTCP XR)", RFC 3611, DOI 10.17487/RFC3611, November 2003.

[29] [RFC4585] Ott, J., Wenger, S., Sato, N., Burmeister, C., and J. Rey, "Extended RTP Profile for Real-time Transport Control Protocol (RTCP)-Based Feedback (RTP/AVPF)", RFC 4585, DOI 10.17487/RFC4585, July 2006.

[30] [RFC4588] Rey, J., Leon, D., Miyazaki, A., Varsa, V., and R. Hakenberg, "RTP Retransmission Payload Format", RFC 4588, DOI 10.17487/RFC4588, July 2006.

[31] Vorbis specification. Retrieved from https://xiph.org/vorbis/doc/Vorbis_I_spec.html (accessed 2015-07-20)

[32] [RFC7587] Spittka, J., Vos, K., and JM. Valin, "RTP Payload Format for the Opus Speech and Audio Codec", RFC 7587, DOI 10.17487/RFC7587, June 2015.

[33] [RFC5215] Barbato, L., "RTP Payload Format for Vorbis Encoded Audio", RFC 5215, DOI 10.17487/RFC5215, August 2008

[34] Hydrogenaudio, Results of the public multiformat listening test @ 64 kbps (March/April 2011) Retreived from: http://listening-tests.hydrogenaud.io/igorc/results.html (accessed 2015-07-20)

[35] H. Wallach, S. College, E. B. Newman, M. R. Rosenzweig, "The precedence effect in sound localization", *The American Journal of Psychology*, vol. 62, pp. 315-336, Jul. 1949.

[36] T. Blank, B. Atkinson, M. Isard, J. D. Johnston, K. Olynyk, "An Internet Protocol (IP) Sound System", *Audio Engineering Society Convention 117*, Oct. 2004.

[37] [RFC5905] Mills, D., Martin, J., Ed., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, DOI 10.17487/RFC5905, June 2010.

[38] *5. How does it work?*. Avilable: http://www.ntp.org/ntpfaq/NTP-s-algo.htm (accessed 2015-07-08).

[39] D. L. Mills. (2012, May 12). *Executive Summary: Computer Network Time Synchronization* [Online]. Available: http://www.eecis.udel.edu/ mills/exec.html (accessed 2015-07-08).

[40] NTP Software Downloads [Online]. Available: http://www.ntp.org/downloads.html (accessed 2015-07-13)

[41] *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, IEEE Standard 1588-2008, 2008.

[42] PTPd [Computer Software]. (2015). Retrieved from http://ptpd.sourceforge.net/ (accessed 2015-07-10)

[43] PTPd Default Configuration [Computer Software]. (2015). Retrieved from http://sourceforge.net/p/ptpd/code/HEAD/tree/trunk/src/ptpd2.conf.default-full (accessed 2015-07-10)

[44] A. Novick, M. Weiss, K. Lee, D. Sutton, "Examination of time and frequency control across wide area networks using IEEE-1588v2 unicast transmissions", *Frequency Control and the European Frequency and Time Forum (FCS), 2011 Joint Conference of the IEEE International*, San Fransisco, CA, 2011, pp. 1-6.

[45] T. Kovácsházy, B. Ferencz, "Performance evaluation of PTPd, a IEEE 1588 implementation, on the x86 Linux platform for typical application scenarios", *Instrumentation and Measurement Technology Conference (I2MTC), 2012 IEEE International*, Graz, AT, 2012, pp. 2548-2552.

[46] LinuxPTP [Computer Software]. (2015). Retrieved from http://linuxptp.sourceforge.net/ (accessed 2015-07-10)

[47] T. Nunome, S. Tasaka, "An Application-Level QoS Comparison of Inter-Destination Synchronization Schemes for Continuous Media Multicasting", *Global Telecommunications Conference, 2003*, vol. 7, pp. 3602-3608, Dec. 2003.

[48] H. Marouani, M. R. Dagenais, "Internal clock drift estimation in computer clusters", *Journal of Computer Systems, Networks, and Communications*, vol. 2008, pp. 1-7, Jan. 2008.

[49] R. Bergsma, "How accurately can the Raspberry Pi keep time?", 2013 Retreived from: http://blog.remibergsma.com/2013/05/12/how-accurately-can-the-raspberry-pi-keep-time/ (accessed 2015-07-19)

[50] Snapcast [Computer Software]. (2015). Retrieved from https://github.com/badaix/snapcast (accessed 2015-07-24)

[51] Advanced Linux Sound Architecture (ALSA) [Computer Software]. (2015). Retrieved from http://www.alsa-project.org (accessed 2015-07-09)

[52] M. Rautiainen et al., "Swarm synchronization for multi-recipient multimedia streaming", in *IEEE International Conference on Multimedia and Expo*, New York, NY, 2009, pp. 786-789.

[53] [RFC7272] van Brandenburg, R., Stokking, H., van Deventer, O., Boronat, F., Montagud, M., and K. Gross, "Inter-Destination Media Synchronization (IDMS) Using the RTP Control Protocol (RTCP)", RFC 7272, DOI 10.17487/RFC7272, June 2014, <http://www.rfc-editor.org/info/rfc7272>.

[54] M. M. Climent, "Design, development and evaluation of an adaptive and standardized RTP/RTCP-based IDMS solution", Ph.D. dissertion, Dep. De Comunicaciones, Universitat Politècnica De València, València, 2015.

[55] GStreamer C# Bindings [Computer Software]. (2015). Retrieved from http://gstreamer.freedesktop.org/modules/gstreamer-sharp.html (accessed 2015-06-15)

[56] NAudio [Computer Software]. (2015). Retrieved from https://naudio.codeplex.com (accessed 2015-06-15)

[57] GLib [Computer Software]. (2015). Retrieved from https://developer.gnome.org/glib (accessed 2015-06-15)

[58] NetEm [Computer Software]. (2015). Retrieved from http://www.linuxfoundation.org/collaborate/workgroups/networking/netem (accessed 2015-08-16)

[59] Sonos Patent, "System and method for synchronizing operations among a plurality of independently clocked digital data processing devices". Retrieved from http://www.pat2pdf.org/patents/pat8234395.pdf (accessed 2015-08-23)