MASTER'S THESIS | LUND UNIVERSITY 2020

Analysing the Audio Latency Contributions in a Networked Embedded Linux System

Gabriel Sjöberg
Emil A. Hammarström

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX: 2020-01

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-01

Analysing the Audio Latency Contributions in a Networked Embedded Linux System

Gabriel Sjöberg, Emil A. Hammarström

Analysing the Audio Latency Contributions in a Networked Embedded Linux System

Gabriel Sjöberg
gabriel.sjoeberg@gmail.com

Emil A. Hammarström
emil.a.hammarstrom@gmail.com

January 17, 2020

Master's thesis work carried out at Axis Communications AB.

Supervisors: Arvid Nihlgård Lindell, arvid.nihlgard.lindell@axis.com
Adam Nilsson, adam.x.nilsson@axis.com
Karl Rikte, karl.rikte@axis.com
Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

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

Abstract

Low latency audio is one of the most important aspects of audio systems designed for human interaction. Voice over IP (VoIP) and public announcement are examples of such systems where high audio latency can prevent regular operation.

In this thesis, we present our investigation into how streaming of low latency digital audio performs on two embedded Linux systems with different CPU architectures. To quantify the performance, we introduce latency measurement methods that build upon previous work and utilize existing methods to measure digital and analog audio latency. We also compare the resource usage of the two architectures, measured during testing.

We conclude that alternative scheduling algorithms may improve the performance of the multi-core system and that currently, the single-core outperforms the multi-core architecture. We also see how network traffic characteristics can severely affect system performance.

Keywords: MSc, Audio latency, Network, GStreamer, ALSA, Linux, Embedded

Acknowledgements

We would like to thank Axis Communications for hosting this master's thesis. Furthermore, we would like to thank our supervisors at Axis, Adam Nilsson, Arvid Nihlgård Lindell, and Karl Rikte whose comments have been invaluable to us. We would also like to extend our thanks to Jonas Skeppstedt for his meticulous review of the thesis. A special thanks to Danny Smith who has contributed with all his extensive knowledge of GStreamer and the hardware platform.

Contents

Acronyms

ARM Advanced RISC Machine.

CCTV Closed-Circuit Television.

CFS Completely Fair Scheduler.

DNS Domain Name System.

EDF Earliest deadline first.

FIFO First-In-First-Out.

GPOS General-purpose operating system.

I/O Input/Output.

IoT Internet of Things.

IP Internet Protocol.

LAN Local Area Network.

MIDI Musical Instrument Digital Interface.

MIPS Microprocessor without Interlocked Pipelined Stages.

MP3 MPEG-1 Audio Layer III or MPEG-2 Audio Layer III.

NIC Network Interface Card.

OS operating system.

PCM Pulse-Code Modulation.

PoE Power over Ethernet.

QoS Quality of Service.

RISC Reduced Instruction Set Computing.

RMS Rate Monotonic Scheduling.

RR Round-robin.

RTOS Real-time operating system.

SSRC Synchronization sources.

TCP Transmission Control Protocol.

TCP/IP Internet protocol suite.

TDCS Time Deterministic Cyclic Scheduling.

UDP User Datagram Protocol.

VoIP Voice over IP.

Glossary

ALSA Advanced Linux Sound Architecture (ALSA) is the sound framework used by the Linux kernel to provide API access to installed sound cards. It provides sampled audio and MIDI functionality.

Architecture A Dual-core CPU based on the Microprocessor without Interlocked Pipelined Stages (MIPS) architecture.

Architecture B Single-core CPU based on the Advanced RISC Machine (ARM) architecture, with a higher clock frequency than architecture A.

Architecture X Generalized description of architecture A and B.

Audio pipeline The GStreamer pipeline which sends an audio stream between two systems using RTP sessions. It connects in both ends to an ALSA interface, allowing practically any application to send audio over it.

Black-box A term generally used to describe a system in which you have no insight into the inner workings but where you can measure the inputs and outputs.

GStreamer Is a pipeline-based multimedia open source framework. It is written in the C programming language using the GObject type and object system and is supported on many operating systems. Bindings exist for many other programming languages.

mDNS Multicast Domain Name System (mDNS) provides DNS-like resolution in the absence of a managed DNS server. A multicast query to resolve a domain-to-IP question is responded to by the device with the specified domain, if accessible.

NTP Network Time Protocol (NTP) is a network protocol for inter-system clock synchronization. It is used for clock synchronization on the internet where it can maintain sync within tens of milliseconds. Additionally it can also be used for local synchronization between systems which we intend to do, there it can maintain sync to sub one millisecond.

Port mirroring A great tool in troubleshooting and monitoring network traffic between devices. What it does is mirror the package streams on one port to another. By doing this a monitoring entity can insert itself between two parties without having to disrupt the usual path through the network.

RTCP RTP Control Protocol (RTCP) resides at the application layer and is used to control RTP streams. It provides some out-of-band statistics as well as control information about its RTP stream, such as an absolute timestamp from the sender to synchronize the RTP stream.

RTP Real Time Protocol (RTP) resides at the application layer of the TCP/IP stack. It is designed for audio and video data delivery. The protocol typically runs over UDP and is used in conjunction with RTCP.

SNMP Simple Network Management Protocol (SNMP) is a network protocol which can be used both to monitor and to configure systems remotely. The protocol gives the ability to poll for system statistics during runtime. This information is then transmitted back to the requesting party in a standardized fashion.

vmstat virtual memory statistics (Vmstat) is a computer system monitoring tool that collects and displays summary information about operating system (OS) memory, processes, interrupts, paging and block I/O. [21].

White-box A term generally used to describe a system in which you have insight into the inner workings as well as the inputs and outputs.

Wireshark Wireshark is a packet capture software. It allows the user to monitor NICs on a computer and inspect all the traffic passing there. The package streams can be filtered and interpreted in real time. It is a great tool when developing networked applications as it provides an insight of what actually ends up on the line and can be a great way to sanity check ones implementations.

Chapter 1
Introduction

Digital streaming audio has in recent years become the mainstream form of music consumption, likely due to its convenience and the accessibility of vast digital music libraries. However, digital audio does have more advantages. It can, for example, be transferred using physical media, like a CD, or through computer networks. Axis Communications have created a networked audio product to utilize existing computer networks to transfer audio digitally between their audio devices. In older public announcement (PA) systems, loudspeakers are usually wired to a central point where all amplifiers are situated, requiring new cabling to be deployed each time a new speaker is installed. Axis Communications instead connects their speakers using standard Ethernet cables with PoE to power them as they have built-in amplifiers. As these systems are supposed to replace the old standard analog PA systems, the new digital implementation at a minimum must have parity in functionality and performance. At the moment of writing, the digital system is subject to a significant audio latency between the audio source and speaker element. We aim to analyze this audio latency to support further decision-making.

1.1 Axis Communications

For the past 20 years, Axis Communications have worked on the networked video segment. They offer thermal cameras, 4K cameras, RADAR, and access-control solutions. Recently they have also entered the audio market with their networked audio systems. The primary use cases for these products are security announcements, music playback, and public announcements.

1.1.1 Networked Audio Systems

Axis' networked audio system aims to provide a flexible and scalable audio solution with several use cases. Achieving this entails streaming multiple audio streams from several

devices to many speakers, with synchronized playback.

Users may play music from different sources, either by analog input or digital music streaming services. Analog inputs can be from both a music player or a microphone. There are also capabilities of playback from the device locally. Live or scheduled announcements may interrupt current streams according to user-specified priorities.

In use cases such as broadcasting live public announcements in a store or a similar location, it is preferable to achieve a low latency audio stream. It ensures that the announcer is not confused or stressed by the delayed auditory feedback [23].

The system is based on Axis hardware running an embedded Linux system. This enables compatibility with existing performance monitoring and open-source software.

1.2 Problem Description

Live playback of human speech can be demanding as the human auditory system is very latency-sensitive. Aubrey Yates [23] described this as early as 1963; "It has also long been known (Cherry & Sayers, 1956) that interference with the natural relationships between ongoing speech and the consequent feedback of information could lead to severe disturbances in the smooth progress of speech …"

In tests performed by Michael Lester and Jon Boley on delayed audio monitoring for musicians [11], they find that for vocal monitoring a latency of just $55\,\mathrm{ms}$ is perceived as "Bad" while a lower latency of 10 to $25\,\mathrm{ms}$ is considered "Good" to "Fair". To reach the level of "Excellent", a latency of $3\,\mathrm{ms}$ or less was needed.

In the current networked audio solution from Axis Communications, the latency for audio from an analog input is around $120\,\mathrm{ms}$ when in close proximity to the destination speaker. The latency becomes problematic for one of the most central use cases for the system, live announcements, due to the effects described above.

Goals

The goals of this thesis are to identify the significant latency contributors and try to mitigate these. We also want to provide a comparison of how different hardware architectures handle audio streaming using the current audio pipeline which is described in Section 2.2.1 under *Logical setup*.

These latency contributors may stem from network-related issues such as packet buffering and network jitter, or other unforeseen characteristics. One of our goals is, therefore, to measure these fundamental causes of increased latency in modern audio systems [22]. The latency contribution in the systems will be the primary focus for this thesis since the placement of a product, and its network, is not well defined for all installations. It is, however, of interest to know how a device behaves on a busy network. We will, therefore, include network characteristics in our analysis.

The first goal is to establish an environment and test suite where audio latency, audio quality, network activity, and system activity may be monitored. We want to test the audio pipeline, monitor the resources of the operating system, and monitor network characteristics; all of which may impact the propagation of real-time audio data.

The second goal is analyzing the gathered statistics for any latency hot-spots. Statistics of interest are buffering delay and propagation delay and their effect on the underlying system.

Research questions

Given the goals described, our research questions (RQ) and supplementary questions (SQ) are as follows:

RQ: What are the significant contributors to latency in an over-the-network audio pipeline and to what extent can these be mitigated?

RQ: How does buffering in a GStreamer [6] pipeline, or a lack thereof, affect the total latency of the pipeline?

SQ: How do other streaming frameworks affect system latency?

Scope

It is possible to create large connected systems using the networked audio system. To keep the complexity down, we decided early on to work on a minimal working example. This setup is described in Section 2.2.1.

We have narrowed down our scope to software related issues as that is our main area of expertise. Some effort has gone into discussing the impact of processor core count and other hardware characteristics based on our results, but it has never been the main focus. We have tried to understand the impact of scheduling, concurrency, and context switching from a software perspective. There are hardware aspects of these, such as the hardware overhead of context switching and scheduling. These aspects have been left out of scope.

In our initial goals, we expressed an interest in trying out different frameworks other than GStreamer. The intention was to see if the problems originated from within the default framework. During testing, it became clear that the issues originated both within the architecture and the framework. Due to the inherent complexities of the system, we decided to allocate more time to testing and analyzing the GStreamer framework and not substituting it for other frameworks.

Initially, our scope was set on the architecture A systems exclusively, described in Section 2.2.1, as these systems were perceived to be slower. The goal was to find solutions to the latency problem on this architecture or present good reasons to migrate to another architecture.

After performing extensive testing, we saw the need to expand the investigation to another architecture to provide a comparison. Architecture B, described in Section 2.2.1, was selected as it represented a possible upgrade to architecture A. Finally, the results from our tests were compared to try to find out architecture-specific characteristics and what impact the framework had on system resources.

1.3 Related Work

Here we present related work and describe how it ties into our work and how we have been able to utilize their theories and discoveries. We will also try to build upon these and extend them as needed to fit our research.

1.3.1 Latency evaluation

In this thesis, we try to evaluate the latency of the networked audio system. We have utilized the following previous work to understand the possible latency contributors better and how we can measure them.

AvCloak

AVCloak: A Tool for Black Box Latency Measurements in Video Conferencing Applications is a paper by A. Kryczka et al. [10], describing methods for evaluating a black-box VoIP system's latency. They propose constructing a timestamp to be transmitted over an encoded communication channel. This timestamp can then be extracted on the receiving end and compared to the current time to find the difference representing the latency.

In the case of an audio-only transfer, they encode the timestamp in audio frequencies sent over the audio channel much like in today's signal transfer techniques over radio frequencies but in the audible spectrum.

We have some differing conditions, mainly that we are working in a white-box environment with full insight into and the possibility to modify the system. We like the idea of sending an encoded timestamp over the communication channel, and with access to the inner workings of the system, we can develop it further. Doing this may allow us to read said timestamps at several points in the system instead of just at the ends of the channel.

Low Latency Audio Processing

Yonghao Wang, at the time of writing this thesis a senior lecturer at Birmingham City University, did his Ph.D. thesis in the area of low latency digital audio processing [22] at the Queen Mary University of London. The thesis delves deep into the subject of handling digital audio and the transmission thereof.

The thesis covers a broader scope than what we will be able to cover. It brings up and discusses the differences between General-purpose operating systems (GPOSs) and Real-time operating systems (RTOSs). GPOS are true to their name and support many workflows and do so with mostly satisfactory results. Low latency audio is characterized by small amounts of data that are passed around frequently with hard deadlines. For this, a GPOS may not be optimal, given its inherent limitations of thread-scheduling under heavy load. To facilitate the quick flow of data in a digital audio system, all processes in that system must be made aware of the time deterministic and cyclic nature of digital audio. They must yield in relation to these cycles to not obstruct any audio deadlines. To be able to yield, different scheduling algorithms can be used. Wang mentions two; Rate Monotonic Scheduling (RMS) and Earliest deadline first (EDF). He also suggests a new scheduling

algorithm for RTOSs, which would take into account the time deterministic nature of digital audio and its intrinsic need for strict deadlines. He calls it Time Deterministic Cyclic Scheduling (TDCS) and presents simulation results showing how it could outperform the two previously mentioned algorithms.

In much the same fashion, he also suggests a hybrid approach to IP networks and how to handle the same time deterministic tendencies in such a best-effort environment. He proposes two classes of traffic with real-time data handled in a time deterministic fashion, much like what optical fiber networks do with predetermined time-slots in all link frames. Implementing this would reserve some amount of the link capacity to the real-time stream, ensuring timely pass-through at all times no matter the network congestion. He prefers this over what the current standard of QoS can provide. He shows that as the congestion in an IP-network increases, its ability to honour QoS decreases, culminating in significant traffic delays that disrupt the flow of the real-time stream.

1.4 Contributions

This thesis provides improvements of existing latency measurement methods and confirms previous bodies of work in an environment of embedded devices. We have built upon the base ideas introduced in the *AVCloak*-paper. By inserting timestamp metadata into a data stream, we have created a tool to visualize the timeline of element events in a GStreamer pipeline. Our results agree with those of Y. Wang [22], a previous body of work in the area of low latency digital audio streaming. In contrast to his work, performed on desktop workstations and purpose-built hardware, our results provide an insight into the GPOS case of low latency digital audio on resource-limited embedded systems.

1.4.1 Contribution statement

We have somewhat divided the work based on our areas of expertise. The design of the tests was collaborative. Gabriel presented the implementation idea of the intra-pipeline test. Emil is the more comfortable C-coder and therefore provided invaluable knowledge through the collaborative coding process. Gabriel had previously worked with GStreamer and computer networking, and Emil had previous knowledge of system profiling. Since GStreamer is part of the foundation of this thesis, it was an helpful pre-knowledge to have as it allowed us to stake out our course early on. The plotting methodology and plotting code was also collaborative and was improved upon many times. Gabriel had additional responsibility for design and readability. Overall we have taken the more abstract decisions together through discussion. As we have different areas of expertise, we have naturally contributed more to those areas. Testing has sometimes been divided between the authors to get more things done in a shorter period, Emil has done more of the digital testing and Gabriel more of the analog and network testing.

When it comes to the report, it is again a collaborative effort. The structure is based on the template provided by the Department of Computer Science at LTH. Writing has been an iterative process where both authors have written and then proofread each other's work during the writing process. Overall we estimate that we have spent an equal amount of time on the report.

Chapter 2
Approach

In this chapter, we introduce the major areas relevant to this thesis. We also introduce our research methods used throughout the thesis.

2.1 Theory

Here we introduce the theory of both analog and digital audio as this lies at the center of the thesis. We briefly cover process scheduling in GPOSs and the different scheduling algorithms we have used. Then, we will cover the GStreamer framework, as the audio pipeline is built using it. Finally, we will cover the Advanced Linux Sound Architecture (ALSA), as the test setup runs Linux and therefore uses ALSA for low-level audio handling.

2.1.1 Audio

Analog sound can be seen as vibrations in air or any other medium. Such vibrations can be represented as sinusoidal waves in continuous time, as seen in Figure 2.1. When we want to transform analog signals into the digital domain, we first have to perform an analog-to-digital conversion. Conversely, a digital-to-analog conversion needs to be performed in order to play the audio back again.

Analog signals

Below are two equations, Equations (2.1) and (2.2) [15]. Equation (2.1) describes a basic sinusoidal signal, $x(t)$ in Figure 2.1. Equation (2.2) describes a delayed counterpart, $x(t - T_d)$, in Figure 2.1.

$$x(t) = A\sin(2\pi Ft - \Phi) \tag{2.1}$$

$$x(t - T_d) \tag{2.2}$$

A	Amplitude
F	Frequency in Hz
Φ	Phase in rad
T_d	System Latency in seconds

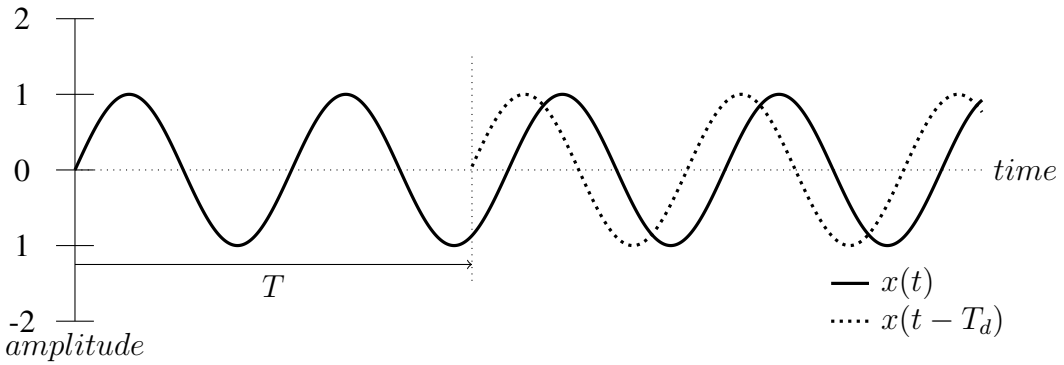


Figure 2.1: Standard sinusoidal wave.

Digital signals

Here follow some explanations of important terminology for digital signal representation.

Time determinism

When performing the jump from analog to digital audio, there is a natural jump from continuous to discrete time, where each sample is deterministically placed relative to each other in time. This interval will not change as long as the sample rate is maintained. In order to playback the sampled audio, the same rate will be needed for playback, the *playback rate*.

Audio Latency

Audio latency refers to the period between the playback of an audio source to the perception of that audio source. E.g., from pressing the play button on a music player until hearing the music in the headphones.

Buffering delay is the delay that is introduced when audio data is waiting for a certain amount of data to accumulate before propagating it further down the audio pipeline.

Propagation delay is a delay in the time of moving data further down a pipeline, the time spent between buffers.

2.1.2 General-purpose operating systems

Process scheduling

Most modern operating systems are what are called multi-programmed systems. They may run multiple processes in what seems parallel to the user.

The scheduler will share the processing resources of the underlying hardware to achieve concurrency. If the underlying hardware contains multiple physical cores, work may be distributed to run in parallel in separate contexts. Otherwise, it is distributed to a single core and achieves concurrency via context switching. Context switching is the act of interrupting a process with the means of resuming it at a later time. Choosing which process to schedule a time slot for, also known as a *time slice*, is done using a *scheduling algorithm* [18].

The Linux kernel version run on architecture A and B supports the scheduling algorithms First-In-First-Out (FIFO), Round-robin (RR), and Earliest deadline first (EDF), for real-time applications. Completely Fair Scheduler (CFS) is the default scheduling algorithm when a real-time algorithm has not been applied.

Completely Fair Scheduler is the default scheduling algorithm in Linux following kernel version $2.6.23$ [16]. CFS uses a tree structure where the runtime of each node is tracked. The tree is ordered such that the leftmost branches of the tree have spent the least amount of CPU-time. The task runtime is incremented at a rate relative to its priority after running [18].

Round-robin will schedule work with a fixed time slice. All applications receive an equal time slice, but the time slice may finish processing earlier by yielding or being blocked by I/O [18]. The process will be preempted after the time slice has expired, and a context switch will start running the next process in the queue.

The Linux implementation of "Round-robin" also allows prioritizing processes. Prioritizing will enable another method of preemption, a process of a higher priority round-robin queue will preempt lower prioritized round-robin processes. However, preemption of a higher prioritized process will not move the process to the end of its priority queue. The process may resume the duration of the time slice that is left when returned to [16].

When selecting the time slice interval for the Round-robin algorithm, we have to consider the time spent context switching. If a context switch takes $1\,\mathrm{ms}$, a time slice interval of $5\,\mathrm{ms}$ will have a $20\,\%$ overhead. While an interval of $100\,\mathrm{ms}$ only has an overhead of $1\,\%$.

Earliest deadline first is a scheduling algorithm that accounts for a process's time until its deadline. In other words, a process with an earlier deadline will have a higher priority. The process will have a deadline for each period that it has been given, shown in Figure 2.2.

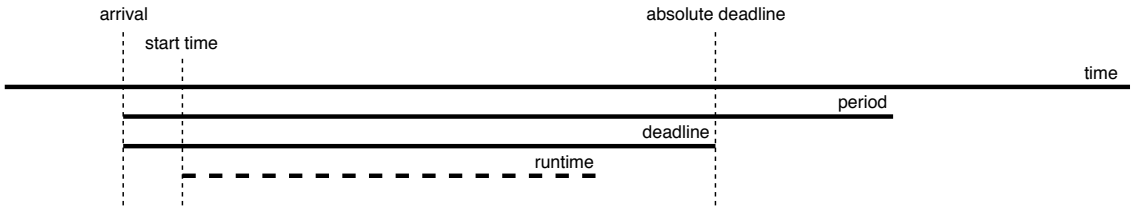


Figure 2.2: A process runtime, deadline, and period are shown in time.

The implementation in Linux consists of assigning a process a *runtime*, *deadline*, and *period*. Runtime is the time guaranteed within a given period, before the specified deadline. If a set of scheduling properties and processes are not schedulable, the scheduling will fail [16]. These property restrictions can be seen in Figure 2.2 and are defined as $runtime \leq deadline \leq period$.

It is worth noting that the deadline scheduling implementation on Linux does not allow forking of processes, and thus is not a suitable initial scheduling algorithm if a program spawns more child processes. Additionally, a process that yields will not be given further processing time until the following period after the yield.

Process sleeping A process may go to sleep in the two modes `TASK_INTERRUPTIBLE` and `TASK_UNINTERRUPTIBLE`. In an un-interruptible sleep, the process is guaranteed to sleep at least the specified amount, leaving the ready-to-run queue, excluding the scheduling latency of the context switch. The interruptible task may resume earlier and is not moved off the ready-to-run queue. It may wake up to respond to signals such as `SIGKILL` and many more [9]. Both sleep modes will hand over processing resources to another process.

A scheduled timeout will sleep a specified amount of jiffies. It also adheres to the interrupt states mentioned above when yielding [19]. Jiffies are a kernel-internal value counting timer interrupts. The timer interrupt rate is platform-specific and specified in hertz. E.g., a timer interrupt rate of $1000\,\mathrm{Hz}$ will result in a jiffy of $1\,\mathrm{ms}$. It is also the sleep resolution [2].

2.1.3 ALSA

The Advanced Linux Sound Architecture (ALSA) is the Linux kernel sound framework that provides an API for interacting with the sound card device drivers. It consists of multiple audio sources and sinks, both virtual and physical. A physical ALSA source interface may correspond to an analog-to-digital converter, sampling analog audio. These samples are buffered for a duration of time, called periods, to ease system load. If the system has support for stereo sound, there is also an additional stream of samples for the second channel. These channel samples can be interleaved or sequential.

Sample A sample in ALSA corresponds to what previously has been described as a digital audio sample. A quantized value captured at a sufficiently high sample rate, in our case 16-bit samples at a rate of $48\,\mathrm{kHz}$.

Frame A frame in ALSA is one sample per channel, in our case 16-bits (2 bytes), for all channels at a specific point in time. The frame is a grouping of samples from different channels. In our system, we have two channels. Meaning a frame size of $2 * 2 = 4$ bytes.

Period time A period time specifies the duration of sound that is retrieved from an ALSA sound device. It can be specified in microseconds or bytes. A smaller period time intuitively leads to more periods per second, as a lower duration per audio period necessitates more periods to fill a second. A period time of $T_{period} = 20\,\mathrm{ms}$ would mean $\frac{1000\,\mathrm{ms}}{20\,\mathrm{ms}} = 50$ periods/s while $T_{period} = 5\,\mathrm{ms}$ would mean $\frac{1000\,\mathrm{ms}}{5\,\mathrm{ms}} = 200$ periods/s potentially causing a higher load on the system.

The maximum and minimum period time is decided by the device driver [1].

Buffer time A buffer time is the size of the audio buffer. If period time is the duration of sound retrieved, then buffer time is the maximum duration of sound that ALSA buffers for extraction at any given point in time. It is generally a good idea to ensure that the buffer time is a multiple of the period time. Using this, one can provide some leeway for higher priority system interrupts, which may delay the retrieval of sound. This would otherwise cause the audio to cut out momentarily.

Loopback devices are supported by ALSA by enabling a loopback soundcard [12]. It provides a full-duplex loopback soundcard. The loopback soundcard will perform no audio conversion between the in- and output device, it is only copied [19].

Reading and writing audio data may be configured to be blocking or non-blocking. Using the blocking behaviour will result in the process changing its state to `TASK_INTERRUPTIBLE` and sleeping for an ALSA period time in jiffies. The non-blocking behaviour is to return with a status signal.

Thus using the blocking behaviour will introduce the resolution latency of the jiffy, described in Section 2.1.2, but allow other processes or threads to be scheduled.

2.1.4 GStreamer

GStreamer [6] is an element-based multimedia framework for transporting data. The data may be encoded, decoded, muxed, demuxed, created, or modified. A component in high-level terms consist of a source and a sink. In between these, the data may be modified.

Pipeline

A pipeline is composed of one or multiple elements, and the simplest form of a pipeline is a single element. A pipeline must begin in a source and end in a sink element. The pipeline has a property called *latency*, which can be set to specify what latency is desired. The pipeline will then attempt to match this latency. If the actual latency of the pipeline is shorter than the specified, additional buffering is performed to achieve that latency. If the actual latency is longer than the specified, the pipeline will do its best to decrease the latency to match the specified amount in a best-effort manner.

Element

Elements are the base components of GStreamer. The element is a fundamental building block in the pipeline. It may be composed of zero or more inputs and zero or more outputs, which are called pads. An element may modify or inspect data, but may also consume and propagate data to consumers external to the pipeline.

An example of an element is *lamemp3enc*, which takes raw PCM audio as its data source and produces MP3 encoded audio as its output. An element that consumes data and passes it to e.g., a file or network buffer, may only have a sink pad. A source-pad-only element may be one that reads bytes from an external hardware buffer and supplies the pipeline with those bytes. There are also cases where many-to-one sink and source pad configurations are needed, such as multiplexing several audio and video streams into one.

Pads

An element taking input and producing output will have a sink pad, and a source pad [3]. These pads construct the interface of the element, being its input (sink) and output (source).

Connecting two elements $A \rightarrow B$ consists of connecting the source pad of element A with the sink pad of element B. This also entails matching the capabilities of the sink and source pads.

Capabilities Describing the data flow through a pad is done by the capabilities of the pad [3]. Two connecting pads will need to have matching capabilities. It is possible to negotiate capabilities dynamically. An example of a capability may be bit rates that a sink pad will accept from a source pad or the format of the data which will be transferred.

Probes Probes may be placed on element pads to execute callbacks on subscribed events [4]. Such an event may be a data buffer passing by. These callbacks may modify or inspect data and metadata, such as buffer duration and buffer timestamps.

Bin The composition of many elements. Similar to mathematical composition, $bin(x) = (f \circ g \circ h)(x)$ where f, g, h are GStreamer elements.

AppSrc An application source is an element where the user can provide an external data stream. It may be used to provide audio, video, or other data streams from custom sources or constructed user sources.

AppSink An application sink provides a custom exit point in the pipeline where a user may define how to handle the pipeline's resulting data stream. Be it saving to a proprietary storage solution or adhering to a custom protocol.

AlsaSrc Given an ALSA capture device, *alsasrc* can extract an audio stream from it. The audio stream can then be transmitted through the pipeline once defined by its capabilities.

The properties `latency-time` and `buffer-time` of AlsaSrc are equivalent to `period-time` and `buffer-time` in ALSA.

An ALSA device may be read in blocking or non-blocking mode. Since more time is spent waiting for an audio buffer, it may be desirable to use blocking mode such that the thread may yield processing power to other threads.

The parent element of `AlsaSrc` provides two (2) algorithms for handling timestamping samples from ALSA: *re-timestamp* and *skew*.

Re-timestamp will ignore timestamps provided by ALSA and re-timestamp with the current time of the GStreamer pipeline clock [5].

Skew will introduce a post-ALSA buffer that will compensate for drifts in the retrieved audio segments. This is done by updating the propagated timestamp and shifting to the appropriate audio segment [5]. This technique introduces buffer delay, thus increasing the total latency of the pipeline.

AlsaSink Acts as a pipeline sink to an ALSA device that consumes an audio stream in the format specified by e.g., a `CapsFilter`.

The `AlsaSink` element will always open an ALSA device in a non-blocking mode.

CapsFilter Used to filter element capabilities between elements to ensure a more strict specification of what capabilities can be negotiated. The configuration needs to be a subset of the capabilities of the two elements. If no `CapsFilter` is specified, the two elements will negotiate the format based on their full capabilities.

Queue A queue is used to create a new thread for the pipeline following the queue's source pad. We can think of it as splitting the pipeline and creating two sub-pipelines which can execute concurrently.

AudioConvert Converts raw audio between the formats specified by the source and sink pads. Such as integer to float representation, width and depth, endianness, dithering, and many more conversions.

RtpL16pay and RtpL16depay Packs and unpacks raw audio for an RTP-packet.

RtpSession Manages the RTP session, keeping track of participants and validating sequence numbers [7, 17].

RtpStorage Packet storage used to recover from packet loss using an encoded error correcting-code [8].

RtpJitterBuffer A buffer where received packets are reordered and duplicates are dropped. It is possible to allow for retransmission if a packet is lost, this may not be viable at lower latencies.

RtpSsrcDemux Demultiplexing of RTP-packets, used when the stream contains several synchronization sources.

InputSelector Forwards one (1) of N input streams to its sink pad, given a selection of stream-properties.

MultiUDPSink UDP packet sink to multiple clients, which provides RTP streaming when combined with RTP-payload encoders.

RtpBin Is GStreamers implementation of RTP session handling. It contains an instance of GstRtpSession, GstRtpSsrcDemux, GstRtpJitterBuffer, and GstRtpPtDemux in the same bin element. By configuring one of these at each end of an RTP connection: session handling, package reordering, and audio syncing are handled automatically from within it.

Funnel Acts like a funnel from many to one stream, much like the name suggests. It does not synchronize the streams but instead forwards them as soon as they arrive.

RtpPtDemux Demuxes RTP packets based on the *payload type*. It allows an application to receive and decode different payload types in an RTP stream.

2.2 Method

There are many parts of the system used in our tests, and several things to test to get a clearer picture. In order to gather this information, we have devised several tests to ensure coverage of different system metrics, which we deem might be of interest.

2.2.1 Test environment

To ensure reproducibility between tests, we created a testing environment. The idea was to have two separate environments with different network characteristics. Further along in the testing, a new hardware architecture was evaluated, and the corresponding hardware was exchanged in the test design. The system we are evaluating can be scaled up to many devices. We decided to narrow down our test environment to two devices, to find improvements that may be applied on a device-to-device scale while avoiding the complexity of a more extensive system. The test setup overview, with a generalized design, can be observed in Figure 2.3.

Architecture X denotes architecture A or architecture B interchangeably throughout our thesis. The architecture A device is equipped with a dual-core CPU that is based on the Microprocessor without Interlocked Pipelined Stages (MIPS) architecture, and a NIC supporting $100\,\mathrm{Mbit/s}$. Architecture B is provided with a single-core Advanced RISC Machine (ARM) CPU with a higher clock rate than that of architecture A, and a NIC supporting $1\,\mathrm{Gbit/s}$.

During all tests, unless noted otherwise, we run the RR scheduling algorithm. The time slice differs between the two architectures; architecture A has a time slice of $25\,\mathrm{ms}$ per thread, and architecture B a time slice of $100\,\mathrm{ms}$.

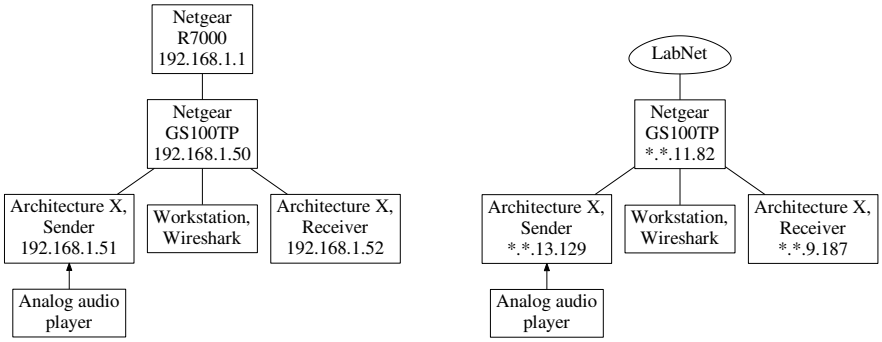
Physical setup

The physical setup is a connected system of two (2) architecture X devices as seen in Figure 2.3. One device will send an audio stream, and the other will receive said audio stream. The devices are connected using a Netgear GS110TP network switch.

The silent setup in Figure 2.3a is used to get well-defined measurements from our tests. The noisy setup in Figure 2.3b is similar to a worst-case scenario of what a busy network could look like interfering with the low latency traffic.

Test setup A can be seen in Figure 2.3a using a local network without any non-purpose traffic.

Test setup B can be seen in Figure 2.3b using the development subnet at the company with lots of stray control and discovery traffic from several similar systems.



(a) Silent network.

(b) Noisy network.

Figure 2.3: The two test setups used to test noisy vs. silent networks.

Logical setup

Once there is a physical test setup in place, and the devices have been configured, the *audio pipeline* can be set up. As mentioned before, the system uses GStreamer to do this. The general architecture of the pipeline can be seen in Figure 2.4. An explanation of all the individual elements utilized in the pipeline can be found in Section 2.1.4. A more detailed description of the full audio pipeline follows.

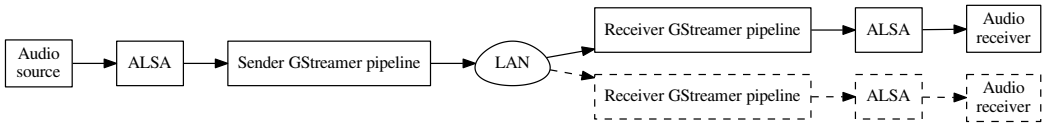


Figure 2.4: The logical structure of the audio pipeline. One sender with one or more receivers on different systems.

Pipeline setup

On the sending architecture X system, a program referred to as the `Audio Sender` will run a GStreamer pipeline, sending audio to a receiving architecture X device. The receiving architecture X device will run a program referred to as the `Audio Receiver`, which will run the receiving end of a GStreamer pipeline. The pipelines are set up as follows.

The Sender GStreamer pipeline can be observed in Figure 2.5a. The sender part of the audio pipeline begins with an interface to ALSA, an *AlsaSrc*. It is followed by a *Queue* to separate the pipeline into two threads. Then follows a CapsFilter to dictate capabilities between the *Queue* and the next element.

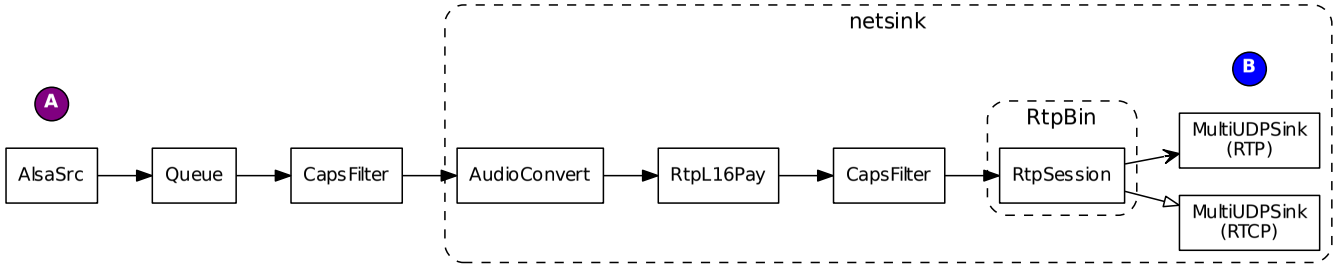
After the *CapsFilter*, there is a *bin* called *netsink*. It contains an *AudioConvert*, followed by an *RtpL16Pay*. Then there is yet another *CapsFilter* to negotiate capabilities.

Inside the *netsink*, the *CapsFilter* connects to a second bin called *RtpBin*. The *RtpBin* then connects to two separate *MultiUDPSink*s, one for RTP and one for RTCP traffic.

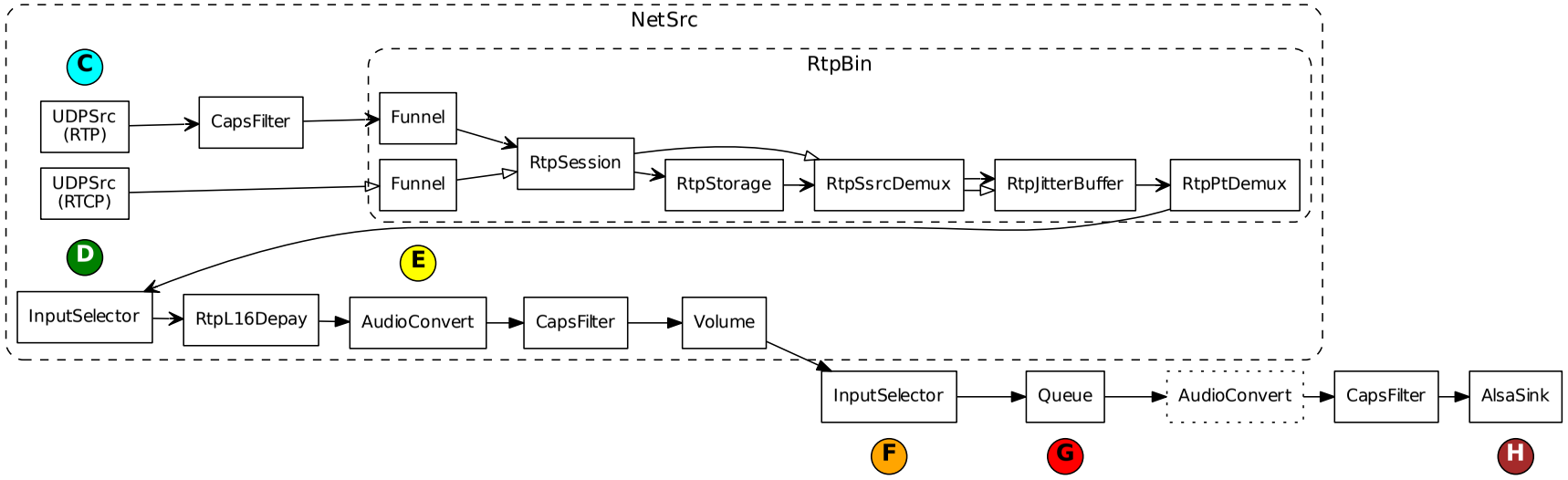
The Receiver GStreamer pipeline found in Figure 2.5b contains more elements due to the session handling, packet reordering, and audio synchronization. It consists of two nested bins and some individual elements. *NetSrc* is the largest bin and contains the elements as follows: It receives two network streams through two separate *UDPSrc*s, one for RTP and one for RTCP. The RTP stream continues to a *CapsFilter*, which the RTCP stream passes by, both then land in one *Funnel* each. Both streams then converge in an *RtpSession*. From the *RtpSession*, the RTP stream passes to an *RtpStorage*. They then enter an *RtpSsrcDemux* followed by an *RtpJitterBuffer*, here the RTCP stream reaches its end, and the RTP packets are reordered as needed. The RTP stream continues through an *RtpPtDemux*, an *InputSelector*, and finally, an *RtpL16Depay* mirroring the payloader in the sender.

Much like in the sender, the audio then passes through an *AudioConvert* and a *CapsFilter* to then land in a *Volume* element. Following this, is another *InputSelector* followed by a *Queue* which separates the pipeline into two threads.

Initially, the system we analyze had an additional *AudioConvert* following this queue, but we found that it did no conversion since there was a preceding *AudioConvert* handling any transcoding. Thus we removed it. A *CapsFilter* follows, and finally, an *AlsaSink* to hand over the audio to ALSA for playback.



(a) Audio Sender GStreamer pipeline.

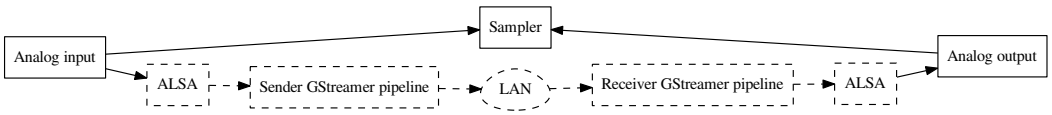


(b) Audio Receiver GStreamer pipeline.

Figure 2.5: GStreamer audio pipelines. V-shaped arrows represent RTP-traffic and hollow arrows represent RTCP-traffic. Elements are marked with rectangular nodes and bins with dashed edges and rounded corners.

2.2.2 Full system latency

(a) Digital whole-system latency measurement.



(b) Analog whole-system latency measurement.

Figure 2.6: The two test setups used to test pipeline latency.

To measure the full system latency between two time-synced systems, we will interleave timestamps in the audio data. Two timestamps t_A and t_B will be collected at point A and B respectively. The timestamps will then be compared resulting in a latency of $\Delta t = t_B - t_A$. Point A, in this case, is represented by `latsend` in Figure 2.6a and *analog input* in Figure 2.6b. Point B is represented by `latrecv` in Figure 2.6a and *analog output* in Figure 2.6b. The full system latency may synonymously be referred to as the system latency, or the total system latency.

Digital measurement

In the AVCloak [10] paper, a black-box latency test showed timestamps encoded as frequencies in the audio stream. The final latency is calculated as outlined above.

In this system, a raw data stream is used to send audio data. This allows a bit-stream to be sent over the pipeline without intermediate transformations. This property will be used to repeatedly send a collection of bytes in the format of a preamble to detect a packet, a timestamp used in the latency calculation, and an index to provide traceability in the pipeline and over the network.

A receiving application is also written to receive the manufactured stream. The objective of this application is to read the inserted timestamps at the receiving system, and compare these with its system clock.

Listings 2.1 and 2.2 provide the initial idea of the sender and receiver in pseudo-code, and the final implementation details are found in Sections 2.3.2 and 2.3.3.

Before this provides reliable results, system clock synchronization must be achieved between the two systems. To ensure this, we will use NTP to synchronize the systems.

```
procedure latsend(index) {
  t0 = get_current_timestamp()
  time_data = { 0xFFFF, index, t0 }
  sound_data = read_sound_source()
  new_sound_data = piggyback(time_data, sound_data)
  send_to_alsa_buffer(new_sound_data)
  latsend(index + 1)
}
```

Listing 2.1: Latsend pseudo code.

```
procedure latrecv() {
  buffer = get_alsa_buffer()
  ptr = find_preamble(0xFFFF, buffer)
  t0 = parse_timestamp(ptr)
  t1 = get_current_timestamp()
  record(index, t0, t1)
  latrecv()
}
```

Listing 2.2: Latrecv pseudo code.

Analog measurement

A logic analyzer is used to perform an analog latency measurement. The result is used to confirm the digital latency measurements. If the digital measurements are confirmed to be reliable, they are used for further analysis. The measured sinusoidal wave is produced on a workstation and provided via an analog interface to the sending system. Producing the sound using a third-party solution ensures that we do not introduce any additional system load to the embedded system.

System latency measurement To perform the full system measurement, we add two hardware probes to the physical system. One at the *analog input (A)* and one at the *analog output (B)*, as can be seen in Figure 2.6b. These are used to find the system delay, as described in Section 2.1.1. Where A measures $x(t)$, Equation (2.1), and B measures $x(t - T_d)$, Equation (2.2).

As the logic analyzer records all the probes simultaneously in time, we can compare the collected data and extrapolate the *system delay (T)* that our system introduces. The extrapolation is done by measuring the time difference between the initial signal appearance on both probes. The quality of the signal can also be evaluated using the same measurement data. We do this by visually inspecting the wave shape and frequency reproduction at B, and comparing it to the input at A.

Audio quality While the perception of audio quality in many cases can be a subjective measurement, we have adopted a trivial but objective measurement. We will observe the system's output signal in the measurements and compare it to the input, both visually and using signal analysis. The signal analysis will be performed using the logic analyzer tool. It outputs information, such as signal amplitude and frequency. Good audio will be a resulting sine wave with the same frequency as that of the input. The amplitude may differ

as the different points of measure may operate at different voltage levels. It is beneficial doing this visually instead of aurally as it allows us to be more precise. Small disruptions in the signal will be more noticeable in these measurements than by merely listening.

Resource usage During the analog testing, resource usage such as interrupts, context switches, kernel time, and user time are measured once per second. The measurements are retrieved by sampling the *procfs* filesystem throughout the length of the test.

This data may provide insight for which resource usage levels the propagation of audio fails to deliver in a time deterministic manner.

2.2.3 Intra-pipeline latency

In our intra-pipeline latency measurement, we aim to utilize a similar idea as *AVCloak*, which we described in Section 1.3.1. It is also similar to the method described in Section 2.2.2. There is one significant difference between *AVCloak*'s and our system. They do not have access to the measured system and perform black-box testing, and we may perform white-box testing. It allows us full insight into the system we are testing, and therefore we will be able to test more parts of the system. This insight creates a better picture of where improvements can be made to the general design. We call this *intra-pipeline* testing as it is performed "inside", or along the pipeline.

As the audio pipeline will be streaming data containing timestamps, see Section 2.2.2, we may extract them. They will be extracted at different points along the pipeline to measure the propagation time between elements. These measurements will paint a clearer picture of where we see the most substantial latency contributions in the audio pipeline

These timestamp calculations assume a system time that does not deviate from the sender and receiver. Thus synchronization is again maintained using NTP.

2.2.4 Linux Network Stack Latency

Latency in the kernel may be measured by collecting a timestamp, t_{socket}, at the socket when the datagram arrives at the network interface. Followed by collecting another timestamp, $t_{userspace}$, when the datagram arrives to user space and calculating the delta: $\Delta t = t_{userspace} - t_{socket}$.

This measurement may provide information on how our network interface and kernel-space propagate different packet sizes per time unit.

Three tests are run on the two network setups described in Section 2.2.1 to measure how the two architectures handle different packet sizes. The tests measure the propagation time taken per packet for the packet sizes $1\,\mathrm{ms}$, $5\,\mathrm{ms}$, $20\,\mathrm{ms}$, which are also sent in the intervals of $1\,\mathrm{ms}$, $5\,\mathrm{ms}$, $20\,\mathrm{ms}$ per packet. The program used to measure this is presented in Section 2.3.1.

2.2.5 Network traffic monitoring

While collecting information on the system-level might provide more detailed information about the inner workings of the system, some form of external monitoring may be beneficial. It provides a macro-perspective of the connected systems and produces other forms of information, which might be harder to gather on the embedded systems.

Data points of interest might be overall link capacity utilization, traffic characteristics such as traffic composition, and composition of transmitted data for debugging. These give us information about whether the issues we see in the networked audio system have their roots in the network or the software implementation. If the results show that the network has a significant influence on the system, it can guide us further to find the root causes.

To monitor overall link capacity on the network, we use SNMP and a free data collection and monitoring tool called Zabbix. For our needs, we only utilize the monitoring tools for network traffic flow and throughput analysis between the devices.

By using port mirroring and Wireshark, we will be able to monitor the traffic composition and individual packets being sent and received between the systems. It lets us inspect this traffic to ensure our test traffic is passing correctly, or inspect unexpected traffic and its purpose.

2.2.6 Test parameters

In Table 2.1, the options for each test we have defined are presented. These options are present in the configuration of both the sending and receiving systems. The different options are described as follows:

period-time Represents the ALSA option with the same name. It describes the size of the audio chunks extracted from ALSA. Value in ms.

buffer-time Represents the ALSA option with the same name. It describes the capacity of the ALSA buffer where audio can be buffered. Value in ms.

p-time The size of the RTP media-payload in each network frame. Value in ms.

jitter-latency The size of the receiver's jitter buffer. Value in ms.

pipeline-latency The total pipeline target latency for the system. Value in ms.

	Test 1		Test 2		Test 3		Test 4		Test 5	
	S	**R**	**S**	**R**	**S**	**R**	**S**	**R**	**S**	**R**
period-time	20	20	20	20	10	10	10	10	5	5
buffer-time	40	40	40	40	20	20	20	20	10	10
p-time	20		20		10		5		5	
jitter-latency		25		25		25		25		25
pipeline-latency		100		80		60		60		60

	Test 6		Test 7		Test 8		Test 9		Test 10	
	S	**R**	**S**	**R**	**S**	**R**	**S**	**R**	**S**	**R**
period-time	5	5	5	5	5	5	5	5	5	5
buffer-time	10	10	10	10	10	10	10	10	10	10
p-time	5		5		5		5		1	
jitter-latency		25		25		10		10		10
pipeline-latency		40		30		30		20		20

Table 2.1: Test suite with decreasing total pipeline latency. All values are in ms, *S* stands for Sender, and *R* stands for Receiver. Values marked with a gray background are changes made from the previous test.

2.3 Implementation

In this section, we present the applications we have written and try to highlight some of our design choices. We have mostly developed small applications in order to facilitate our different testing needs, which we described in Section 2.2.

We present the final solutions for the different applications below. We will also mention some of the changes we have made along the way to get to the current implementations.

2.3.1 Ku-latency

To measure the kernel latency as described in Section 2.2.4, we modify a program, Ku-latency, written by M. Vilimpoc [20]. The modified program can be found in Appendix A.1.

Linux supports the timestamping of packets at the receiving socket. This feature timestamps all received packets due to their protocols being ambiguous at the timestamping level [14]. The timestamping feature will pack a timestamp with microsecond accuracy alongside our received datagram. The user space timestamp is taken as soon as a message has been received. See Appendix A.1 for more details.

Implementation issues We noticed an overhead in logging performance, which impacted our measurements by introducing jitter. This overhead was resolved by logging to a fixed-size in-memory buffer. Overhead was also seen when using a dynamically growing in-memory buffer due to the allocation characteristics of `glibc`.

2.3.2 Latsend

As mentioned in Section 2.2.2, we will utilize the property of the audio streaming in a raw format. To utilise this, we pack a structure of 24 bytes with a preamble, an index, and a timestamp, as seen in Listing 2.3. The index is of interest in the white-box test, described in Sections 2.2.3 and 2.3.4.

```
1 typedef struct {
2   guint32 __preamble;
3   guint32 index;
4   struct timespec time;
5 } TimeData;
```

Listing 2.3: Structure representing the timestamp metadata.

Latsend is implemented as a GStreamer pipeline and found in Appendix A.2. An overview of the pipeline can be found in Figure 2.7. As illustrated at the beginning of the pipeline graph, `AudioTestSrc` was used as an audio producer. This choice was made to make use of the built-in rate matching capabilities, such that we could match the properties of a real audio stream.

The previously mentioned structure, Listing 2.3, will be written to the beginning of each audio buffer that passes the `SrcPad` in Figure 2.7.

Figure 2.7: Latsend GStreamer pipeline.

Implementation issues

Initially, the production of the audio stream was created using `AppSrc`. This required careful calculations of the data rate and buffer sizes passed to the `AlsaSink`. It was more difficult than intended to create a steady stream of audio data due to the data-request callback design that could not adhere to the soft real-time deadlines. It introduced audio production jitter, which propagated throughout the pipeline obfuscating the measurements.

The decision was made to make use of GStreamer's built-in source element `FakeSrc` to alleviate these kinds of issues. It was however quickly interchanged for `AudioTestSrc` because of the lack of native audio stream support in `FakeSrc`.

2.3.3 Latrecv

The implementation of Latrecv is analogous to the pseudo-code and description in Section 2.2.2 and Listing 2.2. The implementation can be found in Appendix A.3. All buffers will pass a pad probe on the `AlsaSrc` `SrcPad` when they are received from the ALSA loopback.

This pad probe scans the buffer for a 32-bit preamble, retrieving the time data described in Listing 2.3. Then the latency time difference between t_{probe} and the initial time t_0 is logged for post-analysis.

The `AlsaSrc` element will request an ALSA buffer with a period time, as described in Section 2.1.4. Doing this will introduce additional latency, due to the extra buffer, to the measurement t_{probe}. After this, the audio data will be discarded by the consumer `FakeSink`.

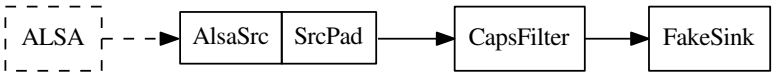


Figure 2.8: Latrecv GStreamer pipeline.

Implementation issues

Previously an `AppSrc` was used to retrieve the audio data after the `CapsFilter` in Figure 2.8. It resulted in relying on a callback signal upon data arrival, and it introduced some additional buffer delay.

2.3.4 Intra-pipeline latency

We used `Probes`, which are described in Section 2.1.4, to measure the intra-pipeline latency, the time spent passing the pads seen in Figure 2.9.

The probes are placed on pads denoted by the *PAD* boxes seen in Figure 2.9. At each probe, a buffer is scanned for a preamble, index, and timestamp. The index is used to correlate the timestamp data at each probe and between the sender and receiver. The calculated difference in system time can be seen in the lower part of Figure 2.9.

Each probe contains a data collection context and an in-memory logging buffer, to reduce the overhead of recording timestamp and index. Like Section 2.3.1, these in-memory buffers are static in allocation. For details, see Appendix A.5: `probe_callback` and `debug_latency`.

This in-memory data collection will stay in memory throughout the test period. When the specified number of timestamps has been collected, the probes will log their in-memory data buffer to disk for further review.

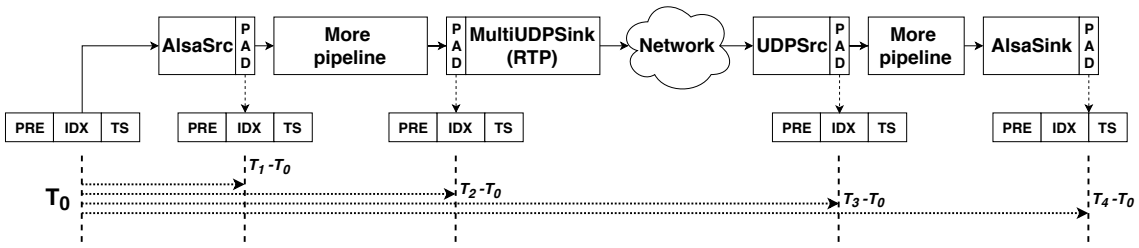


Figure 2.9: Intra-pipeline figure, displaying the probing methodology.

Implementation issues

As an initial implementation, before the usage of probes, the `Tee` element was used. A `Tee` splits one pipeline sink into two. It was used to duplicate the stream into measurement endpoints, namely `AppSrc` elements. Additionally, `Queues` were used to make these measurement points concurrent. Nevertheless, the buffering and overhead of the `Queue` and `Tee` combination resulted in unreliable measurements.

Chapter 3
Evaluation

In this chapter, we present our evaluation of the systems and summarize our findings to showcase what affects the audio system, and in what manner.

3.1 Result

Here we present the results of each test described in Section 2.2.

3.1.1 Network characteristics

Looking at the two networks, we see some differences. There is a constant flow of traffic from the switch's backhaul connection in the noisy network setup. A compilation of these measurements between different nodes can be found in Table 3.1. The *Switch Backhaul* corresponds to the connection between the router and the switch in Figure 2.3a, and correspondingly between the switch and the lab network in Figure 2.3b. The sender corresponds to the sender in both figures, and the same applies to the receiver. In Table 3.1, we see that the switch's backhaul connection has a sustained load of around $4\,\mathrm{Mbit/s}$ on the noisy network.

The type distribution of this traffic can be found in Figure 3.1, where we see something interesting. While RTP has the highest throughput of data, mDNS has a higher packet throughput. The distribution results were obtained using Wireshark when no other tests were being performed by collecting ambient traffic.

Looking at the sender and receiver, we see the propagation of the multicast and broadcast traffic in the noisy networks. Additionally, the sender is transmitting traffic at $1.6\,\mathrm{Mbit/s}$. The traffic then passes the switch and gets forwarded to the receiver, which gets a total ingress of $5.6\,\mathrm{Mbit/s}$ in the noisy network. In the silent network, however, we only see the streaming traffic between the sender and receiver as expected.

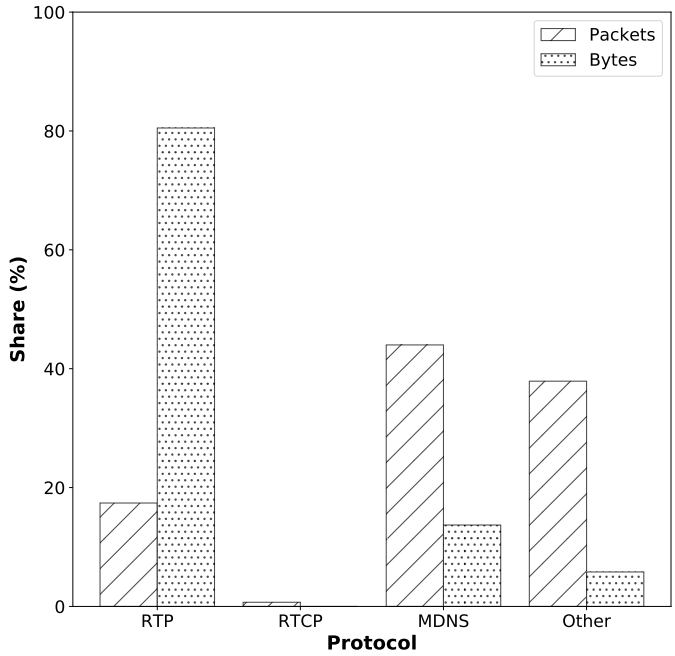


Figure 3.1: Distribution of protocols in ambient traffic of the noisy network.

	Noisy		Silent	
	Ingress	**Egress**	**Ingress**	**Egress**
Switch Backhaul	4	0	0	0
Sender	4	1.6	0	1.6
Receiver	5.6	0	1.6	0

Table 3.1: Sustained traffic throughput of the different devices on the network during testing. Values are in $\mathrm{Mbit/s}$

.

When the *p-time* variable in Table 2.1 changes, the RTP-packet size over the network changes. These sizes are summarized in Table 3.2. There we see the size difference between a $20\,\mathrm{ms}$ and $1\,\mathrm{ms}$ RTP frame, logically telling us that the packet rate needs to grow by $20x$ for the same data rate to be sustained. It is, however, important to remember that each of these RTP frames needs other encasing protocols to be transmitted. This additional padding causes the actual data rate to increase when using smaller packet sizes as the same playback rate of the audio data is required while using additional headers. Looking at the $20\,\mathrm{ms}$ case, we see that for 3840 bytes of audio data, 3974 needs to be transmitted, meaning an overhead of $1 - 3840/3974 \approx 3.4\,\%$. While in the $1\,\mathrm{ms}$ case, it is 192 bytes of audio data and 250 bytes total per frame. Which instead gives $1 - 192/250 \approx 23.2\,\%$ (!) of overhead.

	20 ms	10 ms	5 ms	1 ms
Ethernet	3974	2016	1018	250
IP	3920	1980	1000	232
UDP	3860	1940	980	212
RTP	3840	1920	960	192

Table 3.2: Sizes of the payload in bytes for different TCP/IP layers.

3.1.2 Kernel network latency

As described in Section 2.2.4, we can measure the kernel propagation latency of each incoming packet traveling from kernel space to the socket in user space. This section introduces the measurements of the network latency in the Linux kernel, Figures 3.2 and 3.3, measured on the silent and noisy network using architecture A and architecture B.

Silent network

We note that the mean latency is higher on architecture A, compared to architecture B, in all three measurements of Figure 3.2.

It is notable that the increase in packet size, from $1\,\mathrm{ms}$ to $20\,\mathrm{ms}$ of audio data, increases the propagation time. We note a mean increase of $67.1\,\%$ in architecture A and $29.8\,\%$ in architecture B.

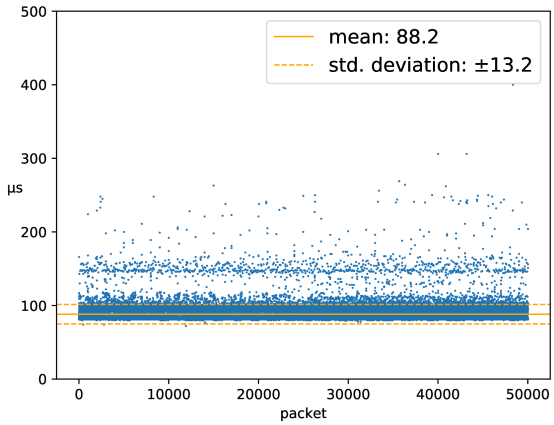
Approximately 25000 packets separate the latency spikes we see in Figures 3.2c and 3.2e. These packets represent$125\,\mathrm{s}$ in the $5\,\mathrm{ms}$ case, and $500\,\mathrm{s}$ in the $20\,\mathrm{ms}$ case. We have not observed this behavior in the $1\,\mathrm{ms}$ configuration, Figure 3.2a.

Noisy network

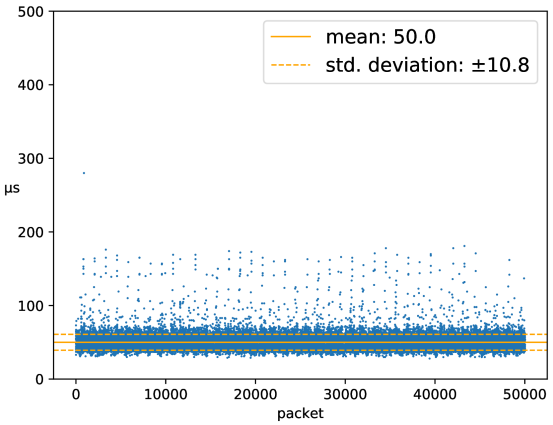
In the case of the noisy network, Figure 3.3, we see an increase in the standard deviation of all three tests on both architectures.

We note that all tests share a standard deviation of approximately $40\,\mu\mathrm{s}$, except for $105.1\,\mu\mathrm{s}$ for architecture A in Figure 3.3e.

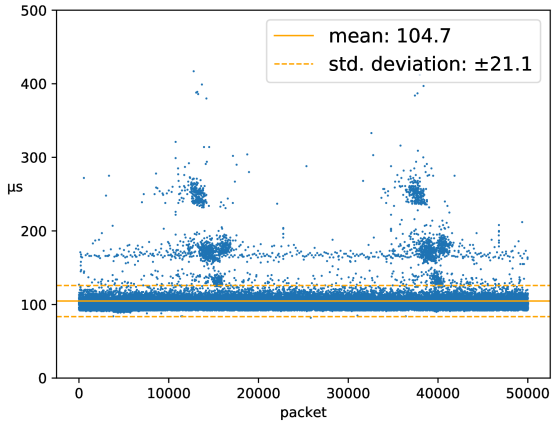
A notable increase in the mean is seen in both architectures when increasing the packet size from $1\,\mathrm{ms}$ to $20\,\mathrm{ms}$. An increase of $108.1\,\%$ in architecture A and $38.6\,\%$ in architecture B.



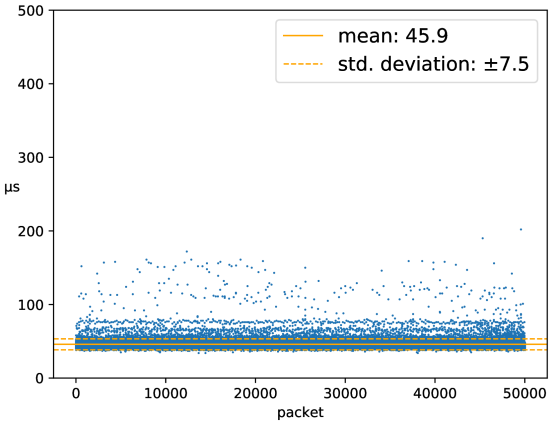
(a) Architecture A receiving a packet size of 1 ms.



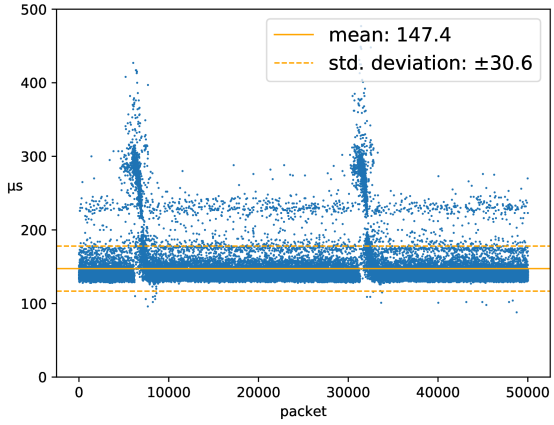
(b) Architecture B receiving a packet size of 1 ms.



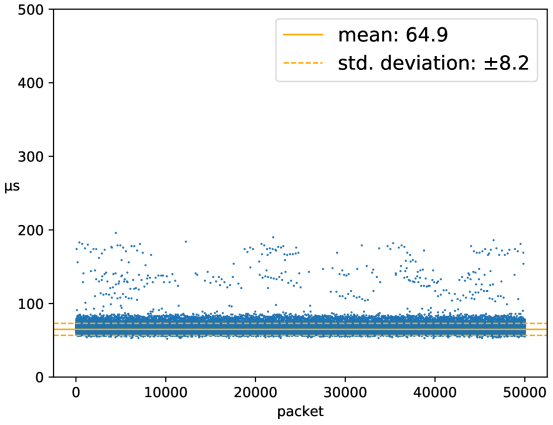
(c) Architecture A receiving a packet size of 5 ms.



(d) Architecture B receiving a packet size of 5 ms.

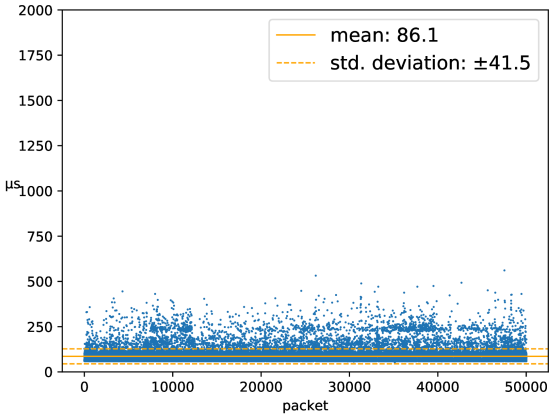


(e) Architecture A receiving a packet size of 20 ms.

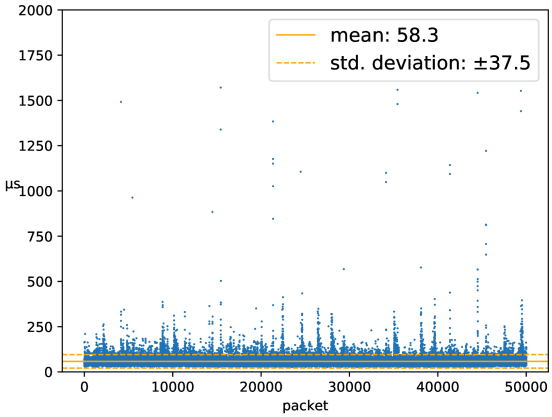


(f) Architecture B receiving a packet size of 20 ms.

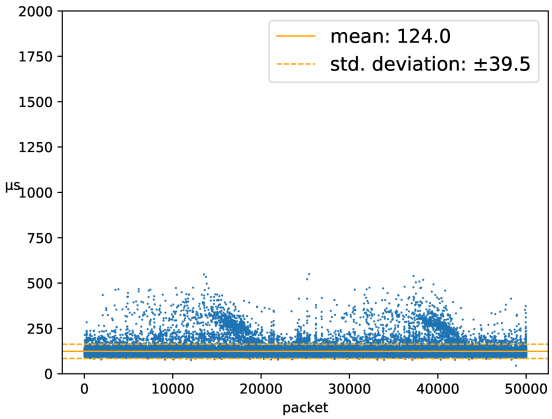
Figure 3.2: Kernel to user space network propagation latency per packet over 50000 packets on a silent network.



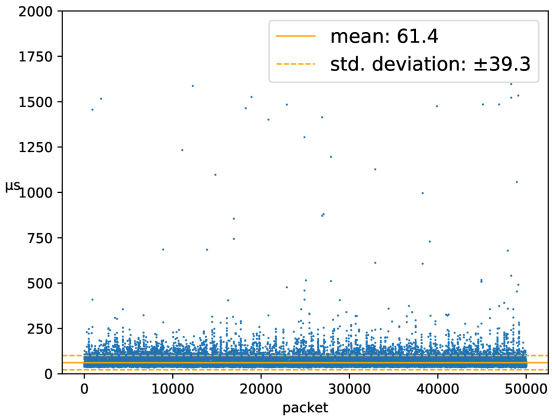
(a) Architecture A receiving a packet size of 1 ms.



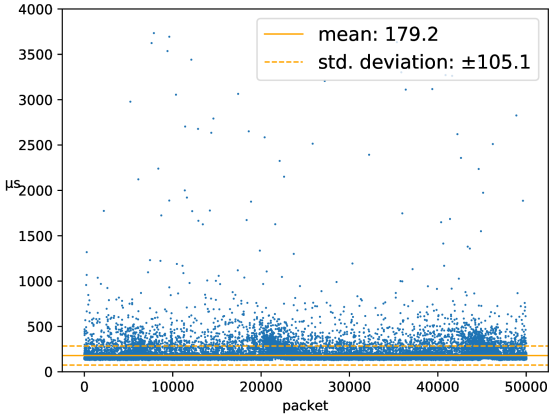
(b) Architecture B receiving a packet size of 1 ms.



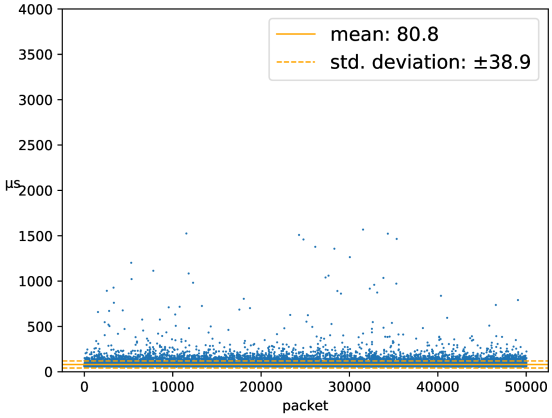
(c) Architecture A receiving a packet size of 5 ms.



(d) Architecture B receiving a packet size of 5 ms.



(e) Architecture A receiving a packet size of 20 ms.



(f) Architecture B receiving a packet size of 20 ms.

Figure 3.3: Kernel to user space network propagation latency per packet over 50000 packets on a noisy network.

3.1.3 Full pipeline latency

The analog signal tests have been performed following the method described in Section 2.2.2. Resource monitoring has been performed simultaneously following the method described in Section 2.2.2.

The analog tests resulted in the graphs presented in Figures 3.5, 3.6, and B.1. These results were then compiled into Tables 3.3 and 3.4 to provide a better overview.

To provide an overview of the resource monitoring results a selection has been made, presented in Figures 3.7 and 3.8.

The digital test did not turn out as expected, a description of why follows in Section 3.1.3.

Analog signal measurements

The time difference for each test, $\Delta t = t_1 - t_0$, was calculated using results of the analog measurements like those in Figure B.1 in Appendix B.1. The resulting latencies can be found in Tables 3.3 and 3.4, where Δt_c represents the configured *pipeline-latency* from the test suite in Table 2.1, Δt_r the actual latency measured during testing, and t_p the configured *period-time* for that test.

There are two factors of interest to be observed in Figure 3.5. First, there is the measured pipeline latency, and secondly, the audio quality at different pipeline latency configurations.

Measured system latency If we observe Tables 3.3 and 3.4, we see that the measured pipeline latency generally decreases when the configured latency does. There are queues at points along the pipeline, which buffer audio passing through them if the latency of the audio does not match the one specified in the configuration. This buffering extends the time spent in the pipeline so that it matches the configured latency.

When comparing Architecture A and Architecture B in Table 3.3, we see that the measured latency is similar for both systems. The measured latency does, however, not match the configured latency, there is always an additional delay. A full explanation of this follows in the discussion. In short, audio samples are collected at the source interface to fill a *period-time* before hand-over causing an additional delay for which the pipeline does not seem to account.

If we instead look at Table 3.4, we see that the overhead decreases for lower latency configurations. Architecture B also drops in latency for the lower latency tests, bringing it closer to the performance of architecture A.

Audio quality Our testing shows that audio quality deteriorates as the configured pipeline latency decreases. The extreme cases can be observed in Figures 3.5a and 3.5b versus Figures 3.5c and 3.5d, where the first two represent a higher pipeline latency configuration, and the second two represent a low pipeline latency configuration, all on the noisy network. The peaks observed in Figures 3.5c and 3.5d are signal disruptions, a magnification of such a disruption can be found in Figure 3.4 where we see that the system does not manage to recreate the $300\,\mathrm{Hz}$ audio signal from the input.

			Architecture A		Architecture B	
Test	Δt_c	t_p	Δt_r	$\Delta t_r - \Delta t_c$	Δt_r	$\Delta t_r - \Delta t_c$
1	100	20	120	20	120	20
2	80	20	100	20	100	20
3	60	10	70	10	70	10
4	60	10	70	10	70	10
5	60	5	65	5	65	5
6	40	5	45	5	50	10
7	30	5	40	10	45	15
8	30	5	40	10	40	10
9	20	5	25	5	35	15
10	20	5	30	10	30	10

Table 3.3: Latency difference between the analog measurements and the configured latency defined for each test on the noisy network.

			Architecture A		Architecture B	
Test	Δt_c	t_p	Δt_r	$\Delta t_r - \Delta t_c$	Δt_r	$\Delta t_r - \Delta t_c$
1	100	20	120	20	120	20
2	80	20	100	20	100	20
3	60	10	70	10	70	10
4	60	10	70	10	70	10
5	60	5	65	5	65	5
6	40	5	45	5	45	5
7	30	5	45	15	45	15
8	30	5	35	5	35	5
9	20	5	25	5	30	10
10	20	5	28	8	28	8

Table 3.4: Latency difference between the analog measurements and the configured latency defined for each test on the silent network.

As the configured latency decreases, we see that the intermediate tests between 1 and 10 show an increasing amount of audio signal disruptions. If, however, the same tests are performed on the silent network, then the resulting audio signal for test 10 can be observed in Figures 3.6a and 3.6b. Here we see a drastic improvement of the audio signal.

Lastly, there are Figures 3.6c and 3.6d. These are measurements with test 10 parameters performed on the noisy network. The only differentiator between these and Figures 3.5c and 3.5d is the mDNS services, which has been disabled on both of the systems. Comparing these two sets of figures we can see a clear decrease in signal disruptions for architecture A and an almost complete disappearance on architecture B except for one single disruption.

Resource utilization

In this section, we introduce the resource usage results that accompany the tests of the analog signal in Section 3.1.3. As described in Section 2.2.2, these measurements are polled at $1\,\mathrm{s}$ intervals. Figure 3.7 visualizes the measurements on the noisy network, and Figure 3.8 on the silent network.

In Figures 3.7 and 3.8, we can see that throughout the tests, architecture A has a lower CPU utilization and a higher interrupt and context switching rate.

We see that network traffic on the noisy network produces a noticeable increase in CPU-time, interrupt rate, and context switch rate, at non-deterministic time instants, which can be observed as jitter in all tests of Figure 3.7. Interestingly, if mDNS is disabled like in Figures 3.9a and 3.9b, we see a decrease in CPU utilization closer resembling that of the silent network than the noisy, the result also shows less jitter. The amount of context switches is only marginally higher than the corresponding silent network tests. Looking at the CPU utilization, we do see some differences, the distribution between user and kernel space usage is not as even as in the silent network tests.

There is a notable increase in resource utilization from test 1 to 10 in Figures 3.7 and 3.8. We also note that the interrupt rate is several orders lower in architecture B Figure 3.7d and that the context switch rate is half of that of architecture A in Figure 3.7c.

Notably, architecture A does not have an even CPU-time utilization in Figure 3.8c, as we would expect on the silent network. The other tests in Figure 3.8 showcase the even pipeline CPU-time utilization.

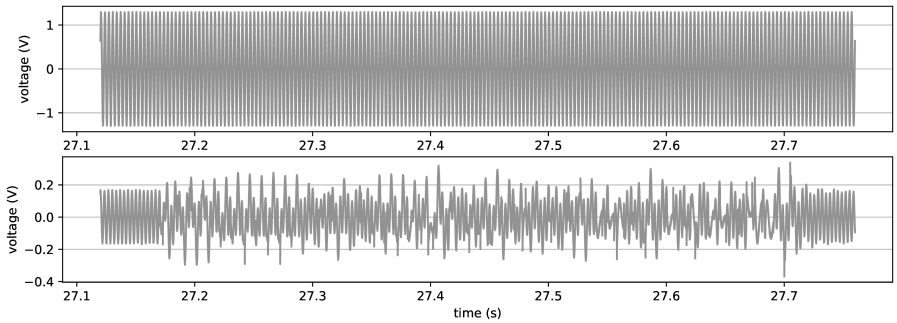
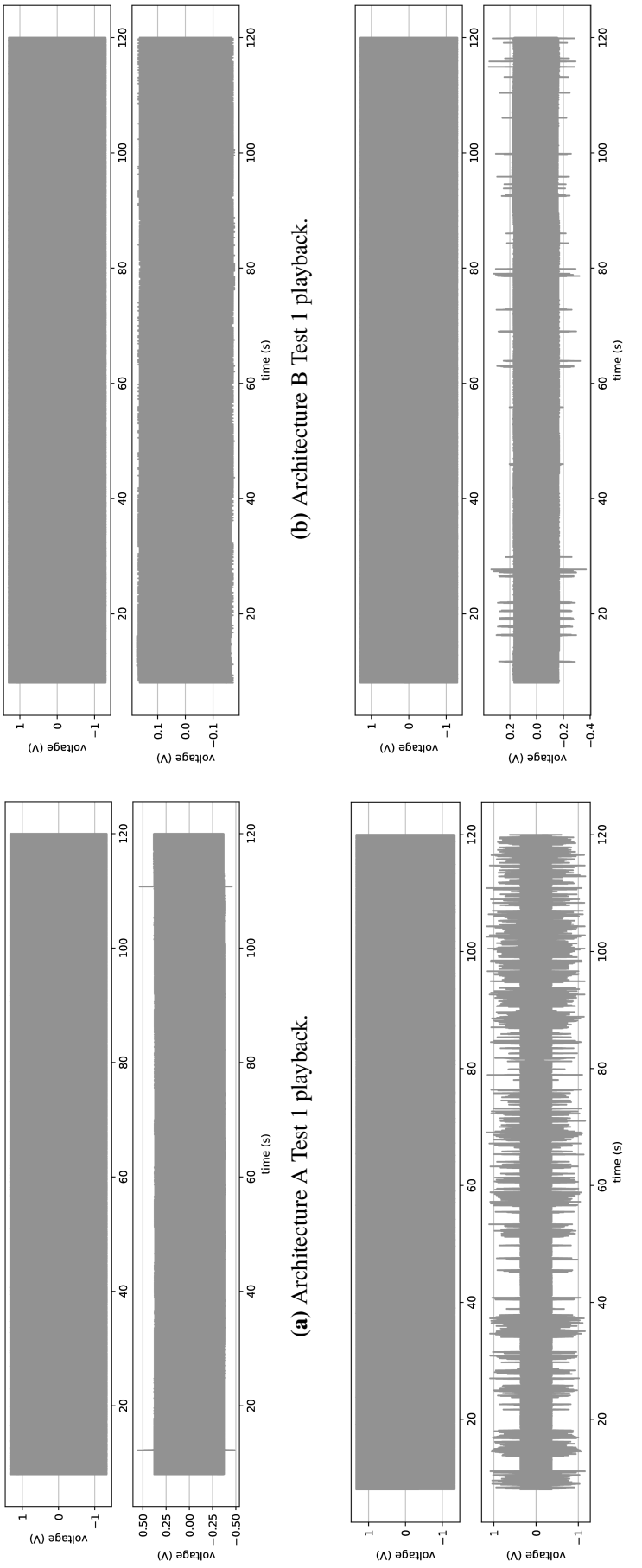


Figure 3.4: Magnification of signal disruptions.



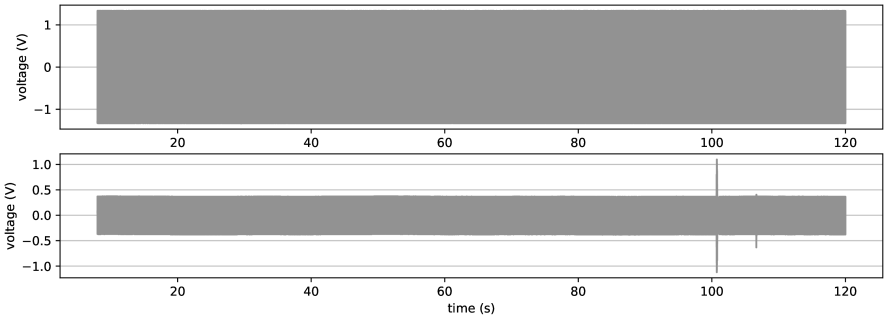
(a) Architecture A Test 1 playback.

(b) Architecture B Test 1 playback.

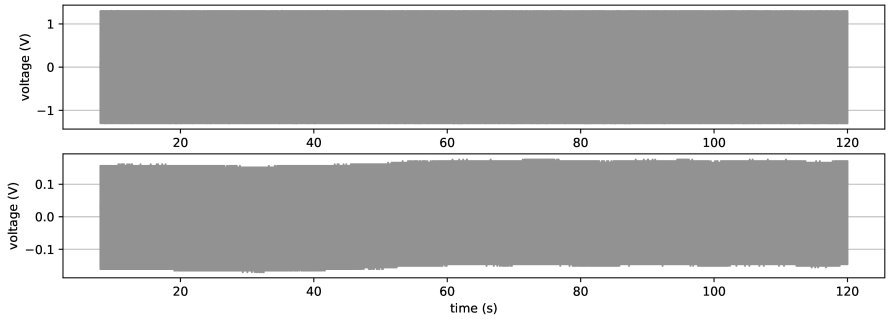
(c) Architecture A Test 10 playback.

(d) Architecture B Test 10 playback.

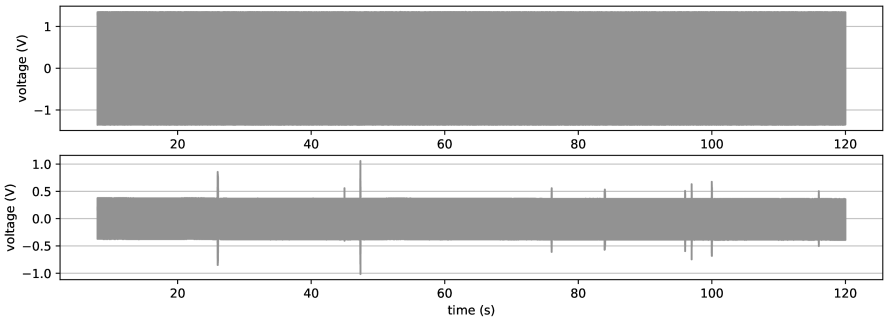
Figure 3.5: Playback signal quality for test 1 and 10 on the noisy network.



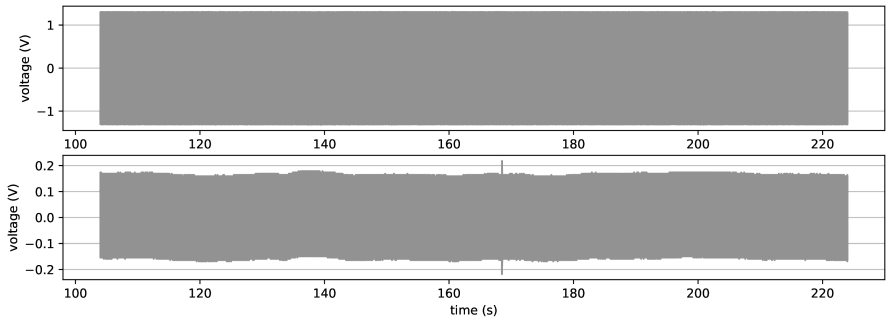
(a) Architecture A Test 10 playback on silent network.



(b) Architecture B Test 10 playback on silent network.

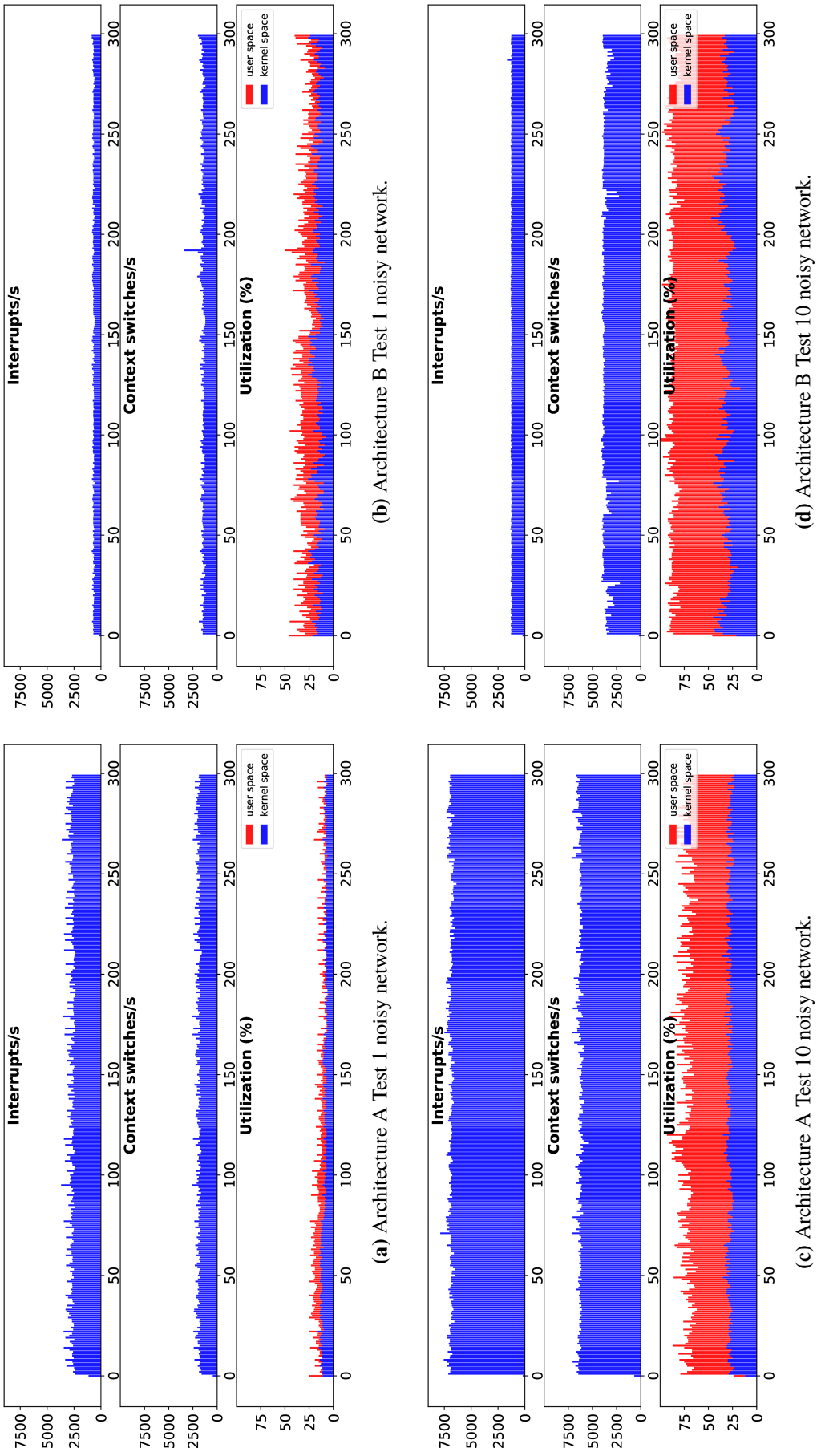


(c) Architecture A Test 10 playback on noisy network without mDNS.



(d) Architecture B Test 10 playback on noisy network without mDNS.

Figure 3.6: Playback signal quality for test 1 and 10 on the silent network versus the same tests on the noisy network but without mDNS.



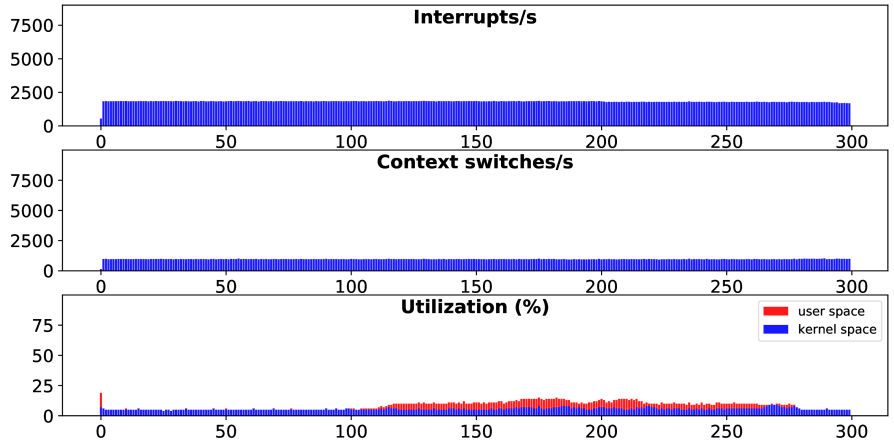
(a) Architecture A Test 1 noisy network.

(b) Architecture B Test 1 noisy network.

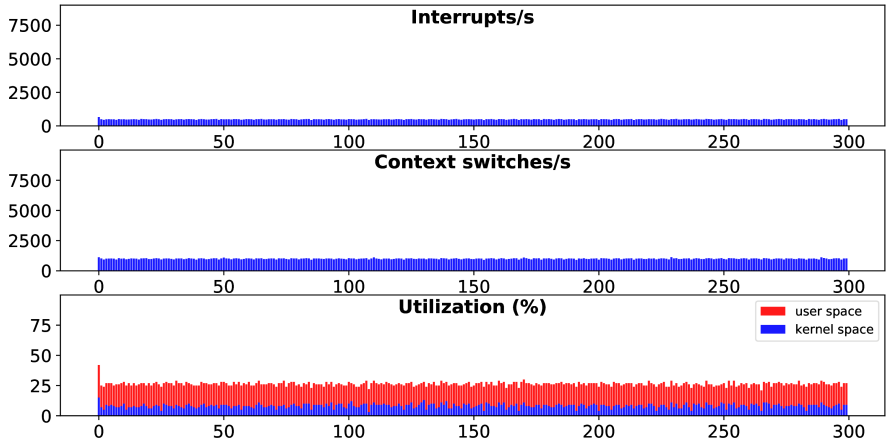
(c) Architecture A Test 10 noisy network.

(d) Architecture B Test 10 noisy network.

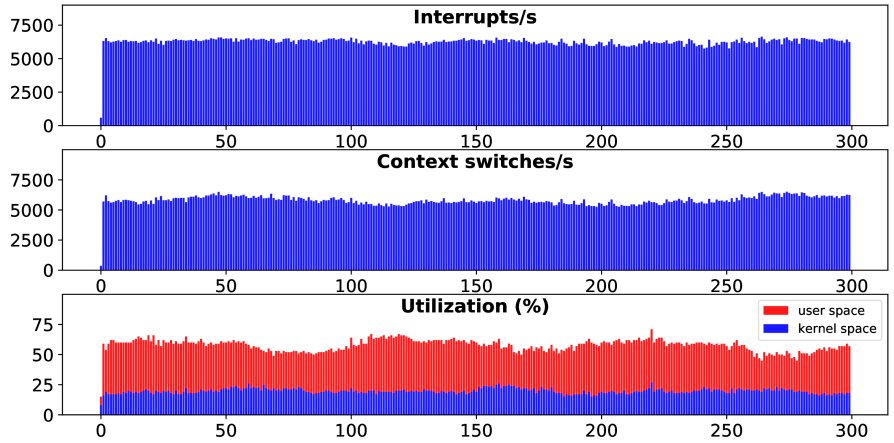
Figure 3.7: System utilization measurements produced by `Vmstat` on the noisy network over 300 s.



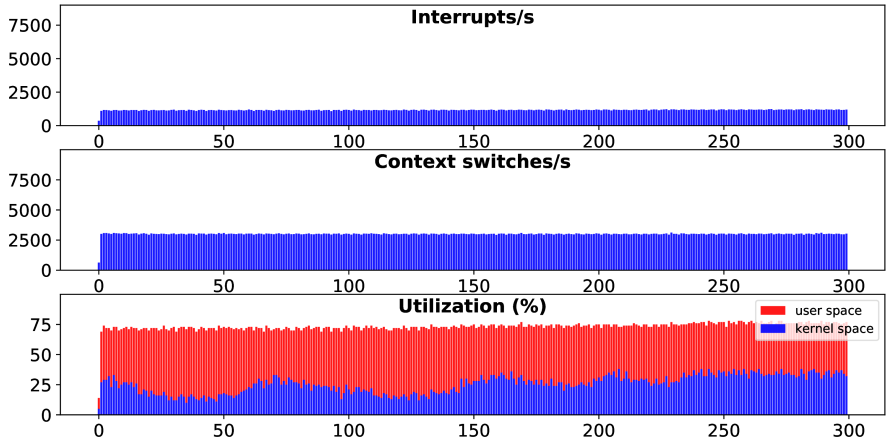
(a) Architecture A Test 1 silent network.



(b) Architecture B Test 1 silent network.

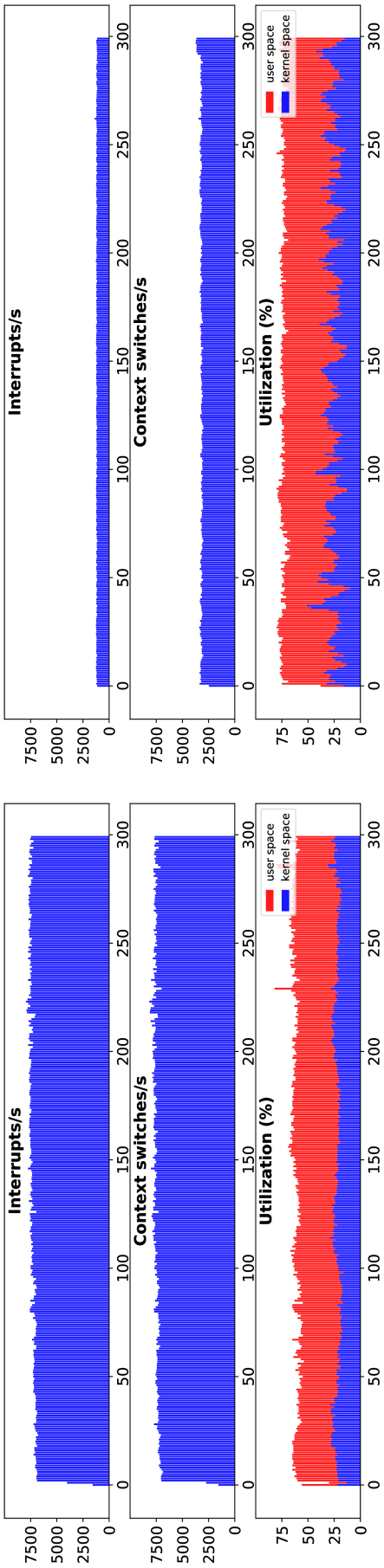


(c) Architecture A Test 10 silent network.



(d) Architecture B Test 10 silent network.

Figure 3.8: System utilization measurements produced by `Vmstat` on the silent network over 300 s.



(a) Architecture A Test 10 noisy network without mDNS.

(b) Architecture B Test 10 noisy network without mDNS.

Figure 3.9: System utilization measurements produced by `Vmstat` on the noisy network without mDNS over 300 s.

Digital measurements

While implementing and testing `latrecv` and attempting to validate the results with the analog test results, we found that the digital tests always reported latencies of several tenths of milliseconds more than the analog tests.

During that time, at the beginning of the thesis, we were developing several applications for testing simultaneously. Due to this test not performing as we had intended, matching the analog test, we decided to put this part outside of our scope and focus on the analog, intra-pipeline, and system performance tests.

Later during the thesis, when we had an increased understanding of the system, and after additional research into the ALSA configuration, we found this to be a natural deviation that we had overseen in the planning stages of our evaluation. Due to the additional interface between `latrecv` and ALSA, there are additional data buffers introduced at the output of ALSA. These are responsible for the increase in measured pipeline latency. The method used to retrieve the timestamp, using a `Pad` on the `AlsaSrc`, or connecting to an `AppSink`, also affect the measurement. As this was discarded at such an early stage no usable results exist.

3.1.4 Intra-pipeline latency

In these tests, we try to visualize the delays within the audio pipeline, which result in an overview of the intra-pipeline latencies between the GStreamer elements, and the two systems. To do this, we apply the method described in Section 2.2.3, using the implementation described in Section 2.3.4.

Due to the extensive amount of data from the tests, we have generated graphs composed of several test graphs to provide an overview and simplify comparison. These can be found in Figures 3.12 and 3.13, and the colors are described in Table 3.11a. The graphs present the latency in the y-axis, and the x-axis enumerates the tests. Each test is composed of $10,000$ probed timestamps that have traversed all probes in the pipeline. These are shown along the x-axis for each test. All tests have been offset such that the first probe appears at latency 0. This offset gives us a relative comparison of the latencies. The total latencies in these figures represent only the intra-pipeline part of the full system latency. A model of the full system latency distribution can be found in Figure 3.10, where the composite graphs represent the `Intra-pipeline measurement` section. The `AlsaSink` section is the rest of the receiver pipeline, and `Pre-AlsaSrc` the audio sample collection previously described in Section 3.1.3. With this, we see that even though tests 1 and 3 might have the same intra-pipeline latency, the latency of the full system has decreased. This disparity explains why test 5 in the composite has the longest relative pipeline latency, while the total system latency is less than that of previous tests.

The probes have been placed in reverse order of what is described in Table 3.11a. The absolute positions of them are described by the points A-H in Figures 2.5a and 2.5b, where the colors correspond to the same colors in both the table and the measurement graphs. Differences in time synchronization on the systems via NTP are in the $<1\,\mathrm{ms}$ range and are therefore considered insignificant and not included in the results.

Architecture A

In test 1 of Figures 3.12a and 3.13a, an audio packet that is buffered for longer than one buffer frame, pre-`AlsaSrc`, will spend equally less time in the receiving audio pipeline. This time disparity is represented by the repeated `AlsaSrc` timestamp lines at $20\,\mathrm{ms}$. This tendency carries over to other test configurations, where we see similar duplicate lines, but for different *period-time*s. These double `AlsaSrc` lines originate between the producer and consumer and can be seen in Figures 3.14a and 3.14c.

We can see that when decreasing the *pipeline-latency* of test 2 by $20\,\mathrm{ms}$ in Figure 3.12a, the playback line, `AlsaSink`, is lowered by a corresponding $20\,\mathrm{ms}$. This decrease causes tight deadlines in the pipeline, and delayed buffer periods will miss their playback deadline, resulting in a second playback line after the second `AlsaSrc` buffer period.

Tests 3 and 5 in Figures 3.12a and 3.13a both decrease the *period-time* by half. The decreased period-time is visualized by the increased time to deadline and the decreased distance between the `AlsaSrc` buffer period lines.

Looking at test 10 in Figures 3.12a and 3.13a, we see that the small *p-time* of $1\,\mathrm{ms}$ causes the system to lose most of the determinism of the previous tests. This jitter, however, was expected due to the increased overhead of processing many small audio packets.

In Figure 3.11, we see the results of using different scheduling algorithms. In Figures 3.11b to 3.11d, EDF has been used, and in Figure 3.11e, the default CFS has been used. As mentioned in Section 2.2.3, all other tests run RR. In the EDF cases, we see that only minor changes to the running time/deadline/period cause significant differences in the pipeline measurements. In the CFS case, we see that without time-slicing and strict prioritization of the pipeline, it deteriorates completely.

Finally, if we compare Figures 3.12a and 3.13a, we see a significant amount of jitter in the tests performed on the noisy versus the silent network. We can also see that while some tests do not exhibit the repeating `AlsaSrc` timestamp lines, such as test 4 on the silent network, all tests on the noisy network exhibit them. However, tests such as test 1 on the noisy network do not have as pronounced of a secondary `AlsaSrc` timestamp line as in the silent counterpart.

Architecture B

Turning our attention to Figures 3.12b and 3.13b, we see a complete lack of repeating `AlsaSrc` timestamp lines indicating less jitter in the handover from ALSA to the pipeline. The measurements further reinforce this in Figures 3.14b and 3.14d. This, in turn, results in an increased time spent in the *Queue* element in Figure 2.5b to adhere to the configured pipeline latency. Following these patterns, we see no double `AlsaSink` playback lines either in the architecture B tests.

Much like in the architecture A tests, we see an increase in the "time until deadline" in tests 3 and 5. It might not be as distinct of a change as in Architecture A's case as there are no repeated `AlsaSrc` timestamp lines to showcase the change in *period-time*.

Looking at test 10, which had a higher variance on Architecture A, we instead see how architecture B seemingly handles this more gracefully. Comparing the silent and noisy network graphs for test 10, we see a general increase in pipeline jitter on both networks as compared to the previous tests but more on the noisy network.

Comparing Figures 3.12b and 3.13b, we see an increased jitter in all tests on the noisy network. Overall, architecture B seems to fare better on the noisy network than its architecture A counterpart because of the lower jitter of the architecture B results.

Finally, we note a constant increase in the latency of Architecture B in Figures 3.12b and 3.13b. It represents a growing buffer in the receiving pipeline and is reset by ALSA's skew correction capabilities once reaching a specified size.

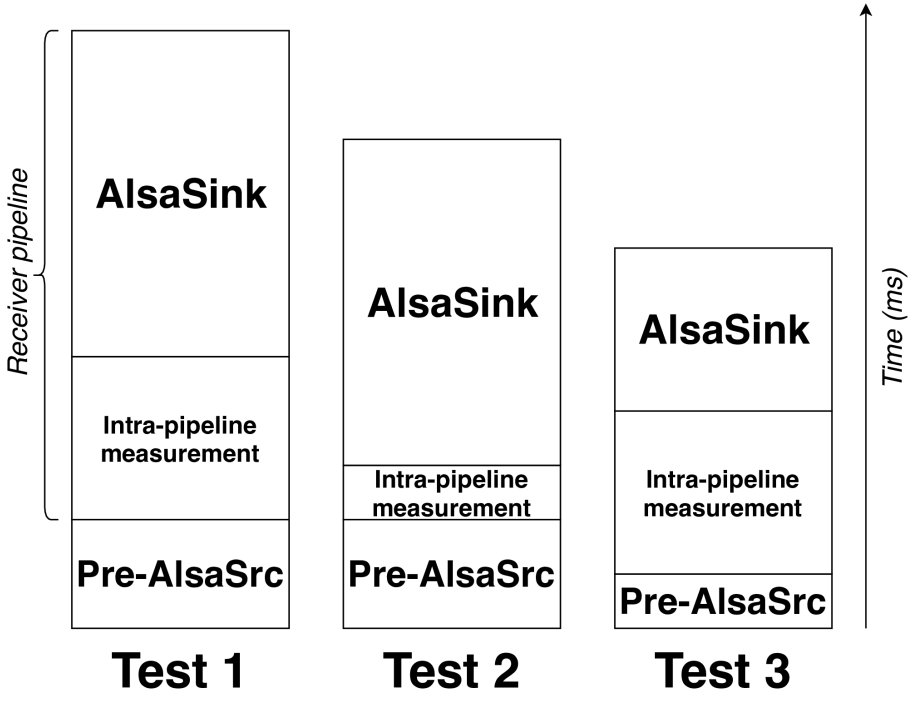
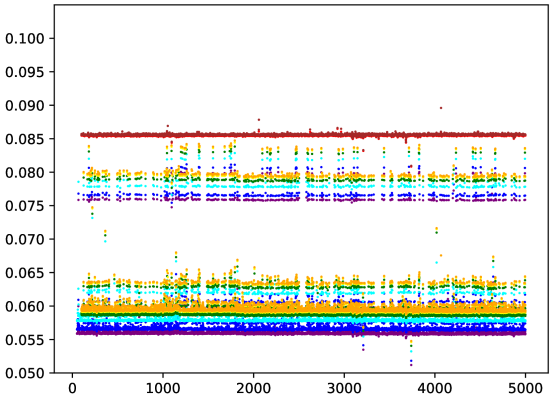


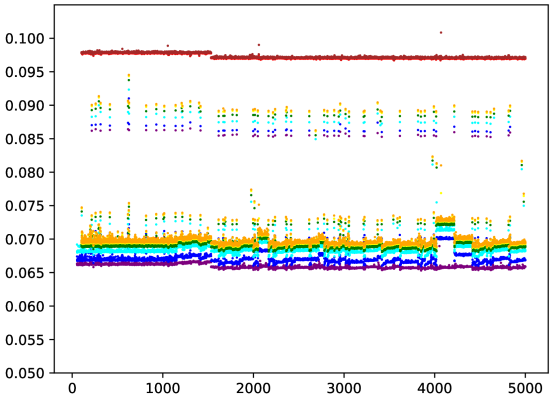
Figure 3.10: Intra-pipeline time distribution visualization model.

Color	Element
•	AlsaSink
•	Queue
•	InputSelector
•	AudioConvert
•	InputSelector
•	UDPSrc (RTP)
•	MultiUDPSink (RTP)
•	AlsaSrc

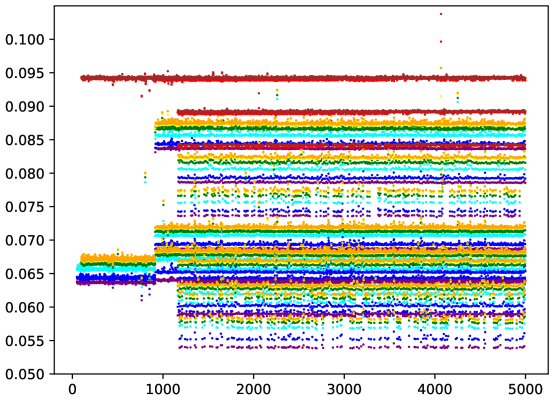
(a) Color representation of the following scatter plots based on the pipeline from Figures 2.5a and 2.5b.



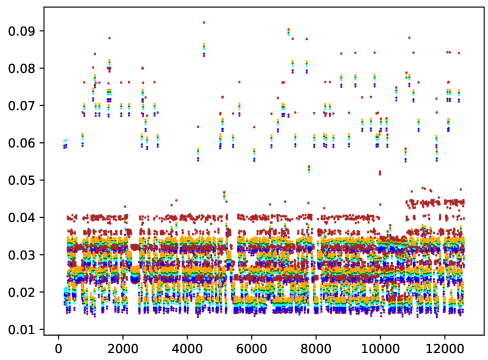
(b) Test 1 with EDF scheduling of 4/12/16.



(c) Test 1 with EDF scheduling of 5/10/20.

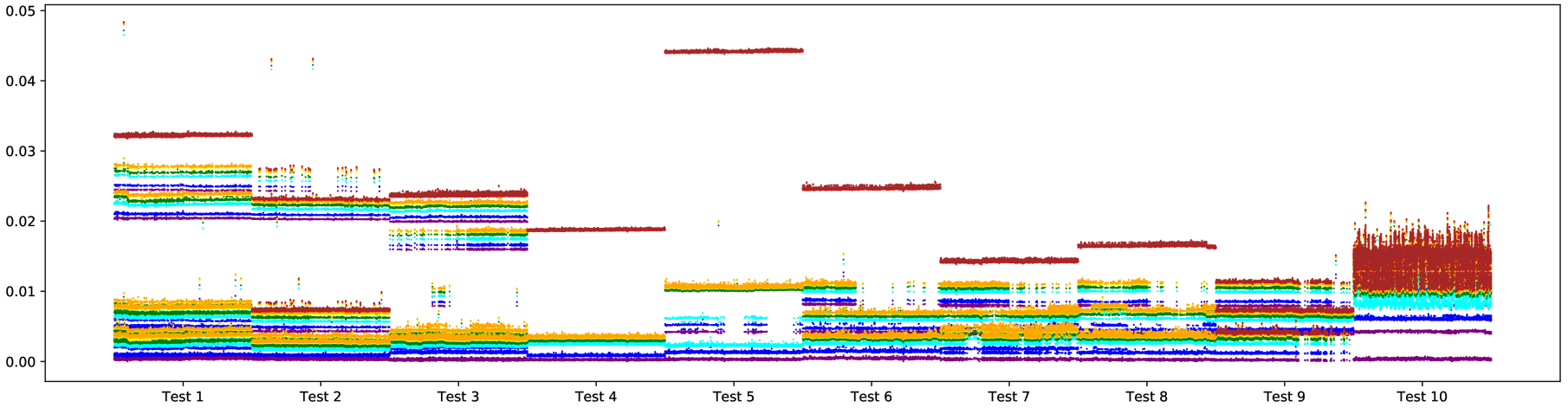


(d) Test 1 with EDF scheduling of 7.5/15/25.

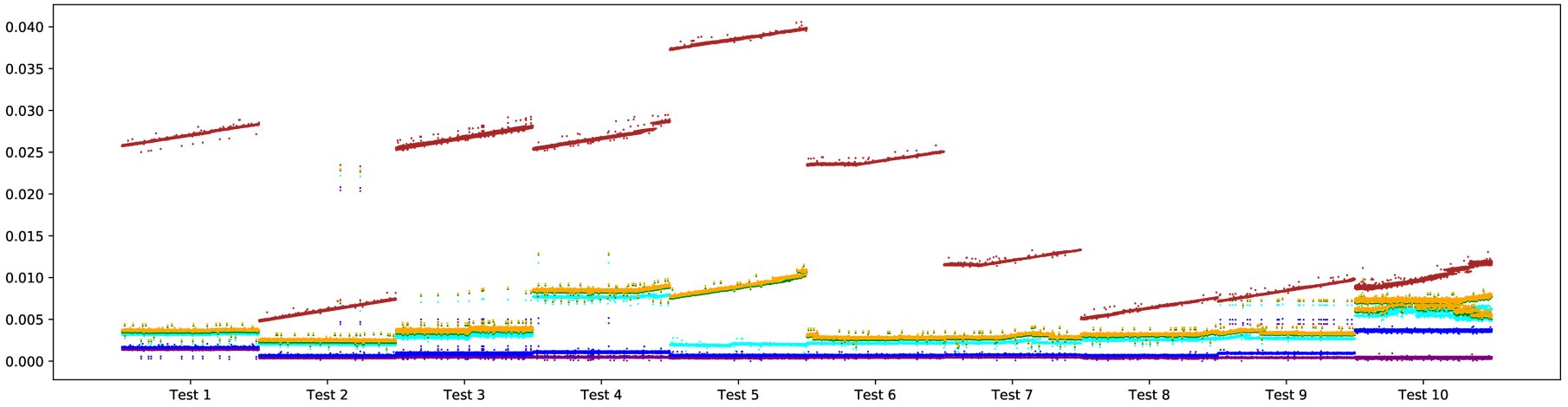


(e) Test 2 with CFS scheduling.

Figure 3.11: Times in between elements in the pipeline at different pipeline latencies. Normalization is adjusted towards entry time in the pipeline.

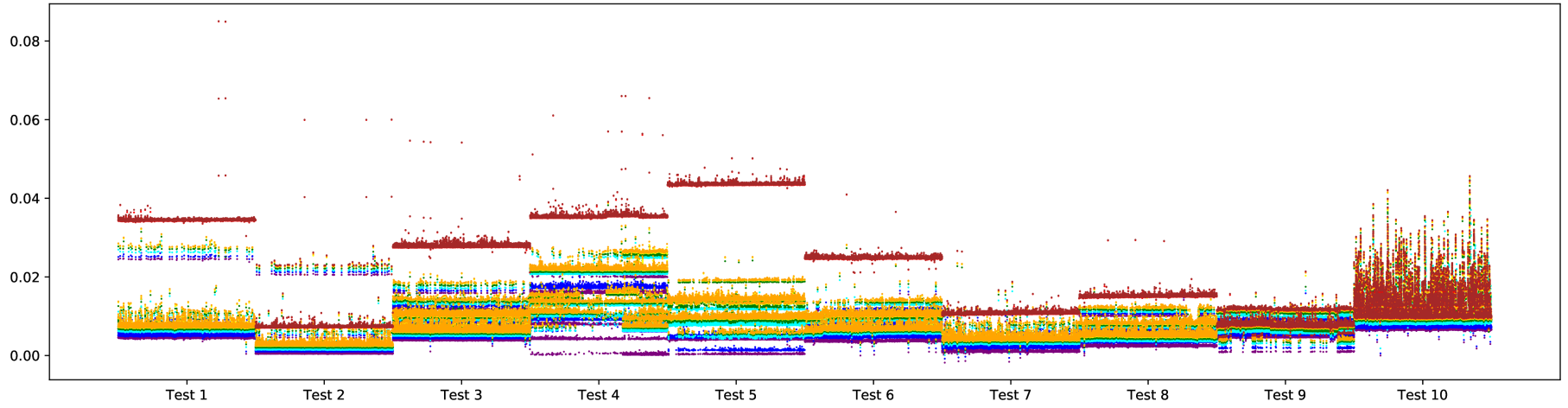


(a) Test 1 through 10 on architecture A.

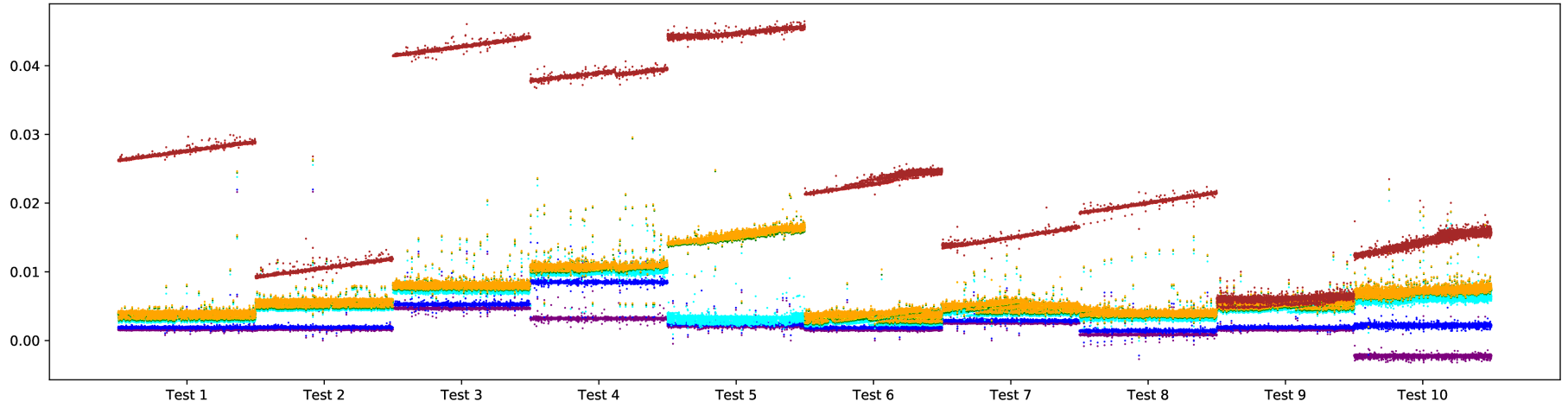


(b) Test 1 through 10 on architecture B

Figure 3.12: Intra-pipeline test 1 through 10 on both architectures for the silent network, with the lowest `AlsaSrc` measurements centered around 0 latency.

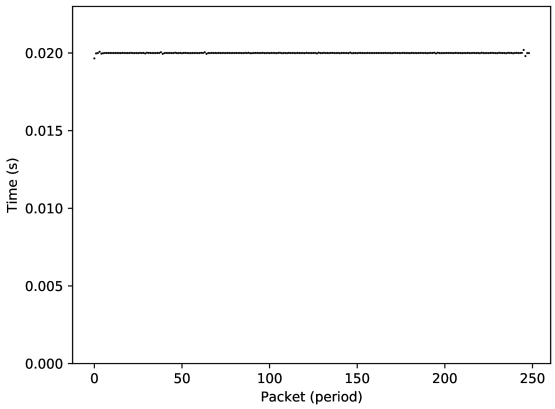


(a) Test 1 through 10 on architecture A.

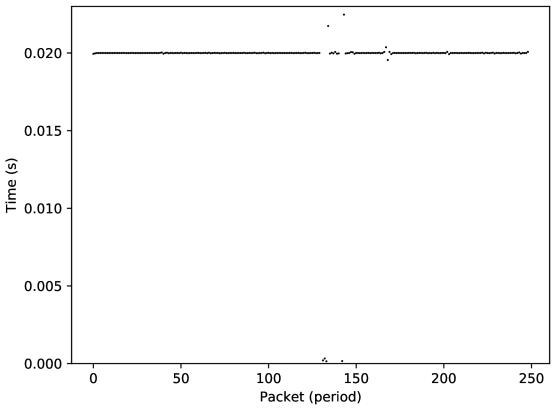


(b) Test 1 through 10 on architecture B

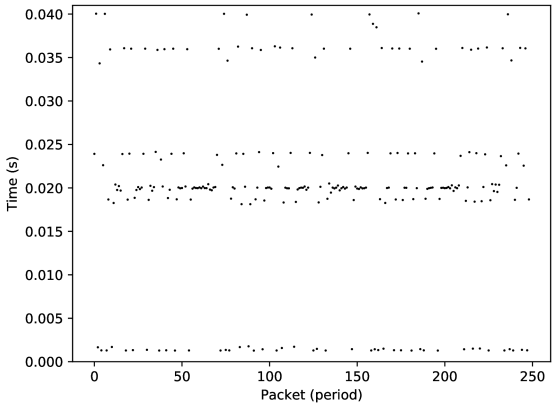
Figure 3.13: Intra-pipeline test 1 through 10 on both architectures for the noisy network, with the lowest `AlsaSrc` measurements centered around 0 latency.



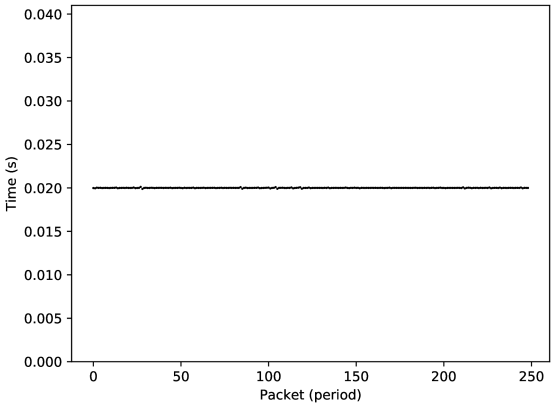
(a) Time between sample handover from audio producer to ALSA on architecture A.



(b) Time between sample handover from audio producer to ALSA on architecture B.



(c) Time between sample handover from ALSA to audio consumer on architecture A.



(d) Time between sample handover from ALSA to audio consumer on architecture B.

Figure 3.14: Representation of where the repeating lines in the architecture A intra-pipeline graphs originate, and that they don't occur on architecture B. The tests were performed on the silent network.

3.2 Discussion

In this section, we analyze our results. We describe why we think the systems perform the way they do and try to compare and discuss their performance.

3.2.1 Impact of network characteristics

Looking at the results in Figures 3.12 and 3.13, we can see that the distance between the `MultiUDPSink` and the `UDPSrc` is almost negligible. It usually corresponds to about 1 to $3\,\mathrm{ms}$, which is reasonable considering a throughput of $100\,\mathrm{Mbit/s}$ results in a transmission propagation time of $3974 * 8/(100 * 10^6) \approx 0.000\,32\,\mathrm{s}$ for each RTP frame on an architecture A system and ten times less than that on an architecture B system.

Early on in our research, we performed a test where we attempted to stress the NICs to cause a significant load on the system. It was performed while receiving an audio stream by sending a large number of useless data packets. We could conclude that receiving large amounts of unhandled traffic did not affect the systems noticeably, the traffic needs to be processed by the system for it to affect the performance. Due to the characteristics of control traffic, such as mDNS and other protocols that use rather small network frames, the high rates can cause an increased load on the system. The small packets also mean that a high packet rate can be achieved without necessarily requiring a high data rate. Important to note is, of course, that if we had saturated the link fully with garbage, we would have displaced the legitimate traffic causing a denial of service attack, which would have affected the performance.

Tying into this, if we look at the overall results, we see a clear trend where the noisy network causes more signal disruptions in the analog tests and jitter in the intra-pipeline measurements. Looking at the traffic analysis of the noisy network in Figure 3.1, we see that the majority of the data rate is created by multicast RTP traffic, which is not handled by our systems. The majority of the packet rate is created by mDNS traffic, which is handled by our systems. We can also see how disabling mDNS on the systems alleviate most of these issues. As mentioned previously, mDNS uses multicast as its communication scheme. Due to this, naturally, the amount of traffic increases exponentially as more devices subscribe to the multicast domain.

3.2.2 Kernel network latency

The kernel network latency, visualized in Figure 3.2, shows that the propagation delay decreases as the packet size decreases. This decrease is expected as the amount of data to propagate through the network stack has decreased. The same is expected and seen in Figure 3.3, where there is noise on the network link. We note that architecture B handles the packet size increase better, with a lower increase in mean propagation time. The overall mean propagation time being lower in architecture B may be attributed to the higher CPU clock speed.

Furthermore, comparing architecture B to architecture A we see that it handles network noise, unpredictable packet bursts, more predictably. From the results in Figures 3.3b, 3.3d, and 3.3f, it can be noted that the network disturbances affect the different packet

sizes equally, not exceeding $1.75\,\mathrm{ms}$ and maintaining a steady standard deviation in all three tests. While architecture A, in Figures 3.3a, 3.3c, and 3.3e, showcases increased jitter when the packet size increases. This property makes it less suitable for packet sizes $\geq 20\,\mathrm{ms}$ on a noisy network.

The latency burst deviations, in Figures 3.2c, 3.2e, and 3.3c, are separated in time and by 25000 packets, as mentioned in Section 3.1.2. Important to note is the difference in received data between the tests as the packet size is different, meaning that the $20\,\mathrm{ms}$ test receives four times more data than the $5\,\mathrm{ms}$ case between the bursts. The cause of these deviations has not been identified. We have theorized that it may be CPU caching, a cache capacity miss will trigger a cache block replacement event, degrading system performance. That does, however, not explain why the same behaviour is seen at different data size intervals in the two tests. Another hypothesis was a periodic task performing a computation. It does not hold up either as the only periodic indicator is the number of packets between the tests, and the tests differ in time between the peaks. All of this leads to the peaks being an undefined behaviour for which we have no further explanation.

3.2.3 Full system latency

Analog measurements

If we look at the analog results in Figure B.1, we can see how the system delay decreases as we decrease the *pipeline-latency*. Reviewing all the tests, we see a variable extra delay that applies to the measurements. These are summarized in Tables 3.3 and 3.4. We believe this extra delay is the *period-time* in ALSA, which corresponds to the *period-time* in Table 2.1. If one adds the *period-time* and *pipeline-latency* in any of the tests in Table 2.1, we see that it very closely corresponds to the actual latency we have measured. There is one exception, test 7, where the *latency-time* is $5\,\mathrm{ms}$, but the extra delay is $15\,\mathrm{ms}$. If we compare test 7 and 8 we see that lowering the *jitter-latency*, which is the size of the network jitter buffer, makes the result align with the other measurements. We believe this extra time gets added to the total latency as a result of the system load, causing propagation delays. When analog audio enters the system, it is sampled and then made available through ALSA. The system extracts *period-time* ms at a time from ALSA, which is then timestamped upon extraction by the `AlsaSrc`. This pre-buffering means that the audio is already delayed by *period-time* ms when entering the pipeline, causing our extra latency.

Hypothesis

Our standing hypothesis is that the pipeline does not take into account the *period-time* of the ALSA input, and as such, the measured pipeline latency follows Equation (3.1).

If we now look back to the results from Tables 3.3 and 3.4 and cross-reference the differences between corresponding tests on different architectures and different networks some patterns start to emerge.

When comparing architecture A between the noisy and the silent network, we see that the extra delay always corresponds to at least the *period-time* of that test. Sometimes the extra delay is greater, test 7 is a clear outlier, but as previously mentioned, this is likely due to the relatively large *jitter-buffer* in test 7, which prevents GStreamer from decreasing the

$$\begin{gathered}\Delta t_r = \Delta t_c + t_p + r, \\ 20 \leq t_c \leq 100,\ 0 \leq t_p < 20,\ 0 < r\end{gathered} \tag{3.1}$$

t_r	measured latency
t_c	configured latency
t_p	*period-time*
r	rest

pipeline latency. When the jitter-buffer is shrunk in test 8, it brings the results back in line. We can also see how the resulting rest r in the noisy tests is generally higher than that of the silent network. This disparity suggests that some of the traffic on the noisy network is causing higher propagation delays.

If we look at architecture B, we see the same behaviour. However, the results on the noisy network show, in general, a greater rest r than that of architecture A.

We can see that as the configured latency decreases the rest increases, this is especially evident in the noisy network. In test 9, we see a difference between t_c and t_r of $15\,\mathrm{ms}$ on architecture B, which means a rest r of $10\,\mathrm{ms}$ while the difference is $5\,\mathrm{ms}$ for architecture A in the same test. Unlike previous tests such as tests 1 through 5, we see that as the *pipeline-latency* decreases, and the load increases, the rest r also increases. The difference is, however, less pronounced on the silent network where the systems approximately perform the same. It is only in the final test 9 and 10 on the silent network where the architectures differ, and architecture B performs worse than architecture A. This behaviour is likely due to the single-core nature of architecture B, architecture A can instead offload the extra network traffic to a different core.

Digital measurements

Developing the test for the full system latency was trickier than we thought. Using ALSA as the abstraction interface between producers and consumers created problems where our streams were buffered, and therefore our audio stream latency was higher than in the analog tests. As mentioned in Section 3.1.3, we did not proceed with these tests due to a lack of understanding of how ALSA buffered audio. In the end, however, we used a combination of the intra-pipeline results, the analog system results, and the difference therebetween to deduce buffer sizes of the different ALSA buffers. This analysis, in combination with the relative timestamps of the intra-pipeline test, told us where time was spent in the system.

3.2.4 CPU Scheduling

We have seen apparent differences between the different scheduling algorithms, and when looking over the results, we agree with Y. Wang [22]. Time deterministic scheduling is the better solution. As mentioned in the literature study in Section 1.3.1, the author suggests a scheduling algorithm that takes the time determinism of live audio into account. The introduced algorithm, TDCS, is believed possible to incorporate into modern GPOS' and is described at length in the thesis [22].

In our results, we see a clear advantage for RR scheduling in the digital tests. If the system is allowed to schedule according to CFS, we see complete disarray in the pipeline. This chaos can be seen in Figure 3.11e, where the system does not supply the application with execution time when it is needed. In turn, causing the audio pipeline to deteriorate quickly and miss its deadlines, thus increasing the pipeline jitter and negatively impacting the audio quality. If we instead look at EDF in Figure 3.11b, we see that it can perform well, but it must be carefully tuned to the properties of the pipeline. Otherwise, it also deteriorates, as seen in Figure 3.11c. If there is no regard taken to the properties of the pipeline configuration, a situation like the one in Figure 3.11d could occur. It is also not feasible to reconfigure the system scheduling each time the pipeline configuration changes. There are also other constraints to the Linux EDF implementation, such as a process not being able to spawn new threads once scheduled.

As seen in these results, the most consistent scheduling algorithm is RR, which has performed sufficiently in all our tests. Looking at the definitions of the different described algorithms, we see why this might be. CFS maximizes the running time of every application, EDF guarantees the specified time allocations every scheduling cycle, and RR provides equal execution time for every process in the same priority bracket. Wang describes how TDCS calculates optimal scheduling for processes in a layered approach, there are high priority processes that are provided a truly time deterministic schedule, and all low priority processes are scheduled according to a best effort scheme. Comparing this to our different algorithms described above, and in Section 2.1.2, we see that CFS is too general and does not provide any of the time determinism required. EDF provides it, almost to a fault, where the configuration needs to be spot on for it to function. However, it is the closest in spirit to time determinism as in TDCS, which also utilizes static configuration parameters for each scheduled process. RR provides the least configurable implementation of "real-time" scheduling. It provides a process hierarchy to provide some processes a recurring time slice while other processes get the scraps. While it does not take the required execution time of each of the high priority applications into account, it gives all processes of the same priority an equal amount of execution time. This prioritization means that no process gets left behind and prevents a process from completely hogging the CPU resources. A similar behaviour could seemingly be achieved with EDF, but then the deadline would need to be set very close to the period time so that the process receives a new execution period when the next audio period arrives.

Note that our audio consumer, when running under RR, is run with a higher priority than our producer on the sending system, allowing the consumer to pre-empt the producer. This prioritizes the execution of in-house applications (consumers) over third-party applications (producers).

3.2.5 Embedded systems

The resources on the embedded systems used to perform our tests have been limited in the scope of real-time audio processing. When comparing our work to that of Y. Wang[22] this is a clear difference. His tests are performed on workstations running the most popular GPOS' or even state of the art audio manipulation hardware.

Running our test-suites on the embedded systems, while running the pipeline being analyzed, has most likely contributed an additional amount of jitter. This sensitivity adds

additional importance to a comparative analysis of two separate systems, where conclusions may be drawn irrespective of an additional jitter contribution. Further, the jitter contribution has been assumed to be negligible in our test cases, due to seeing similar behaviours in both our analog and digital tests where the analog tests are not influenced by additional jitter from digital measurements.

3.2.6 Intra-pipeline latency

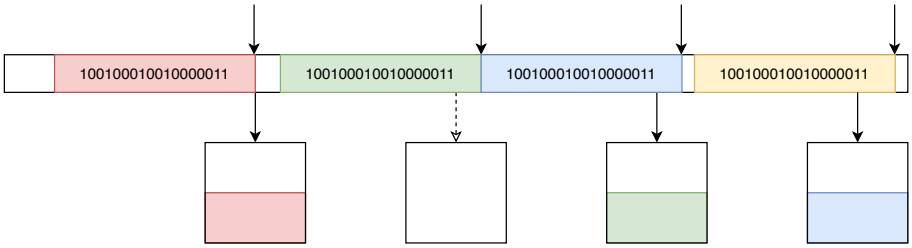
Looking over our results, we are inclined to agree with Y. Wang, who states that "The timing performance of scheduling the audio processing tasks is the key to reduce excessive buffer size." [22, p. 156]. This quote relates to consumer jitter, as explained below.

Having complete insight into the audio path has allowed us to dig as deep as we want into some issues. Being able to test minor parts of the whole system extensively has contributed to giving us a greater overall insight by widening the possible scope.

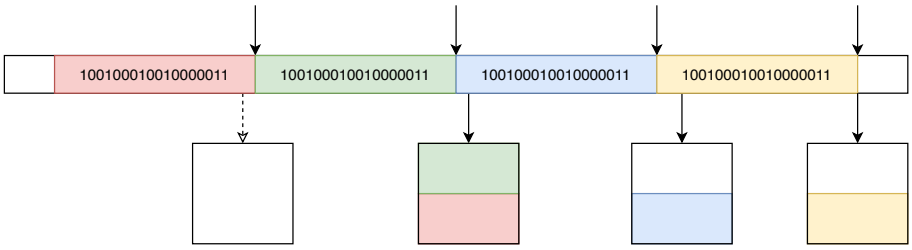
When studying the measurements from architecture A in Figures 3.12a and 3.13a, we see, as described in the result, repeated `AlsaSrc` timestamp lines. Provided that the pipeline has a large enough configured *pipeline-latency*, the pipeline can account for these delays internally without any outgoing jitter. If the deadlines instead tighten, as in test 2, we see how the requirement for time deterministic scheduling increases as the pipeline can no longer account for those delays. These buffer delays, which manifest as repeated `AlsaSrc` timestamp lines, seem to originate in the interface between the producer (*latsend*) and the consumer (audio sender) through ALSA. Interestingly we only see these on architecture A and not on architecture B in Figures 3.12 and 3.13. If we look back on Figure 3.14, this can also be observed.

Our theory is that this is due to process scheduling on single and multi-core processors. In Figure 3.15, we try to showcase how this can occur. In Figure 3.15a, we see that provided a perfect consumer consuming with an exact *period-time* the producer can never manage to have two audio frames buffered at any given time. Looking at Figures 3.14a and 3.14b, we see that the producer is indeed nearly perfect in its audio buffer handovers. If we instead look at Figure 3.15b, we see the opposite case. Provided a perfect producer that produces audio frames with an exact *period-time*, as seen in Figure 3.14a, a consumer not following the same *period-time* but instead polling sometimes before or after, a situation could occur where the consumer misses the introduction of a new audio frame into the audio buffer. When it then polls again after a period slightly longer than a *period-time*, it allows the producer to introduce yet another audio frame causing the producer to be able to extract two audio frames at the next poll, as seen in Figure 3.14c. How could this uneven polling behaviour occur? By running the producer and the consumer without proper synchronization, as mentioned before, the lack of true time determinism in the process scheduling cannot account for the periodic nature of the audio production. Instead, it may execute in parallel and out of sync on a multi-core system. This problem is seemingly alleviated on a single-core system as architecture B does not indicate any of these issues.

Through all the digital testing, time sync has been critical, as mentioned in Section 2.2.3. Without it, there would be very little we could assert about the audio transmissions between the sender and receiver. At some points during the project, the sync has failed with at first seemingly arcane results. For example, audio arriving at the receiver before the sender had extracted it from the source. In the end all tests with such errors were discarded, and



(a) Example of uneven producer.



(b) Example of uneven consumer.

Figure 3.15: Comparison of what happens if there is an uneven handover from the producer or polling from the consumer.

all remaining results in the report fulfill the time sync requirement.

Our test suite, Table 2.1, was created very early in the process and could probably be improved on now that we have gathered a better understanding of the system, and what its limitations are. That said, it has been invaluable to have a predetermined standard for all our tests allowing us to compare old results from the beginning of the research phase to results from the very end. It has provided us with excellent continuity, and as a result of this, we have been able to create graphs such as the composite graphs from Figures 3.12 and 3.13. While these might not show the complete tests, we have tried to make them as representative as possible for each test by slicing out defining parts of each test result. We could have included the whole graphs for each test but, in the interest of conserving space and providing both the reader and us with a better overview, the decision was made to create said composite graphs.

Chapter 4
Conclusions

We conclude that architecture A would be a better fit for achieving lower latencies if the producer and consumer were synchronized. Currently, architecture B showcases room for achieving much lower latencies and is less affected by the noisy network, as seen in the kernel network latency, intra-pipeline latency, and analog latency tests.

Achieving synchronization between the producer and consumer on architecture A may currently not be possible if ALSA is used. To continue using ALSA as the middle-man between the producer and consumer, one would have to introduce a scheduling algorithm that accounts for time determinism such that there is no need to bypass ALSA. Providing such synchronization may resolve the current mismatch in the production and consumption of audio periods. Furthermore, removing the ALSA abstraction would remove the currently simple interface used by third-party producers.

Looking at the network stack results in Section 3.1.2, we see clear differences between the architectures. Regardless of the peaks in Figures 3.2c and 3.2e, we conclude that architecture B has a better response to increases in packet size. As in all tests, we see more jitter in our test results when performing the tests on the noisy network.

Turning our attention to the analog latency and resource utilization results, we conclude that large amounts of control traffic which the systems process cause an overall reduction in the performance of the system. Close to 5 to $10\,\%$ in the case of architecture B in Figures 3.7d and 3.9b. We also see a definite improvement in the CPU-utilization when mDNS is disabled. Our results reflect this in Figures 3.6c, 3.6d, and 3.9. Overall, architecture B performs better in the analog tests.

When looking through all of our results and discussion, it is clear that we see significant differences based on the network load. However, it is only network traffic that needs to be processed, such as mDNS, that severely affects system performance. Dividing the systems into smaller cells may alleviate some of these issues as all units would not need full knowledge of all other available systems. We can draw parallels between the shift made long ago in routing protocols from the *link state* to the *distance vector* routing algorithm [13]. *Link state* keeps a complete table of all routes in the whole network. It quickly

got out of hand, and performance plummeted as the size of modern networks grew. Instead, the choice was made to move to *distance vector* where each router only keeps track of its neighbours and only updates these or receives updates from these as their configuration changes. The same applies in the case of our network-connected devices. The more units we add, the more units each unit has to keep track of. It might be better to try implementing an approach more akin to that of *distance vector* instead. While the zero-configuration networking nature of mDNS is attractive for the sake of usability, one might wonder if the performance degradation is worth it when trying to achieve low latency.

4.1 Future work

To improve the latency for architecture A, the multi-core system, we believe that scheduling algorithms that account for the time determinism of the applications will reduce, if not eliminate, the extra period buffering discussed earlier. Looking into implementing the TDCS algorithm [22] for the Linux kernel would be a valiant effort, and provide an excellent base for one or more master's theses.

It could be interesting to research creating smaller "cells" of broadcast domains to alleviate some of the mDNS traffic barrage we are seeing. If these cells were then made to communicate in between, more sparsely or in batches, a much better traffic situation could be achieved. Alternatively, one could gather inspiration from *link state* and *distance vector* and try to solve the issue similarly.

Looking into including the producer into the consumer, one could eliminate some of the synchronization issues. They seem to arise when the producer and consumer run on different clocks or priority levels, or each side of the ALSA loopback device on multi-core systems.

One of the initial research questions set out to answer whether other streaming frameworks could provide a better foundation for lower latencies. GStreamer provides usability, but it is probably not the highest performing nor the most resource-efficient framework. Looking at the lower latencies, we saw that GStreamer had a noticeable overhead in packet processing of smaller packets. A comparative study of low latency audio streaming using different media frameworks may provide beneficial information for low latency audio streaming on embedded devices.

References

[1] The Linux Kernel documentation. *Writing an ALSA driver*. URL: https://www.kernel.org/doc/html/v4.17/sound/kernel-api/writing-an-alsa-driver.html (visited on 03/20/2019).

[2] E-Linux. *Kernel Timer Systems*. URL: https://elinux.org/Kernel_Timer_Systems (visited on 05/28/2019).

[3] GStreamer. *Documentation - Pads and Capabilities*. URL: https://gstreamer.freedesktop.org/documentation/application-development/basics/pads.html (visited on 03/19/2019).

[4] GStreamer. *Documentation - Probes*. URL: https://gstreamer.freedesktop.org/documentation/design/probes.html (visited on 03/19/2019).

[5] GStreamer. *gst-plugins-base*. 2019. URL: https://gitlab.freedesktop.org/gstreamer/gst-plugins-base (visited on 06/04/2019).

[6] GStreamer. *GStreamer: open source multimedia framework*. 2019. URL: https://gstreamer.freedesktop.org/ (visited on 12/18/2019).

[7] GStreamer. *rtpsession*. URL: https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-good/html/gst-plugins-good-plugins-rtpsession.html (visited on 03/26/2019).

[8] GStreamer. *rtpstorage*. URL: https://gstreamer.freedesktop.org/data/doc/gstreamer/head/gst-plugins-good/html/gst-plugins-good-plugins-rtpstorage.html (visited on 03/26/2019).

[9] Linux Journal. *Kernel Korner - Sleeping in the Kernel*. URL: https://www.linuxjournal.com/article/8144 (visited on 05/28/2019).

[10] Andrew Kryczka, Ahsan Arefin, and Klara Nahrstedt. "AvCloak: A Tool for Black Box Latency Measurements in Video Conferencing Applications". In: Dec. 2013, pp. 271–278. DOI: 10.1109/ISM.2013.52.

[11] Michael Lester and Jon Boley. "The Effects of Latency on Live Sound Monitoring". In: *Audio Engineering Society Convention 123*. Oct. 2007. URL: http://www.aes.org/e-lib/browse.cfm?elib=14256.

[12] *Matrix:Module-aloop*. May 2019. URL: https://alsa-project.org/wiki/Matrix:Module-aloop.

[13] John M. Mcquillan. "The Birth of Link-State Routing". In: *IEEE Annals of the History of Computing* 31.1 (2009), pp. 68–71. DOI: 10.1109/mahc.2009.17. URL: https://ieeexplore.ieee.org/document/4802433.

[14] Linux Kernel Organization. *networking - timestamping*. URL: https://www.kernel.org/doc/Documentation/networking/timestamping.txt (visited on 03/26/2019).

[15] John G. Proakis and Dimitris G. Manolakis. *Digital signal processing: principle, algorithms, and applications*. Pearson Prentice Hall, 2007.

[16] *Sched(7) Linux Programmer's Manual*. 5.10. Mar. 2019.

[17] H. Schulzrinne et al. *RTP: A Transport Protocol for Real-Time Applications*. RFC 1889. IETF, Jan. 1996.

[18] Andrew S. Tanenbaum and Herbert Bos. *Modern operating systems*. 4th ed. Pearson, 2017.

[19] Linus Torvalds. *Linux Kernel*. 2019. URL: https://git.kernel.org/ (visited on 03/29/2019).

[20] Max Vilimpoc. *Ku-latency*. 2008. URL: https://vilimpoc.org/research/ku-latency/ (visited on 02/12/2019).

[21] *Vmstat*. Mar. 2019. URL: https://linux.die.net/man/8/vmstat.

[22] Yonghao Wang. "Low Latency Audio Processing". PhD thesis. School of Electronic Engineering and Computer Science, Queen Mary University of London, 2017.

[23] Aubrey J Yates. "Delayed auditory feedback." In: *Psychological bulletin* 60.3 (May 1963), p. 213.

Appendix A
Implementations

A.1 Ku-latency

The full implementation of the modified Ku-latency, originally by M. Vilimpoc [20], is published online and available at: `https://gist.github.com/eHammarstrom/7444aeeffcddb21c34eabdd1cd61c514`.

A.2 Latsend

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <gst/gst.h>
#include <glib.h>
#include <glib-utils.h>

#include <time.h>

#include "latency.h"
#include "intprops.h"

#define STR(a) #a
#define CHECK(code)                \
  do {                    \
    if (0 == (code)) {              \
      printf("Internal error: NULL/FALSE from \n\t\"%s\"\n", STR(code))
    ; \
    }                \
  } while (0)

```

```
21 #define AUDIO_CAPS            ("audio/x-raw, format=S16LE, channels=2,
       rate=48000")
22 #define ALSA_DEVICE           ("audiosink_0") // loopbacksrc_0 <>
       audiosink_0
23 #define ALSA_SKEW_DRIFT_TOLERANCE (15000)
24
25 // #define INSPECT_JITTER
26
27 typedef struct {
28   GMainLoop *loop;
29   guint32 idx;
30 #ifdef INSPECT_JITTER
31   FILE *logfd;
32 #endif
33 } Context;
34
35 typedef struct {
36   guint32 __leading_ones;
37   guint32 index;
38   struct timespec time;
39 } BUFData;
40
41 /* This function is called when an error message is posted on the bus
       */
42 static void error_cb(GstBus *bus, GstMessage *msg, Context *ctx) {
43   GError *err;
44   gchar *debug_info;
45
46   gst_message_parse_error(msg, &err, &debug_info);
47   g_printerr("Error received from element %s: %s\n", GST_OBJECT_NAME(
       msg->src), err->message);
48   g_printerr("Debugging information: %s\n", debug_info ? debug_info : "
       none");
49   g_clear_error(&err);
50   g_free(debug_info);
51
52   g_main_loop_quit(ctx->loop);
53 }
54
55 GstPadProbeReturn stamp_buffer_cb(GstPad *pad, GstPadProbeInfo *info,
       Context *ctx)
56 {
57   (void *) pad;
58   GstBuffer *buffer;
59
60   buffer = GST_PAD_PROBE_INFO_BUFFER(info);
61
62   struct timespec time;
63   clock_gettime(CLOCK_REALTIME, &time);
64
65   BUFData buf_data = { 0xFFFFFFFF, ctx->idx, time };
66   ctx->idx += 1;
67
68   // modify buffer by adding timestamp
69   GstMapInfo map;
70   if (gst_buffer_map(buffer, &map, GST_MAP_WRITE)) {
```

```
71      memcpy(map.data, &buf_data, sizeof(BUFData));
72      gst_buffer_unmap(buffer, &map);
73    }
74
75  #ifdef INSPECT_JITTER
76    fprintf(ctx->logfd, "%ld.%.9ld\n", time.tv_sec, time.tv_nsec);
77  #endif
78
79    return GST_PAD_PROBE_OK;
80  }
81
82  int main(int argc, char* argv[])
83  {
84    GMainLoop *loop;
85    GstElement *pipeline, *audiotestsrc, *capsfilter, *alsasink;
86    GstPad *sourcepad;
87    GstCaps *caps;
88    Context ctx;
89
90    memset(&ctx, 0, sizeof(ctx));
91
92  #ifdef INSPECT_JITTER
93    ctx.logfd = fopen("latsend-times.out", "w+");
94  #endif
95
96    gst_init(&argc, &argv);
97
98    loop = g_main_loop_new(NULL, FALSE);
99    ctx.loop = loop;
100
101   CHECK(pipeline = gst_pipeline_new(NULL));
102
103   // https://gstreamer.freedesktop.org/documentation/application-
      development/advanced/pipeline-manipulation.html#manually-adding-or-
      removing-data-fromto-a-pipeline
104
105   CHECK(audiotestsrc = gst_element_factory_make("audiotestsrc", "src"))
      ;
106   g_object_set(audiotestsrc, "is-live", TRUE, NULL);
107   g_object_set(audiotestsrc, "do-timestamp", TRUE, NULL);
108
109 #define ALSA_LATENCY_TIME (20000)
110 #define ALSA_BUFFER_TIME (40000)
111
112   // 960 samples = 3840 bytes / 192 bytes = 20ms
113   // 480 = 10 ms // 240 = 5 ms // 120 = 1 ms
114   g_object_set(audiotestsrc, "samplesperbuffer", 960, NULL);
115   g_object_set(audiotestsrc, "wave", 1, NULL);        // 1 ==
      GST_AUDIO_TEST_SRC_WAVE_SQUARE
116
117   // attach modifying probe to audiotestsrc
118   GstPad *audiotestsrc_srcpad;
119   CHECK(audiotestsrc_srcpad = gst_element_get_static_pad(audiotestsrc,
      "src"));
120   gst_pad_add_probe(audiotestsrc_srcpad,
121             GST_PAD_PROBE_TYPE_BUFFER,
```

```
122             (GstPadProbeCallback)stamp_buffer_cb,
123             &ctx, NULL);
124
125   CHECK(alsasink = gst_element_factory_make("alsasink", "sink"));
126   g_object_set(alsasink, "device", ALSA_DEVICE, NULL);
127   g_object_set(alsasink, "provide-clock", FALSE, NULL);
128   g_object_set(alsasink, "sync", FALSE, NULL);
129   g_object_set(alsasink, "latency-time", ALSA_LATENCY_TIME, NULL);
130   g_object_set(alsasink, "buffer-time", ALSA_BUFFER_TIME, NULL);
131   g_object_set(alsasink, "drift-tolerance", (gint64)
        ALSA_SKEW_DRIFT_TOLERANCE, NULL);
132
133   caps = gst_caps_from_string(AUDIO_CAPS);
134   CHECK(capsfilter = gst_element_factory_make("capsfilter", "
        srcsinkcaps"));
135   g_object_set(capsfilter, "caps", caps, NULL);
136
137   gst_bin_add_many(GST_BIN(pipeline), audiotestsrc, capsfilter,
        alsasink, NULL);
138
139   if (gst_element_link_many(audiotestsrc, capsfilter, alsasink, NULL)
        != TRUE) {
140     g_printerr("Elements could not be linked.\n");
141     gst_object_unref(pipeline);
142     return -1;
143   }
144
145   GstBus *bus;
146   bus = gst_element_get_bus(pipeline);
147   gst_bus_add_signal_watch(bus);
148   g_signal_connect(G_OBJECT(bus), "message::error", (GCallback)error_cb
        , &ctx);
149   gst_object_unref(bus);
150
151   gst_element_set_state(pipeline, GST_STATE_PLAYING);
152
153   g_main_loop_run(loop);
154
155   gst_element_set_state(pipeline, GST_STATE_NULL);
156   gst_object_unref(GST_OBJECT(pipeline));
157   return 0;
158 }
```

Listing A.1: latsend.c

A.3 Latrecv

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <gst/gst.h>
5 #include <glib.h>
6 #include <glib-utils.h>
7
8 #include "latency.h"
```

```
#include "intprops.h"

#define STR(a) #a
#define CHECK(code)           \
  do {                  \
    if(0 == (code)) {       \
      printf("Internal error: NULL/FALSE from \n\t\"%s\"\n", STR(code))
    ; \
    }               \
  } while(0)

#define AUDIO_CAPS ("audio/x-raw, format=S16LE, channels=2, rate
    =48000")
#define ALSA_DEVICE ("loopbacksrc_0") // loopbacksrc_0 <> audiosink_0
#define ALSA_SKEW_DRIFT_TOLERANCE (15000)

static void error_cb(GstBus *bus, GstMessage *msg, gpointer lul)
{
  GError *err;
  gchar *debug_info;

  gst_message_parse_error(msg, &err, &debug_info);
  g_printerr("Error received from element %s: %s\n", GST_OBJECT_NAME(
    msg->src), err->message);
  g_printerr("Debugging information: %s\n", debug_info ? debug_info : "
    none");
  g_clear_error(&err);
  g_free(debug_info);
}

int main(int argc, char* argv[])
{
  GMainLoop *loop;
  GstElement *pipeline, *alsasrc, *capsfilter, *fakesink;
  GstPad *sourcepad;
  GstCaps *caps;

  gst_init(&argc, &argv);

  loop = g_main_loop_new(NULL, FALSE);

  CHECK(pipeline = gst_pipeline_new(NULL));

  CHECK(alsasrc = gst_element_factory_make("alsasrc", "src"));
  g_object_set(alsasrc, "device", ALSA_DEVICE, NULL);
  g_object_set(alsasrc, "provide-clock", FALSE, NULL);
  g_object_set(alsasrc, "latency-time", 20000, NULL);
  g_object_set(alsasrc, "buffer-time", 40000, NULL);

  caps = gst_caps_from_string(AUDIO_CAPS);
  CHECK(capsfilter = gst_element_factory_make("capsfilter", "
    srcsinkcaps"));
  g_object_set(capsfilter, "caps", caps, NULL);

  CHECK(fakesink = gst_element_factory_make("fakesink", "sink"));

```

```
60   gst_bin_add_many(GST_BIN(pipeline), alsasrc, capsfilter, fakesink,
       NULL);
61
62   GstPad *alsasrc_srcpad;
63   CHECK(alsasrc_srcpad = gst_element_get_static_pad(alsasrc, "src"));
64
65   // Initialize latency ctxt
66   LatencyContext *lat_ctxt = init_lat_ctx("ALSASRC_PROBE", FALSE);
67
68   // Attach a pad probe
69   gst_pad_add_probe(alsasrc_srcpad,
70             GST_PAD_PROBE_TYPE_BUFFER,
71             (GstPadProbeCallback)probe_callback,
72             lat_ctxt, NULL);
73
74   if(gst_element_link_many(alsasrc, capsfilter, fakesink, NULL) != TRUE
       ) {
75     g_printerr("Elements could not be linked.\n");
76     gst_object_unref(pipeline);
77     return -1;
78   }
79
80   GstBus *bus;
81   bus = gst_element_get_bus(pipeline);
82   gst_bus_add_signal_watch(bus);
83   g_signal_connect(G_OBJECT(bus), "message::error", (GCallback)error_cb
       , NULL);
84   gst_object_unref(bus);
85
86   gst_element_set_state(pipeline, GST_STATE_PLAYING);
87
88   g_main_loop_run(loop);
89
90   gst_element_set_state(pipeline, GST_STATE_NULL);
91   gst_object_unref(GST_OBJECT(pipeline));
92   return 0;
93 }
```

Listing A.2: latrecv.c

A.4 Intra-pipeline probe example

```
1 #ifdef LAT_PROBE_RECV_QUEUE
2 GstPad *queue_srcpad;
3 check(queue_srcpad = gst_element_get_static_pad(queue, "src"));
4
5 // Initialize latency ctxt
6 LatencyContext *queue_ctxt = init_lat_ctx("QUEUE_PROBE", "netrecv",
    FALSE);
7
8 gst_pad_add_probe(queue_srcpad,
9 GST_PAD_PROBE_TYPE_BUFFER,
10 (GstPadProbeCallback) probe_callback,
11 queue_ctxt, NULL);
```

```
#endif
```

Listing A.3: Probe attached to a Queue GStreamer element.

A.5 Shared latency code

```
#ifndef _LATENCY_H_
#define _LATENCY_H_

#include <glib.h>
#include <gst/gst.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>


#define LAT_PROBE_SEND_ALSA
#define LAT_PROBE_SEND_RTP

#define LAT_PROBE_RECV_UDP
// #define LAT_PROBE_RECV_RTP
#define LAT_PROBE_RECV_ALSA
#define LAT_PROBE_RECV_SSRCSELECTOR
#define LAT_PROBE_RECV_AUDIOCONVERT_NETSRC
#define LAT_PROBE_RECV_PRIO_SELECTOR
#define LAT_PROBE_RECV_QUEUE

#define NUM_SAMPLES (10000)

// #define LAT_SIGNAL_RECV_QUEUE

#define UNUSED(x) (void)(x)

#define ROLLING_AVG_SIZE (50)

#define PREAMBLE64 (0xFFFFFFFFFFFFFFFF)
#define PREAMBLE32 (0xFFFFFFFF)

typedef struct {
  char *name;
  char *program;
  gboolean flip_data;

  guint32 prev_index;

  char *logbuffer;
  size_t logsize;
  FILE *logfile;
  long long logged_samples;

  long long acc_diff;
  long long num_samples;
  long long roll_avg[ROLLING_AVG_SIZE];
  struct timespec min;
  struct timespec max;
```

```
50 } LatencyContext;
51
52 typedef struct {
53   char *name;
54   char *message;
55   guint32 counter;
56   GstElement *alsasink;
57   GstElement *queue;
58 } SignalContext;
59
60 typedef struct {
61   guint32 __leading_ones;
62   guint32 index;
63   struct timespec time;
64 } TimeData;
65
66 LatencyContext* init_lat_ctx(char*, char*, gboolean);
67
68 void queue_sig_callback(GstElement *, SignalContext *);
69
70 GstPadProbeReturn probe_callback(GstPad *, GstPadProbeInfo *,
     LatencyContext *);
71
72 #endif
```

Listing A.4: latency.h

```
1  #include <byteswap.h>
2  #include <unistd.h>
3  #include <string.h>
4
5  #include "latency.h"
6  #include "intprops.h"
7
8  #if 0
9  static void print_byte_as_bits(char val) {
10   for(int i = 7; 0 <= i; i--) {
11     g_print("%c",(val &(1 << i)) ? '1' : '0');
12   }
13 }
14
15 static void print_bits(char * ty, char * val, unsigned char * bytes,
     size_t num_bytes) {
16   g_print("(%*s) %*s = [ ", 15, ty, 16, val);
17   for(size_t i = 0; i < num_bytes; i++) {
18     print_byte_as_bits(bytes[i]);
19     g_print(" ");
20   }
21   g_print("]\n");
22 }
23
24 #define SHOW(T, V)                                                \
25  do {                                                    \
26   T x = V;                                                 \
27   print_bits(#T, #V, (unsigned char *) &x, sizeof(x));
          \
```

```
 } while (0)
#endif

#if 0
static int
timespec_cmp(struct timespec a, struct timespec b)
{
 if (a.tv_sec == b.tv_sec) {
  if (a.tv_nsec == b.tv_nsec)
   return 0;
  else if (a.tv_nsec > b.tv_nsec)
   return 1;
  else
   return -1;
 } else if (a.tv_sec > b.tv_sec)
  return 1;
 else
  return -1;
}
#endif

static struct timespec
timespec_sub(struct timespec a, struct timespec b)
{
 struct timespec r;
 time_t rs = a.tv_sec;
 time_t bs = b.tv_sec;
 int ns = a.tv_nsec - b.tv_nsec;
 int rns = ns;

 if (ns < 0) {
  rns = ns + 1000000000;
  if (rs == TYPE_MINIMUM(time_t)) {
   if (bs <= 0)
    goto low_overflow;
   bs--;
  } else
   rs--;
 }

 if (INT_SUBTRACT_OVERFLOW(rs, bs)) {
  if (rs < 0) {
  low_overflow:
   rs = TYPE_MINIMUM(time_t);
   rns = 0;
  } else {
   rs = TYPE_MAXIMUM(time_t);
   rns = 999999999;
  }
 } else
  rs -= bs;

 r.tv_sec = rs;
 r.tv_nsec = rns;
 return r;
}
```

```
84
85 #if 0
86 static long long
87 get_avg(long long *arr)
88 {
89  long long sum = 0;
90
91  for (size_t i = 0; i < ROLLING_AVG_SIZE; ++i)
92   sum += arr[i];
93
94  return sum / ROLLING_AVG_SIZE;
95 }
96
97 static void
98 insert_avg(long long *arr, long long stamp)
99 {
100  for (size_t i = ROLLING_AVG_SIZE-1; i >= 1; --i)
101   arr[i] = arr[i - 1];
102
103  arr[0] = stamp;
104 }
105 #endif
106
107 static void
108 debug_latency(const char *prefix, guint32 index,
109        struct timespec a, struct timespec b, LatencyContext *ctx)
110 {
111  // we have stopped logging
112  if (ctx->logfile == NULL)
113   return;
114
115  // skip alsa sound duplication
116  if (ctx->prev_index == index)
117   return;
118
119  ctx->prev_index = index;
120
121  struct timespec diff = timespec_sub(a, b);
122
123  ctx->logged_samples += 1;
124
125  UNUSED(prefix);
126  fprintf(ctx->logfile, "%.5u,%ld.%.9ld\n", index, diff.tv_sec, diff.
     tv_nsec);
127
128  if (ctx->logged_samples >= NUM_SAMPLES) {
129   fclose(ctx->logfile); // done logging, trigger memmap FILE->char*
     buffer
130
131   char *str_logfile = g_malloc0(1024);
132   sprintf(str_logfile, "/var/volatile/log/%s%s.log", ctx->program, ctx
     ->name);
133   FILE *real_logfile = fopen(str_logfile, "w+");
134
135   fprintf(real_logfile, "index,time\n");
136
```

```
137   fwrite(ctx->logbuffer, sizeof(char), strlen(ctx->logbuffer),
        real_logfile);
138   fclose(real_logfile);
139
140   printf("Logged to %s\n", str_logfile);
141   ctx->logfile = NULL;
142  }
143 }
144
145 static guint32
146 unflip_u32(guint32 n)
147 {
148  return __bswap_32((n >> 16) | (n << 16));
149 }
150
151 #if 0
152 static guint64
153 unflip_u64(guint64 x)
154 {
155  return __bswap_16((x) & 0xffff000000000000ull)
156       | __bswap_16((x) & 0x0000ffff00000000ull)
157       | __bswap_16((x) & 0x00000000ffff0000ull)
158       | __bswap_16((x) & 0x000000000000ffffull);
159 }
160 #endif
161
162 static TimeData* retrieve_time_data(gpointer data, gint size)
163 {
164  guint32 *walk = data;
165
166  // look for TimeData preamble
167  while (*walk != PREAMBLE32 && (gpointer) walk - data < size)
168   ++walk;
169
170  // check if we found TimeData or hit end of buffer
171  if (*walk != PREAMBLE32)
172   return NULL;
173
174  return (TimeData *) walk;
175 }
176
177 static void flip_data(TimeData *time_data)
178 {
179  struct timespec *time = &time_data->time;
180
181  time->tv_sec = unflip_u32(time->tv_sec);
182  time->tv_nsec = unflip_u32(time->tv_nsec);
183
184  time_data->index = unflip_u32(time_data->index);
185 }
186
187 /********************************** APPSINKS
      ****************************/
188
189 LatencyContext *
190 init_lat_ctx(char *name, char *program, gboolean flip_data)
```

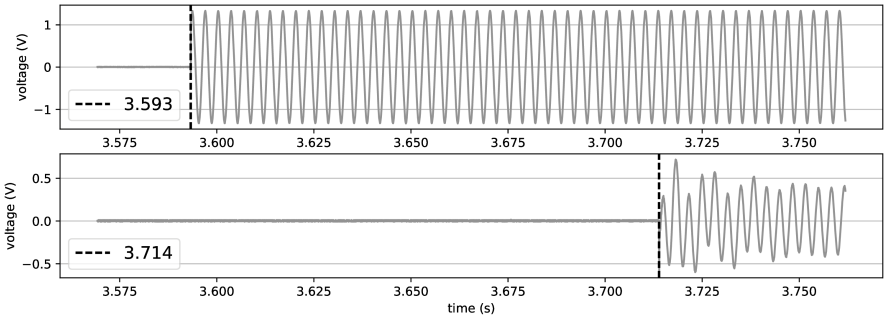
```
191 {
192  LatencyContext *ctx = g_malloc0(sizeof(LatencyContext));
193  struct timespec min = {0x7FFFFFFF, 0x7FFFFFFF};
194
195  // ptr = a pointer to the buffer char**
196  // sizeloc = size of buffer size_t*
197  // FILE *logfile = open_memstream(&ctx->logbuffer, &ctx->logsize);
198 #define BUFSIZE (1048576)
199
200  ctx->logbuffer = calloc(2 * BUFSIZE, sizeof(char));
201  ctx->logfile = (FILE *) fmemopen((void *) ctx->logbuffer, 2 * BUFSIZE,
        "w");
202
203  if (ctx->logfile == NULL) { printf("open_memstream: failed\n"); exit
       (1); }
204
205  ctx->min = min;
206  ctx->name = name;
207  ctx->program = program;
208  ctx->flip_data = flip_data;
209
210  return ctx;
211 }
212
213 struct timespec t = {0,0};
214 void queue_sig_callback(GstElement *q, SignalContext *ctx)
215 {
216  UNUSED(q);
217  ++ctx->counter;
218  // guint current_level_bytes;
219  // g_object_get(q, "current-level-bytes", &current_level_bytes, NULL);
220  // g_print("%s: %s - %u\n", ctx->name, ctx->message,
       current_level_bytes);
221
222  struct timespec curr;
223  clock_gettime(CLOCK_REALTIME, &curr);
224
225  if (timespec_sub(curr, t).tv_sec > 1) {
226   g_print("Underruns/s = %u\n", ctx->counter);
227
228   gint64 asink_buftime;
229   g_object_get(ctx->alsasink, "buffer-time", &asink_buftime, NULL);
230
231   ctx->counter = 0;
232   clock_gettime(CLOCK_REALTIME, &t);
233  }
234 }
235
236 GstPadProbeReturn probe_callback(GstPad *pad, GstPadProbeInfo *info,
       LatencyContext *ctx)
237 {
238  UNUSED(pad);
239
240  GstBuffer *buffer;
241  GstMapInfo map;
242  TimeData *time_data = NULL;
```

```
243 TimeData cp_time_data;
244
245 // check if correct pad event
246 if ((info->type & GST_PAD_PROBE_TYPE_BUFFER) == 0)
247  return GST_PAD_PROBE_OK;
248
249 buffer = GST_PAD_PROBE_INFO_BUFFER(info);
250
251 if (buffer == NULL)
252  return GST_PAD_PROBE_OK;
253
254 if (gst_buffer_map(buffer, &map, GST_MAP_READ)) {
255  time_data = retrieve_time_data((gpointer) map.data, map.size);
256  gst_buffer_unmap (buffer, &map);
257 }
258
259 if (time_data == NULL)
260  return GST_PAD_PROBE_OK;
261
262 cp_time_data = *time_data;
263
264 if (ctx->flip_data)
265  flip_data(&cp_time_data);
266
267 struct timespec namnam;
268 namnam = cp_time_data.time;
269
270 struct timespec klossen;
271 clock_gettime(CLOCK_REALTIME, &klossen);
272
273 // print latency and update LatencyContext
274 debug_latency(ctx->name, cp_time_data.index, klossen, namnam, ctx);
275
276 return GST_PAD_PROBE_OK;
277 }
```

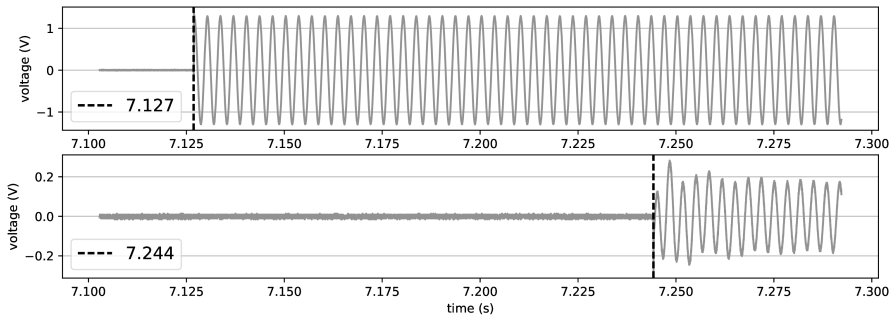
Listing A.5: latency.c

Appendix B
Result

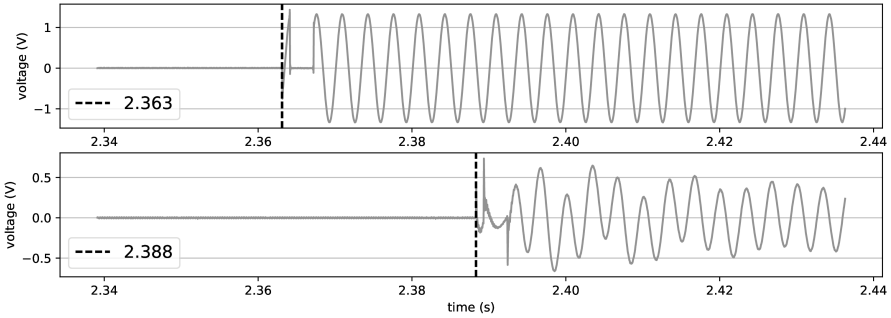
B.1 Analog measurements



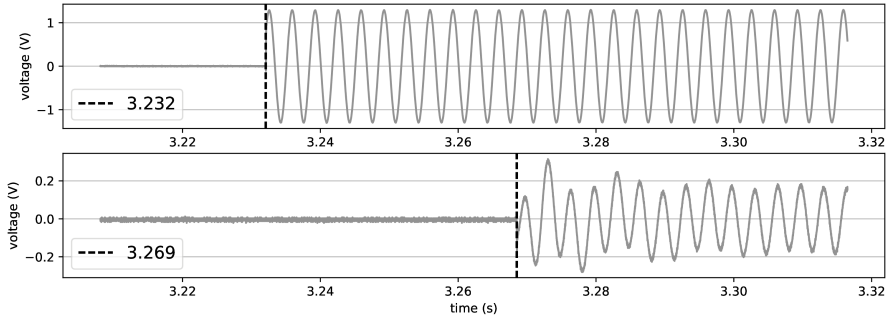
(a) Architecture A Test 1 latency.



(b) Architecture B Test 1 latency.



(c) Architecture A Test 9 latency.



(d) Architecture B Test 9 latency.

Figure B.1: System latency for test 1 and 9 on the noisy network.

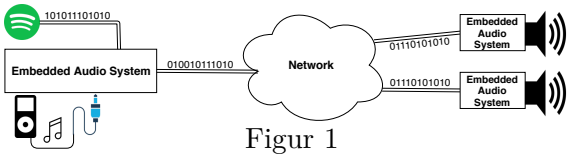
INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2019-08-29

EXAMENSARBETE Analysing the Audio Latency Contributions in a Networked Embedded Linux System
STUDENTER Gabriel Sjöberg, Emil Hammarström
HANDLEDARE Jonas Skeppstedt (LTH)
EXAMINATOR Per Andersson (LTH)

Undersökning av låg ljudlatens i inbyggda linux system

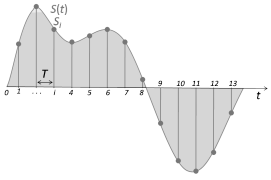
POPULÄRVETENSKAPLIG SAMMANFATTNING **Gabriel Sjöberg, Emil Hammarström**

Strömning av digitala medier via inbyggda system, små datorenheter, blir allt mer populärt. Allt från smarta högtalare till videoströmning via Google Chromecast eller Apple TV. Denna trend syns även i den industriella sektorn. Detta arbete undersöker ljudlatens i ett ljudsystem för public announcement (PA) och musik i större skala.



Figur 1

För att ett PA-system ska vara användbart och inte orsaka så kallad "Delayed Auditory Feedback (DAF)", fördröjt ljud som påverkar talförmågan, krävs en låg ljudlatens. Latens är en fördröjning mellan en händelse och registrering av den händelsen, till exempel ett eko. I stora kraftfulla datorsystem kan detta lösas med rå datorkraft. Men i mindre kraftfulla datorsystem såsom inbyggda system finns inte samma datorkraft tillgänglig. För att förstå varför digitala ljudsystem beter sig annorlunda än analoga ljudsystem behöver vi först förstå de grundläggande skillnaderna. Analogt ljud är kontinuerligt, medan digitalt ljud är diskret. Skillnaden kan observeras i Figur 2 där det grå området representerar kontinuerligt ljud och staplarna den diskreta representationen. I det diskreta fallet är staplarna separerade, de kallas på fackspråk för sampel. Dessa sampel skickas oftast i grupper.



Figur 2

I vårt examensarbete har vi undersökt den väg sampel följer i ett inbyggt ljudsystem. Detta system kan man se i Figur 1 där man till vänster ser en sändande enhet som tar emot ljud och sedan skickar ut det till andra enheter. Detta tas sedan emot av enheterna till höger som spelar upp det.

Vi har tittat på hur man kan mäta ljudlatensen i dessa system, både analogt och digitalt. Den analoga mätningen ger oss en total latens för hela systemet medan de digitala mätningarna låter oss mäta delar av resan. I de digitala mätningarna har vi sett hur mellanförvaringen av dessa sampels på olika ställen längs vägen har orsakat de största fördröjningarna. Genom att minska gruppstorleken av sampel kan man minska den totala ljudlatensen i systemet då dessa mellanförvaringar då kan krympas. Vi har observerat hur minskande gruppstorlekar leder till en ökad belastning på enheterna. Vi har jämfört två olika enhetstyper, bland annat med olika antal processorkärnor.

Vårt resultat visar att högre prestanda generellt är bättre men att det finns andra faktorer som också kan påverka ljudlatensen. En stor bov är andra program som kör på samma enhet som kan försena hanteringen av våra ljud-sampels och fördröja dessa ytterligare.