MÄLARDALEN UNIVERSITY
SCHOOL OF INNOVATION, DESIGN AND ENGINEERING
VÄSTERÅS, SWEDEN

Thesis for the Degree of Bachelor of Science in Computer Network Engineering
15 credits

GET IN SYNC WITH TSN

A Study of Partially Synchronized TSN Networks

Andreas Johansson
ajn19017@student.mdu.se

Examiner: Mohammad Ashjaei
Mälardalen University, Västerås, Sweden

Supervisor: Daniel Bujosa Mateu
Mälardalen University, Västerås, Sweden

2022-06-04

ABSTRACT

Automotive and industrial embedded systems are increasingly dependent on real-time capabilities. TSN aims to offer flexibility of the traffic by providing Ethernet with hard and soft real-time capabilities which allows for integration with other protocols in legacy systems. TSN requires the network to be fully synchronized to achieve high performance. However, there are cases where legacy systems are not able to synchronize with TSN. These systems might nonetheless be able to synchronize with each other through their legacy synchronization mechanisms.

In this thesis, we have investigated effects in terms of jitter and clock drift in endpoints by synchronizing them with each other and passing communication through an unsynchronized intermediary TSN switch. Our results revealed that with the introduction of TSN, jitter was reduced, while clock drift between endpoints and the TSN switch was introduced. The results show that negative clock drift leads to packets missing their scheduled TSN windows and positive drift leads to packets being dropped in the switch buffer queues. We proposed two solutions in order to manage the experienced clock drift. In one solution we statically changed the switch cycle, and in the other, we let the receiver node dynamically update the sending period in the sender node. In the static solution, the clock drift was reduced from negative eight microseconds per second to two nanoseconds per second. In the dynamic solution, a packet error rate of one per 100 seconds was reduced to zero errors in 19 hours.

ABBREVIATIONS

ARP	Address Resolution Protocol
AVB	Audio Video Bridging
BMCA	Best Master Clock Algorithm
CAN	Controller Area Network
CBS	Credit Based Shaper
CDT	Control Data Traffic
CSMA/CD	Carrier-sense multiple access with collision detection
FIFO	First In, First Out
FQTSS	Forwarding and Queuing Enhancements for Time-Sensitive Streams
GCL	Gate Control List
gPTP	Precision Time Protocol
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
ISP	Internet Service Provider
LAN	Local Area Network
MAC	Medium Access Control
NTP	Network Time Protocol
PCP	Priority Code Point
PDM	Propagation Delay Measurement
NP	Nondeterministic Polynomial
OS	Operating System
QoS	Quality of Service
SRP	Stream Reservation Protocol
ST	Scheduled Traffic
TAS	Time-Aware Shaper
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
TSN	Time-Sensitive Networking
TTI	Transport of Time-synchronization Information
UDP	User Datagram Protocol
VLAN	Virtual LAN
WAN	Wide Area Network

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

Ethernet is a family of computer network technologies that are known and used worldwide. Ethernet connects our computers and devices we use daily to different geographical networks, whether they span from a Local Area (LAN) to a Wide Area (WAN). However, in doing so, its network service is mostly based around the use of Best-Effort traffic with limited options for Quality of Service (QoS)[1] [1]. When best-effort traffic is sent from one point to another, it is delivered as the name implies, with a best-effort. There are no guarantees that traffic will arrive on time for time-sensitive applications. QoS might be employed by network administrators or Internet Service Providers (ISP) to try and address these issues. However, QoS is limited in its use and is not suitable for systems where timely packet deliveries are required. Systems with these requirements are called real-time systems [2]. Real-time systems are required to perform their actions within specified time frames i.e., deadlines, to achieve deterministic behavior.

Several network protocols can be used for real-time communication. Many of these protocols, such as Controller Area Network (CAN) or FlexRay specify their own physical layer. Therefore, in order to take advantage of the determinism provided by these protocols, it is necessary to use their specific cables and devices. However, depending on the application, trying to integrate them with other technologies can be difficult. Instead of migrating systems to dedicated real-time protocols, it would be logical to try and provide Ethernet with these features since it is a cheap, well-known, and widely used protocol. Furthermore, the Ethernet standard provides high bandwidth which can be interesting to take advantage of in real-time applications. The use of high bandwidth is especially relevant, according to [3], the next generation of systems might increase the traffic in real-time networks by two orders of magnitude.

Due to QoS limitations in Ethernet, the IEEE Audio Video Bridging (AVB) task group was created in 2005 [4], [5]. The goal of the task group was to provide real-time behavior for audio-video streaming over Ethernet. Its scope was later broadened in 2012 and the task group was renamed to the Time-Sensitive Networking (TSN) task group. The TSN task group aims to develop technologies that suit other systems such as automation and automation systems. TSN is a promising real-time solution that provides flexibility of traffic. With TSN, generic traffic can flexibly share the same network as time-critical traffic by dividing traffic into different classes, flows, and priorities. It uses traffic shapers like the Credit-Based Shaper

[1] QoS is a mechanism that is used to prioritize different data traffic flows through a network [25].

(CBS) and the Time-Aware Shaper (TAS) to achieve satisfactory real-time communication. However, the shapers require the network to be fully synchronized since switches and devices need to work in unison for TSN to perform properly.

A TSN deployment could prove to be cost-effective and beneficial for companies. Researchers aim to extend current legacy networks by adopting TSN into them and providing them with real-time capabilities [6], [7]. However, such integrations present challenges in terms of synchronization and scheduling. As mentioned, TSN requires a network to be fully synchronized to achieve high performance in terms of delay and jitter [8]. Because of these performance challenges, we are investigating and analyzing the effects of partial synchronization in TSN. For example, networks where only the endpoints are synchronized or those that are left unsynchronized can be considered partially synchronized networks. The investigation is motivated in the sense that it could help researchers improve TSN and provide valuable information on how synchronization impacts a network. Viable solutions to the problems posed will also be investigated.

Since TSN is supposed to integrate with different legacy systems, it is interesting to analyze results from heterogeneous networks where operating systems and protocols have notable differences. This thesis is following up on Nguyen and Nasiri's work [9] on analyzing the performance of non-synchronized heterogeneous TSN networks. We are, in turn, investigating the effects of synchronized endpoints in a similar heterogeneous network through a series of experiments. To obtain consistent results the experiments are conducted with the same endpoints. The effects of a partially synchronized Ethernet network will be compared to a partially synchronized TSN network. In both cases, only the endpoints will be synchronized with each other.

2. Background

In the background section, an overview of the hurdles with Ethernet is given and the solutions switched Ethernet offers. Moreover, switched Ethernet has its own set of problems when used together with real-time constrained traffic which can be solved with AVB and/or TSN.

2.1. Ethernet

Inspired by the Aloha network, Bob Metcalfe invented Ethernet in the early 1970s [10, pp. 5-8]. ALOHAnet was used as a radio network to connect the different Hawaiian Islands while sharing the same communications channel. Sharing the channel means having some sort of system where information could be sent and received on the same channel by different nodes. ALOHAnet used acknowledgments for confirmation of successful communication, and if no acknowledgment was received the communication attempt was assumed to have collided with an attempt by another node. Collisions generated a random backoff time with subsequent retransmission.

Metcalfe expanded upon ALOHAnet by suggesting a communication protocol that detected collisions and preemptively avoided collisions by listening to the channel before starting a transmission [10, pp. 5-10]. He also further developed ALOHAnet's backoff timer for collision detection. The technology later became known as Ethernet and the protocol for channel access was named Carrier Sense Multiple Access with Collision Detect (CSMA/CD). The Ethernet standard became published in 1980 and with help of the Institute of Electrical and Electronics Engineers (IEEE), it was pushed to become a standard for LANs. The IEEE 802.3 Ethernet standard is the most popular network technology in use today, being a worldwide standard. Ethernet's most prominent features are the cost-effectiveness of implementation and the throughput it offers [11].

Wired Ethernet was half duplex in its infancy, meaning that communication could only occur in one direction at a given moment [10, pp. 28-29]. Speakers needed to listen to the shared channel before sending to avoid collisions, and even then, if two or more speakers decide to speak at the same time, collisions would occur. In terms of reliability and dependability, using such communication creates delays in message delivery, which in turn is unsuitable for time-critical applications.

2.2. Switched Ethernet

Before switches were introduced to Ethernet, hubs were used to connect several devices to a single communication channel. However, all incoming messages to the hub become repeated and *broadcasted* to all outgoing network ports on the hub [12]. This method of communication is inefficient since not all devices might be interested in listening to the speaking device at the given moment. Apart from excessive resource usage the method also introduces a greater potential for collisions. Switches, on the other hand, solve this problem

by keeping track of connected devices through Medium Access Control (MAC) addresses with the Address Resolution Protocol (ARP) and then placing the messages into buffers and queues. The simplest queuing method for a switch is First In, First Out (FIFO) where incoming messages simply are put in a queue to be sent out through the corresponding output port. A switch also negotiates with connected devices to operate in full-duplex, if possible, to eliminate collisions.

The use of traditional Ethernet switches solves the problem of collision domains within a LAN and allows for high throughput. However, they are still limited in their capabilities of providing predictable service to time-critical *real-time applications* [13]. Since Ethernet switches make use of simple queuing mechanisms without scheduling there are no guarantees in packet delivery. For example, in the case of several nodes simultaneously sending data through a switch to the same node, there is no way to tell how far back a message will be in the queue, which limits predictability.

2.3. Audio Video Bridging

AVB is the predecessor of TSN and was developed by the IEEE AVB Task Group. AVB aimed to provide switched Ethernet networks with real-time capabilities such as synchronization, reliability, and low latency [14], [15]. The task group introduced several standards such as:

- IEEE 802.1AS - Timing and Synchronization
- IEEE 802.1Qav - Forwarding and Queuing Enhancements (FQTSS)
- IEEE 802.1Qat - Stream Reservation Protocol (SRP)

Devices in a real-time network must share the same notion of time to be able to act in unison. 802.1AS is the standard that defines clock synchronization within AVB and TSN networks. It can provide synchronization errors of less than one microsecond. 802.1Qav defines forwarding and queueing rules to guarantee a timely fashion reception of messages. AVB also introduced the CBS which shapes traffic flows of different traffic classes and priorities while providing fairness, so lower priority traffic classes do not become starved. These traffic classes will be discussed later in section *2.4.2.1 Credit-Based Shaper*. 802.1Qat is a protocol that reserves resources throughout a network. The resources are reserved in the data flow path, also called the stream. In this fashion, devices can reserve bandwidth to meet necessary QoS requirements when needed. AVB still falters with fulfilling requirements needed by hard real-time systems since messages can be delayed as CBS does not ensure zero jitter, which makes it not fully deterministic. Moreover, AVB does not handle channel congestion well.

2.4. Time-Sensitive Networking

As mentioned in the introduction, by 2012 the AVB task group was renamed to the TSN task group. The rebranding was made to further develop real-time technologies suitable for industries such as automation and automotive industries [4], [5]. TSN builds upon AVB standards and is aiming to provide Ethernet with deterministic features able to be used in hard real-time systems [16]. The TSN task group introduced all-around improvements to the AVB standard. For example, they revised the clock synchronization in 802.1AS and aimed to support time-triggered traffic by improving the queuing mechanism of AVB. This was done in the form of TAS in the IEEE 802.Qbv standard. TSN also introduced a new traffic class of higher priority, called Scheduled Traffic (ST) as well as setting a maximum frame-size restriction on all traffic classes.

2.4.1. Clock Synchronization

To achieve deterministic behavior in a TSN network it needs to be synchronized. Devices throughout the network need to share the same notion of time to be able to fully cooperate [17], [6]. The clock synchronization that takes place in TSN is described in the IEEE 802.1AS standard. It is performed with Precision Time Protocol (gPTP) which can synchronize devices in the order of nanoseconds. Several mechanisms are used when synchronizing clocks, such as the Best Master Clock Algorithm (BMCA), Propagation Delay Measurement (PDM), and Transport of Time-synchronization Information (TTI).

The BMCA determines which candidate clock is most suitable to act as a grandmaster clock, i.e., the reference clock for the network [6], [18]. The grandmaster selection is done by comparing several properties of the candidate in *announce* messages which are broadcasted from time-aware systems. Time-aware systems receiving announce messages pick the system that is believed to have the best clock to be the grandmaster, they also compute what state the port is going to be. Ports are determined to be master, slave, passive, or disabled ports in a hierarchical fashion where ports connected to the grandmaster system become slaves. Master and slave ports are used for synchronization purposes and passive ports are used to avoid synchronization loops. See Figure 2-1 for an overview of BMCA and its selection of ports and grandmaster clock.

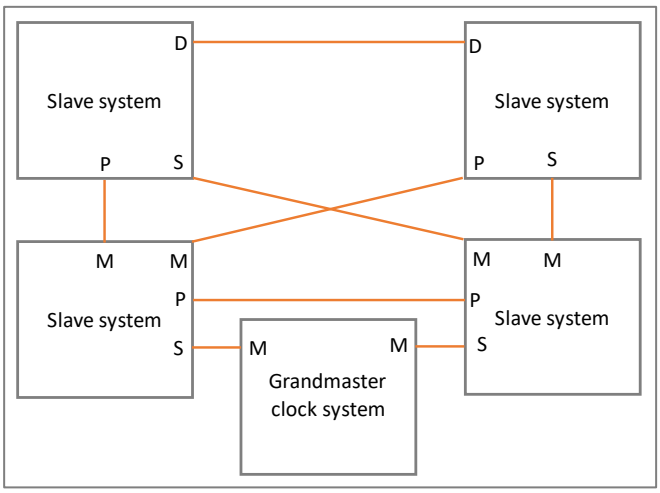

Figure 2-1: *Best Master Clock Algorithm*

The PDM mechanism is tasked with calculating the delay between systems in the time-aware domain, see Figure 2-2. This is done by timestamping messages from the different systems, comparing them, and calculating the delay in the requesting system [6]. The delay is calculated with the formula: $Delay = \frac{(T4-T1)-(T3-T2)}{2}$. The TTI is a process that takes place once a grandmaster clock is decided and PDM executed. During the TTI process systems send their local time through their master ports where the receiver adds the measured delay and updates their local time. Thus, the network becomes fully synchronized.

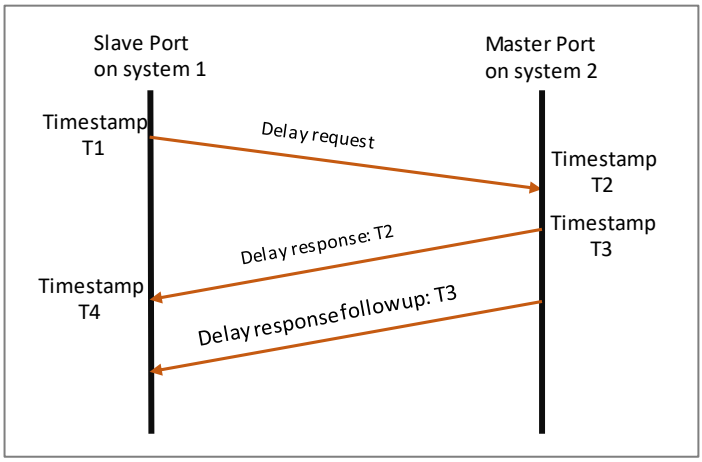

Figure 2-2: *Propagation Delay Measurement operation*

2.4.2. Scheduling

There are two main schedulers, or *shapers*, within the TSN framework, CBS, and TAS. These shapers impose rules for queuing and forwarding of messages, as well as scheduling traffic on output ports [16]. Both shapers are explained in more detail in the following sections.

2.4.2.1. Credit-Based Shaper

CBS is the algorithm used to decide which AVB queue is permitted to send. AVB frames are typically of class A or B where A is the frame with the highest priority [19], [20]. Outside the CBS there can also be Best-Effort traffic and ST. CBS works similar to a "leaky bucket" where frames of the highest priority are sent first. However, if there is an uninterrupted flow of frames CBS makes sure that frames of lower priority do not become starved. Figure 2-3 shows an example of the CBS operation. In the figure there are two defined slopes, idleSlope and sendSlope, there is also a zero line. Messages can only be transmitted when their credit value is above zero. In the figure, three messages are queued for sending. Since m_1 in this example is of higher priority it will start sending first and subsequently consume credit as shown with the sendSlope. When m_1 is done sending it has a credit value below zero as shown in line 1, this means the next message queued for class A cannot be sent. M_2 on the other hand, which has gained a positive credit will start sending its message and go into its own sendSlope. Message m_3 will be sent when m_2 is done since class A's credit value is above zero. If no frames are queued and credit is positive, the credit is set to zero, as shown after m_3 is sent. If the credit is negative with no frames queued, credits will be gained with the rate of idleSlope until zero, as shown in line 2. It is performance beneficial for TSN to combine CBS with TAS. TAS is explained in the upcoming section.

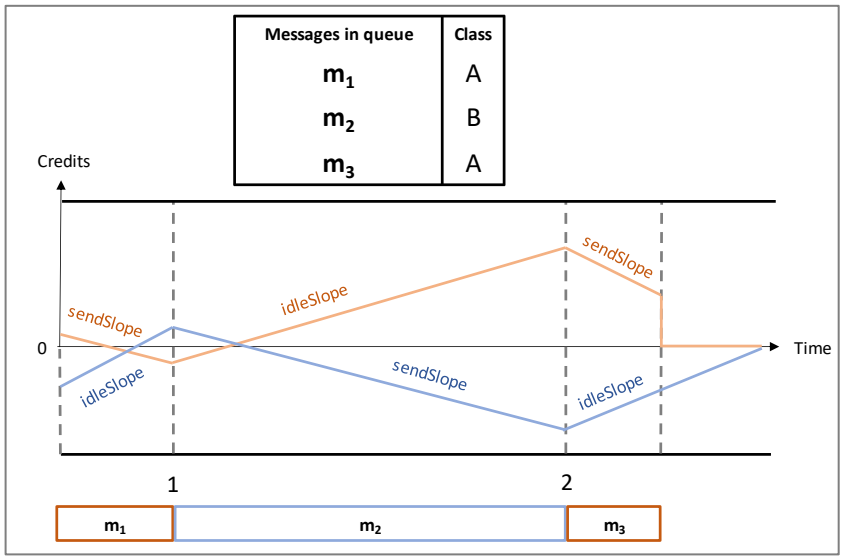

Figure 2-3: *Credit Based Shaper operation*

2.4.2.2. Time-Aware Shaper

TAS was defined in the IEEE 802.1Qbv standard and enables time-triggered communication with the use of the network clock, in combination with time-aware gates [19], [20]. There can exist up to eight time-aware gates which reside on every egress port of a TSN switch. In Figure 2-4 an overview of the TAS is shown. In this figure, four FIFO queues are considered containing Control Data Traffic (CDT), AVB queues A and B as well as best-effort Traffic. Messages are queued until their gates open at a scheduled time. A Gate Control List (GCL) decides which gates are open and closed in each time slot. Here, such instances are marked with 1 in the GCL table for open gates and 0 for closed gates. When the GCL reaches the bottom of its list it repeats.

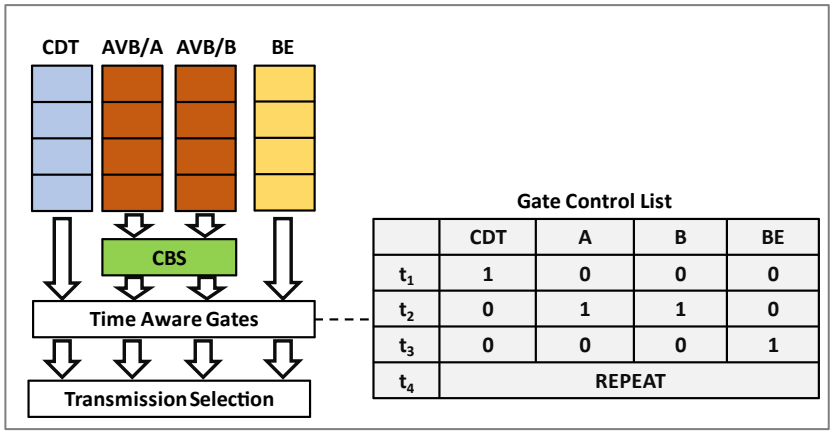

Figure 2-4: *Time-Aware Shaper overview*

Traffic scheduled in slots like in a Time Division Multiple Access (TDMA) scheme provides some advantages. It offers great support for time-triggered communication since time-scheduled messages are known and can be adjusted with minimal jitter and latency. However, the effects are worse for event-triggered traffic where the origin time of generated messages is unknown.

3. Related work

There are several studies focused on studying and evaluating TSN shapers and different synchronization methods as well as improvements to these. This thesis is not focused on studying the synchronization mechanism or improving it, but rather on the consequences a network or system may experience when synchronization is not present in some parts of the TSN network. Since we explore the possibilities of integrating TSN together with legacy protocols we must also consider scenarios where some systems cannot synchronize with TSN. For example, nodes might only be able to synchronize with themselves if they are not fully compatible with TSN.

Chouksey et al. [21] conducted an experimental study on the coexistence of TSN and non-TSN devices which is similar to what we aim to do. In this thesis we are using non-TSN endpoints connected to a TSN switch to measure the effects of non-synchronized. In their work, they used a TSN gateway to convert non-TSN devices packets to TSN by adding a VLAN TAG. In their work, they observed difficulties in topology discovery and time synchronization of devices. However, these problems were solved with modifications to the configuration, which achieved promising results.

Barzegaran et al. [7] present work similar to this thesis. While we are focusing on analyzing and evaluating the effects of end systems that are synchronized with each other they relaxed the synchronization requirements on end systems. In our work, end systems are assumed to not be able to synchronize with the TSN infrastructure, but with each other, through synchronization protocols that are not compatible with TSN ones. In their work, they mentioned that it is often not realistic for systems like microcontrollers and legacy systems to have TSN capabilities and be synchronized with the TSN network. Barzegaran et al. finally concluded a solution to provide real-time guarantees for time-critical traffic.

Other works that are essential to this study are the evaluations of shapers, scheduling, and synchronization. A broad understanding of the functions of TSN is needed to evaluate the effects of the lack of synchronization. Kim et. al [22] show in their work how unscheduled high-priority traffic in TSN may induce negative consequences such as high delay and jitter in the scheduled traffic. These effects were avoided by introducing separate protection to the standard TAS schedule. Le et al. [17] present a simulation model for TSN and utilize synchronization and TAS to obtain high accuracy scheduling. Mateu et. al [6] show how clock synchronization in TSN plays an essential role when integrating a legacy EtherCAT network into a TSN network. Reports including heterogeneous network approaches are interesting to compare to this work. Especially since the end systems in this thesis will be separated, synchronization-wise, from the TSN network. In [6] it was concluded that their proposed synchronization mechanism obtained at least three times higher precision among nodes compared to using no synchronization mechanism.

4. Problem formulation

Embedded systems in automotive and industrial applications are increasingly dependent on real-time capabilities and therefore it is important to meet this need. TSN aims to offer hard and soft real-time capabilities to Ethernet by having flexible management of traffic which allows for integration with other protocols in legacy systems. Using Ethernet technology, network deployment with TSN would be cost-effective [20].

TSN requires the network to be fully synchronized to achieve high performance in terms of delay and jitter [8]. However, it would be important to analyze network behavior when parts of the network are not synchronized. For example, there are cases where legacy nodes do not support clock synchronization and they cannot be synchronized with the rest of the network. There are also cases where endpoints such as microcontrollers are desired in the network because of their price and simplicity. However, these units may not support the clock synchronization mechanism provided by TSN but may synchronize with each other with the use of their own synchronization mechanisms. This thesis is therefore motivated by analyzing network behavior when diverting from best practice synchronization recommendations.

We would like to investigate and see the effects in such partially synchronized networks where only the endpoints are synchronized. It is implied that some performance issues will exist by not fully synchronizing the network since devices in the TSN network need to perform in unison to guarantee bounded latency and avoid jitter. The research questions we intend to answer are:

- With regard to clock drift and jitter, what are the effects of only synchronizing endpoints through their legacy synchronization mechanisms in a heterogeneous TSN network?
    - If the effects result in clock drift between devices, in what ways can we manage this and is it possible without interfering with the TSN switch?

To answer these questions, we are limiting the scale of the experiment by reducing complexity and using a small network consisting of two nodes and a TSN switch. This network is smaller in size and not comparable to networks used in real applications. However, with thorough research methods, we believe the results can be indicative of the function in real networks, thus they can be extrapolated and applied in real scenarios.

5. Method

The functions and intricacies of TSN are complex and therefore we are conducting a literature review about TSN, its background, and mechanisms. The literature review is also necessary to study the related works in the field to obtain an understanding of what is done today in terms of research. Additionally, approaching the problem described in the problem formulation can be done pragmatically, via experiments. According to Säfsten and Gustavsson [23], experiments are an effective way to investigate how systems behave under different scenarios and circumstances. We are designing experiments as an improvement on earlier experiments done by students at MDH by adding partial synchronization into a TSN network and analyzing if the synchronization is beneficial or not. The experiments help us gather relevant information and quantitative data about the network's behavior during different synchronization scenarios which we are analyzing and evaluating.

The methodology is structured according to the system development research process as described by Nunamaker, Chen, and Purdin [24]. This is an iterative process with the following steps:

(1) Construct a conceptual framework.
(2) Develop a system architecture.
(3) Analyze and design the system.
(4) Build the prototype.
(5) Observe and evaluate the system, then iterate to (1) if needed.

All steps are not covered in this thesis but serve as a fundamental base for the research process that is carried out in this thesis:

(1) Gathering information via literature reviews.
(2) Design an experiment with a partially synchronized heterogeneous TSN network.
(3) Building and setting up the experiment.
(4) Investigating, analyzing, and evaluating the effects of the network not being fully synchronized.
(5) If performance issues are found, propose a solution to the issues and iterate to (1) or (2).

6. Experiment construction and implementation

To answer the questions posed in the problem formulation several experiments were designed and carried out. The experiments were constructed to investigate the effects of having a partially synchronized TSN network. In the network used, the endpoints synchronized with each other and a TSN switch acted as a "black box", meaning that the endpoints had no knowledge of the TSN switch and did not synchronize with it. The design and implementation are described in the following sections.

6.1. Network structure and hardware

We constructed a small network consisting of two microprocessors; Raspberry Pi 3 Model B[2] running Raspberry Pi OS[3] and a SoC-E (System on Chip engineering) Multiport TSN kit switch[4] was constructed, see Figure 6-1. The Raspberry Pi's operate on 100 Base Ethernet while the TSN switch operates on 1000 Base Ethernet. The Raspberry Pi's were only configured to synchronize their software clocks with each other, this was done with Network Time Protocol (NTP). However, any clock synchronization protocol can be used between the endpoints since we are emulating scenarios where the TSN switch is unable to synchronize with the endpoints. The TAS windows in the switch were configured according to the receiver node's maximum allowed throughput. Information on how to configure the switch can be found in Appendix A. Switch configuration.

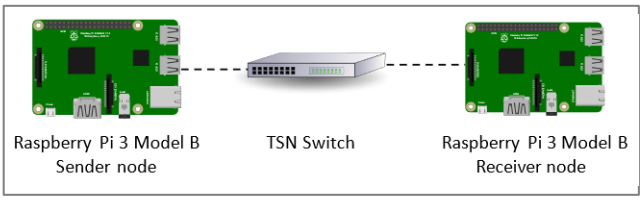

Figure 6-1: *Network topology*

[2] Raspberry Pi 3 Model B. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-3-model-b/

[3] Raspberry Pi OS. [Online]. Available: https://www.raspberrypi.com/documentation/computers/os.html

[4] MTSN Kit: a Comprehensive Multiport TSN Setup. [Online]. Available: https://soc-e.com/mtsn-kit-a-comprehensive-multiport-tsn-setup/

6.2. Experiments

Descriptions of the different experiment scenarios can be found below. First, a baseline experiment was constructed to understand how the network performs without TSN and any kind of clock synchronization between the endpoints. Further experiments with TSN and clock synchronization were constructed as well as suggested solutions to emerging problems.

To enable sending of periodic traffic that could be processed by the TSN switch the Python module Scapy[5] was used in the running scripts on the sender and receiver node to forge Ethernet frames. The frames must have Virtual LAN (VLAN) information, also known as 802.1Q to be priority-handled by TSN. TSN makes use of the Priority Code Point (PCP) field inside the 802.1Q-header (see Figure 6-2) which consists of three bits, meaning the PCP can assume eight different values. These values in turn correspond to the eight time-aware gates within TAS. See Appendix B to Appendix E for the Python scripts that were used during different experiment scenarios. Packets sent from the sender node as well as packets received in the receiver node were logged in the network protocol analyzer Wireshark[6].

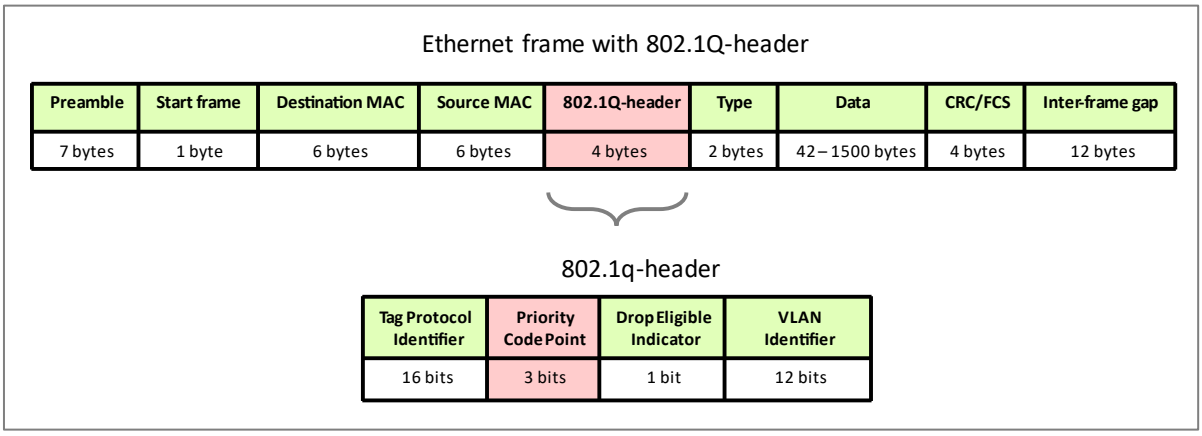

Figure 6-2: *Priority Code Point inside 802.1Q-header*

6.2.1. Initial scenarios

Three different experiment scenarios were set up. The first experiment was designed to gather data on how the network behaves between two endpoints, without any influence of a TSN switch or clock synchronization, see Figure 6-3. The second experiment was constructed as a

[5] Scapy is a tool that allows the user to manipulate network packets. It can be used to forge and decode packets of various protocols. [Online]. Available: https://scapy.readthedocs.io/en/latest/index.html

[6] Wireshark. [Online]. Available: https://www.wireshark.org/

comparison to the first with the addition of clock synchronization by NTP between the endpoints, without the use of a TSN switch, see Figure 6-4. Finally, an experiment with synchronized endpoints and an unsynchronized TSN switch was constructed, see Figure 6-5. The third experiment was made to observe the behavior when two already synchronized endpoints are subjected to an intermediary TSN switch that has its own notion of time.

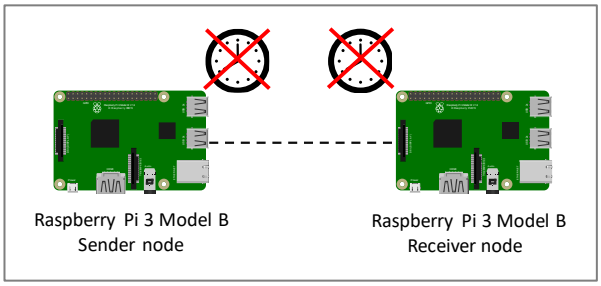

Figure 6-3: *Scenario 1 – unsynchronized endpoints.*

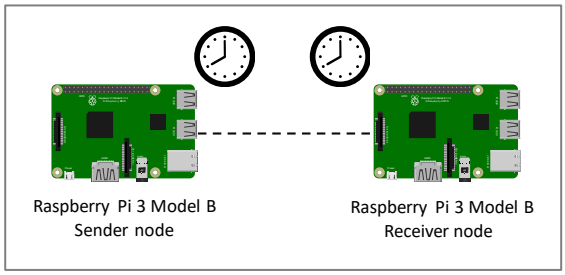

Figure 6-4: *Scenario 2 – synchronized endpoints*

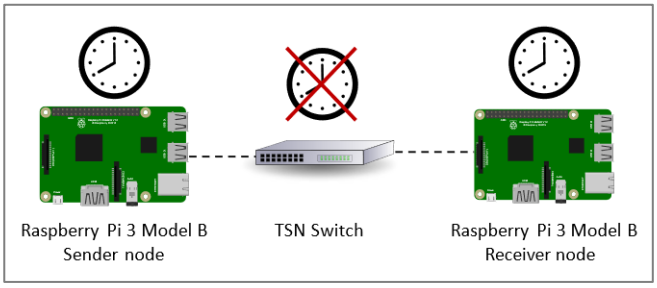

Figure 6-5: *Scenario 3 - synchronized endpoints, unsynchronized TSN switch*

In the third scenario with the introduction of the TSN switch, we had to decide on a window size for the prioritized packets, i.e., the scheduled packets. In our case, the packets forged by Scapy should only be allowed during their allotted TAS windows, and two packets should not be allowed to traverse together in one TAS window. Since the Raspberry Pis use Base 100

Ethernet and the Scapy packets were 60 bytes, we could calculate the minimum window size by dividing the packet size with the throughput accordingly:

$$\frac{\text{Packet size}}{\text{Throughput}} = \frac{60 \text{ bytes}}{100 \text{ Mbit/s}} = \frac{480 \text{ bits}}{100 \text{ Mbit/s}} \Leftrightarrow Minimum\ window = 4800\ ns$$

However, when testing these window sizes, packets were not traversing their TSN windows. When extending the window sizes, it was discovered that packets flowed freely at a window size of 6725 ns. The windows were then extended by 10% to account for any issues jitter may cause giving $6725 \cdot 1.1 \approx 7400$ ns.

In the three experiments, 1000 packets were sent from the sender- to the receiver node in one-second intervals and logged in both endpoints with Wireshark.

6.2.2. Extreme scenarios

In the scenario with synchronized endpoints and an unsynchronized TSN switch, the receiver node experiences reception time drift in the messages since it does not share the same notion of time as the TSN switch (see the results section 7.1.3). However, the reception drift is small, which means that within 1000 seconds and 1000 packets no errors were noticed in the network.

We hypothesized that if there is a negative reception time drift in the receiver node, eventually packets will miss their windows in the TSN switch. See Figure 6-6 for an illustration of this scenario. The opposite scenario was also hypothesized to be plausible, i.e., when the receiver node experiences a positive reception time drift. In this scenario, packets will be sent in every TSN switch window. However, packets will instead be queued in the switch since it only allows for one packet to be sent through said window, see Figure 6-7. To investigate these extreme circumstances new experiments were conducted.

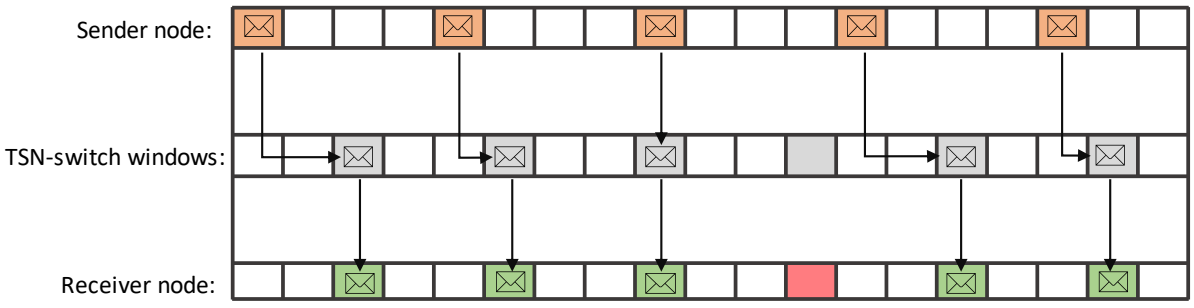

Figure 6-6: *Negative reception time drift in the receiver, leading to packets missing their schedule*

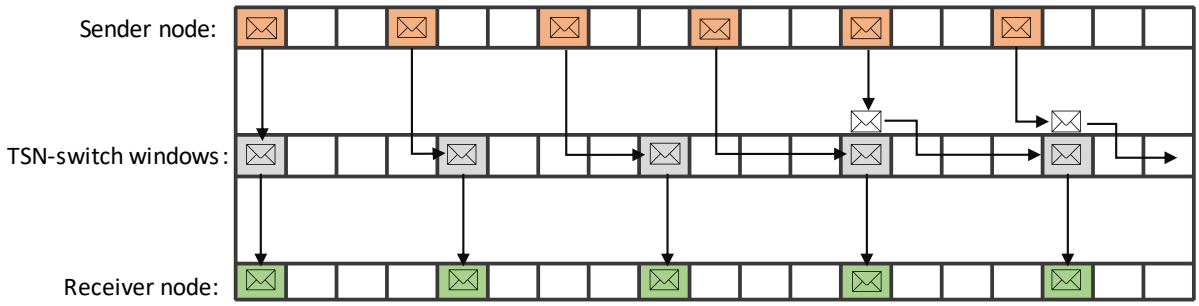

Figure 6-7: *Positive reception time drift in the receiver, leading to queued packets*

The experiments for the extreme scenarios were constructed in the same manner as the initial scenarios in section 6.2.1. However, in some cases, we modified the sending intervals and/or the size of the TAS cycle time in the switch to artificially construct a larger negative reception time drift in the receiver. Specifically, the sending interval of packets was kept at one second, but the switch cycle was decreased to 990 milliseconds. According to our hypothesis, packets would miss their window every ~100 packets since the TSN switch will have a synthetic clock drift of negative one second every 100 seconds compared to the endpoints.

To test the opposite scenario where we introduce an artificial positive reception time drift in the receiver node the sending interval was set to 500 milliseconds in the sender node while the TSN switch cycle was kept at one second. The packets were also reconstructed to carry data, which were used for labeling, to keep track of eventual packets being lost. Lost packets were thought possible to occur when queue buffers become overflowed in the switch. See Appendix E for the script which includes a payload in the packets.

6.2.3. Implementing solutions

Legacy networks can exhibit different problematic characteristics. In homogeneous networks, all endpoints share the same drift, but this is not true for heterogeneous networks with different sets of endpoints. Therefore, we considered solutions that could cope with drift varying between different subnetworks, as well as drift varying the same way throughout a homogeneous network. Other characteristics to account for were whether the drift is positive or negative.

6.2.3.1. Adjusting cycle time in the switch

Upon reviewing the results from extreme scenarios where we induced a synthetic clock drift, we suggested measuring the perceived drift in the receiver node and then updating the TSN switch accordingly. For a measured negative drift of ten milliseconds per second in the receiver, we hypothesized that the instances where a *critical error* takes place would be halved if the switch cycle were increased by five milliseconds. A *critical error*, in this case,

would be when a packet misses a window in the switch. By adjusting the cycle time in the switch according to the endpoints notion of time we should improve the performance of the network.

To improve the solution, we tried to accurately measure the drift in the receiver node by having both the endpoints send packets every second as well as having the switch cycle time set to one second. This is the same type of experiment as the initial experiment in Figure 6-5. After 1000 seconds (1000 packets) we measured the reception drift in the receiver node. By dividing the measured drift by 1000 we received the clock drift per second and adjusted the cycle time in the switch, see Figure 6-8.

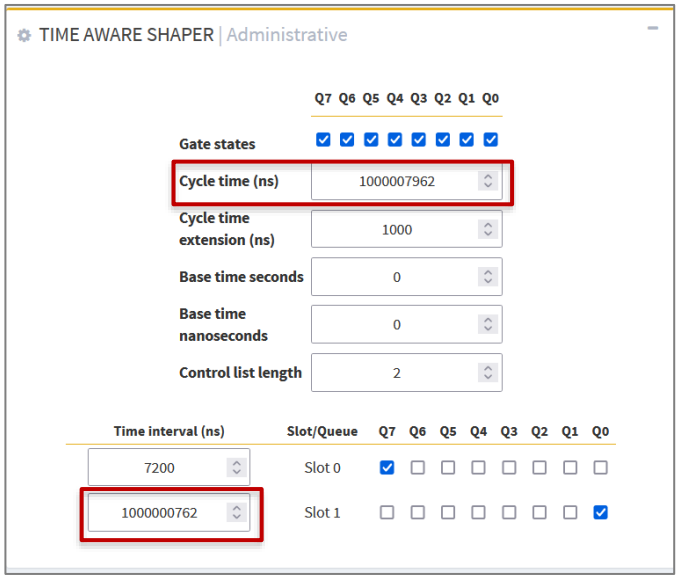

Figure 6-8: *Adjusting the cycle time in the switch*

6.2.3.2. Dynamic adjustments of sender interval

The second proposed solution consisted of adjusting the sender interval when considering the perceived reception drift in the receiver node. This is achieved by letting the sender node be aware of the current drift perceived in the receiver node by having it periodically send an update message to the sender node. We had the receiver node sample several incoming packets at a time and calculate the slope value over a set of points by using the trendline formula:

$$y = ax + \beta$$

Where the slope's formula ***a*** is given by:

$$a = \frac{n\sum(xy) - \sum x \sum y}{n\sum x^2 - (\sum x)^2}$$

Where ***n*** represents the number of points (packets), ***x*** represents the expected time of packet arrival and ***y*** is the time the packet was recorded on the incoming interface.

For example, the following graph (Figure 6-9), consists of two slopes, a_1 = 4 and a_2 = -0.5, the expected packet schedule is set to one second. With the slope formula we get:

$$a = \frac{3\sum(xy) - \sum x \sum y}{3\sum x^2 - (\sum x)^2} = \frac{3 \cdot 23 - 6 \cdot 10}{3 \cdot 14 - 36} = \frac{9}{6} = 1.5$$

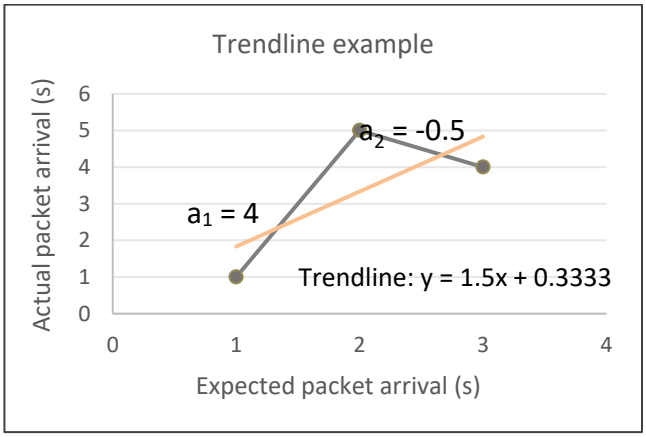

Figure 6-9: *Trendline over a set of points*

In this implementation, the receiver node is calculating packet reception drift between the TSN-switch and the receiver node. For example, if the packet reception drift between switch and receiver is detected to be positive 500 milliseconds per second, then it updates the sender node of a change in drift of positive 500 milliseconds per second. In future iterations the measured drift may change due to variations in the clocks of the end stations or the TSN network. For example, if the measured drift become positive 400 milliseconds per second, the receiver node will update the sender node with a change in drift of negative 100 milliseconds per second to adjust the transmission period to the new drift and this way prevent the aforementioned problems. If the mechanism was implemented in the TSN switch instead, the receiver might notice a slight drift in reception at the beginning which would disappear after the first few iterations of the mechanism. This kind of solution is outside of the scope for this thesis. The code for the dynamic solution can be found in Appendix C and Appendix D.

7. Results

This section shows and describes the results from the different experiment scenarios.

7.1. Initial scenarios

The results from the initial scenarios encompass the investigating phase of our network's behavior in three different states. As described in section 6.2.1 these scenarios include the following:

1. Unsynchronized endpoints without intermediary TSN switch
2. Synchronized endpoints without intermediary TSN switch
3. Synchronized endpoints with an intermediary TSN switch

7.1.1. Unsynchronized endpoints without TSN switch

In this scenario four graphs are presented, both sender graphs and both receiver graphs are the same, but one is focused on showing jitter while the other is focused on the drift.

In Figure 7-1, we can see the jitter of the unsynchronized sender node and in Figure 7-2 the slope value is represented as a trendline. The slope value is the average drift the receiver node experience per second and packet since the packets are sent in one-second intervals. In these graphs, the sender experiences a jitter of 43.5 milliseconds and a drift per second of 40 nanoseconds. However, since the packets are scheduled, sent, and sampled by the same system clock the drift is zero.

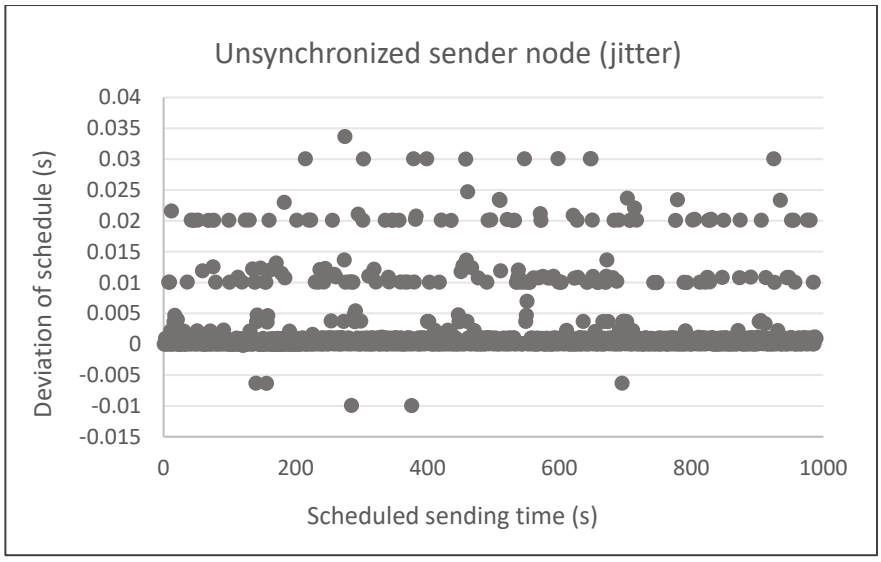

Figure 7-1: *Unsynchronized sender node without TSN (jitter).*

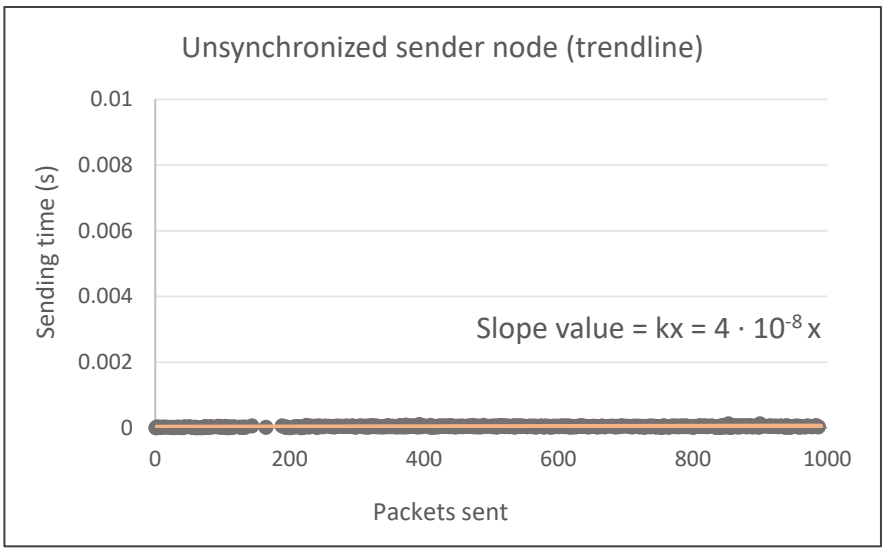

Figure 7-2: *Unsynchronized sender node without TSN (trendline).*

As in the previous figures, we can see the jitter of the unsynchronized receiver node in Figure 7-3 and the slope value in Figure 7-4. In these graphs, the receiver experiences a jitter of 43.6 milliseconds. The slope value, or drift, in the receiver node, is calculated to be 500 nanoseconds per second, which means that over the duration of the experiment the receiver node has experienced a packet reception drift of 500 microseconds.

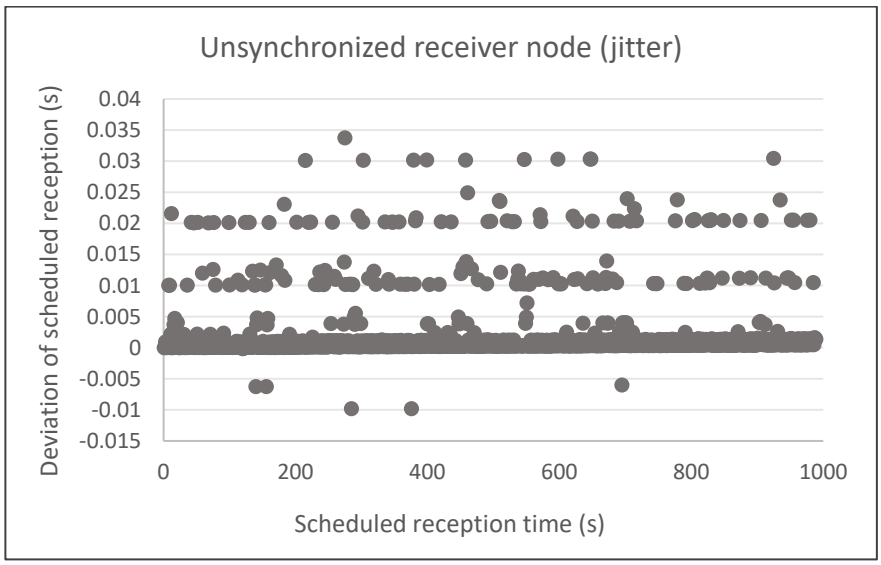

Figure 7-3: *Unsynchronized receiver node without TSN (jitter).*

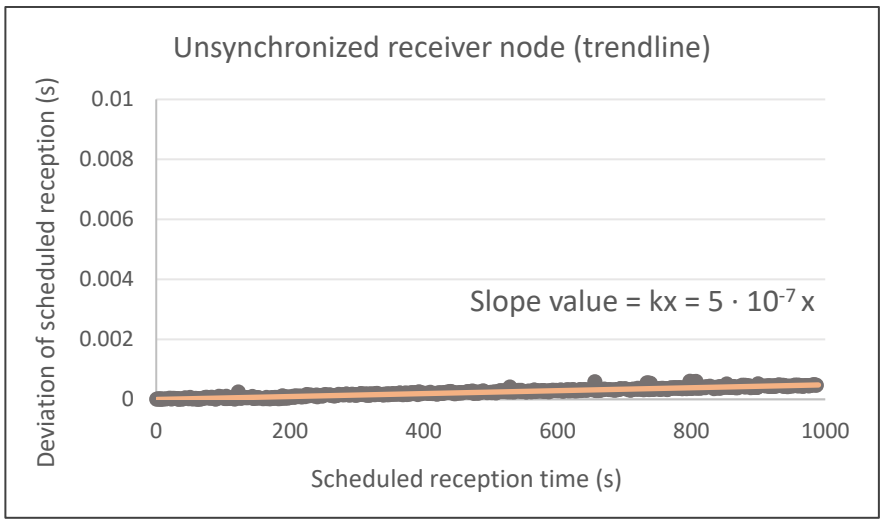

Figure 7-4: *Unsynchronized receiver node without TSN (trendline).*

7.1.2. Synchronized endpoints without TSN switch

In Figure 7-5 and Figure 7-6 we can see the jitter and drift of the receiver node. The graphs for the sender node are not included since the sender node acts as the NTP-server it is not affected by new experiments and will experience the same amount of jitter as well as zero drift. For comparison, the jitter was still recorded in the sender during this experiment and was measured to be 60 milliseconds, the jitter for the receiver node in Figure 7-5 was measured to be 59.5 milliseconds. The drift in the receiver node after synchronization between the endpoints was measured to be -60 nanoseconds per second.

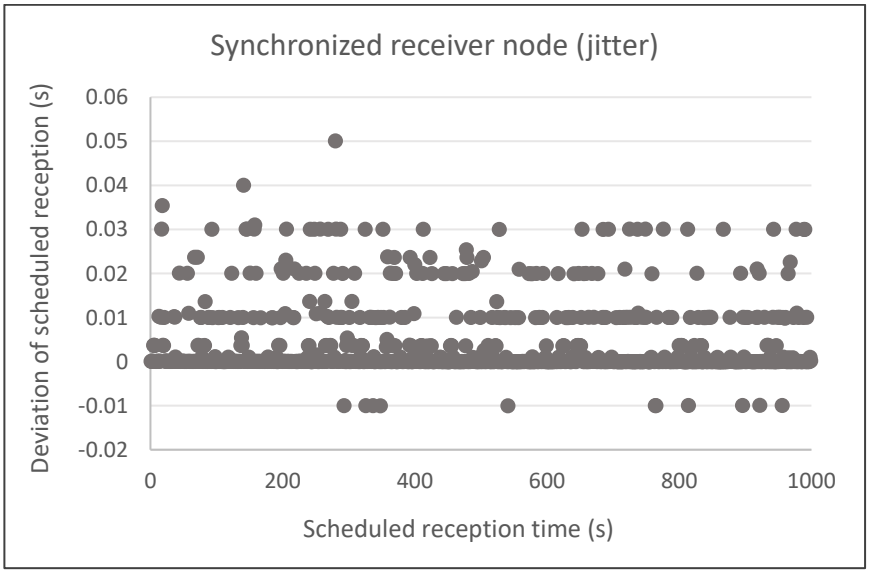

Figure 7-5: *Synchronized receiver node without TSN (jitter).*

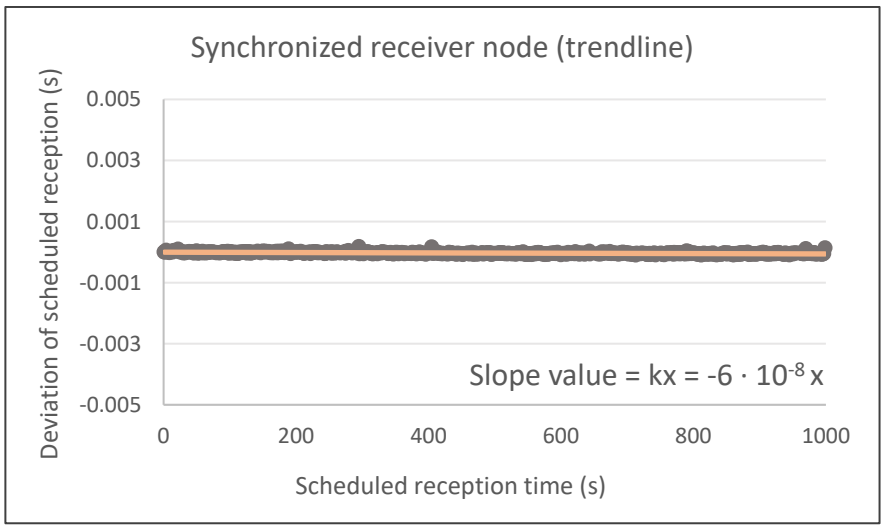

Figure 7-6: *Synchronized receiver node without TSN (trendline)*

7.1.3. Synchronized endpoints with intermediary TSN switch

In Figure 7-7 we can see that with the introduction of a TSN switch in the network the jitter is heavily reduced, it is measured to be ~180 microseconds. However, the drift has increased and is now calculated to be -8 microseconds per second. The experienced packet reception drift in the receiver node over the duration of the experiment is -8 milliseconds.

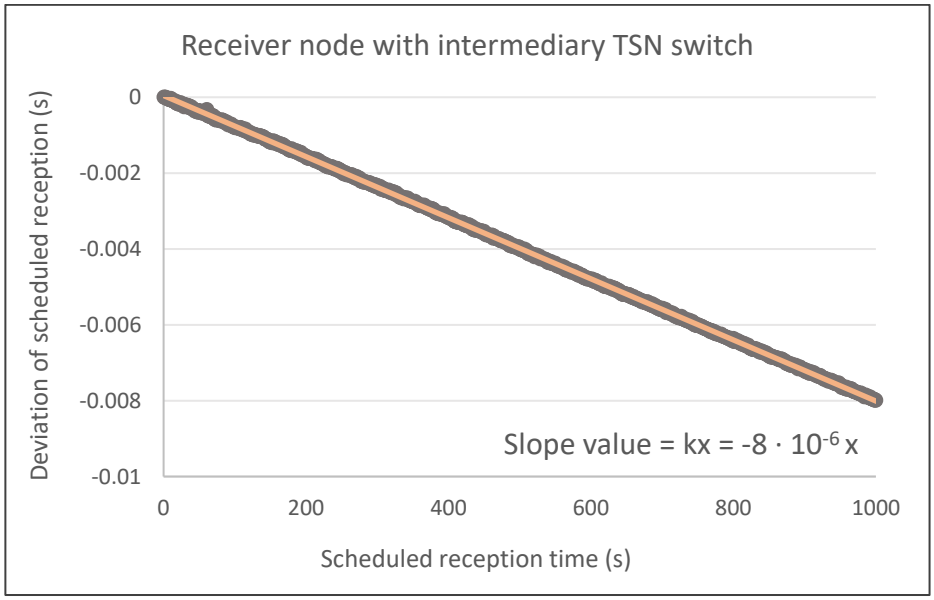

Figure 7-7: *Synchronized receiver node with intermediary TSN switch*

7.1.4. Summary

An overview of jitter and drift in the different experiments is shown in Table 7-1. Jitter in the receiver node is directly correlated to the sender node's jitter in the topologies without an intermediary switch. In the last experiment, jitter in the receiver instead depended on the TSN switch, which sends its packets in a fixed schedule. The jitter is lowest in the receiver when TSN is introduced while having the highest reception drift. The reception drift was lowest in the synchronized network without a TSN switch.

Node	Jitter (ms)	Drift/s (µs)
Sender unsynchronized	43.59	N/A
Receiver unsynchronized	43.62	0.50
Sender synchronized	60.05	N/A
Receiver synchronized	59.56	-0.06
Sender with TSN switch	89.93	N/A
Receiver with TSN switch	0.17	-8.00

Table 7-1: *Jitter and drift in initial scenarios*

7.2. Extreme scenarios

As described in section 6.2.2, the extreme scenarios aim to investigate the network behavior when introducing a synthetic clock drift. The reason for applying a synthetic drift is to investigate the long-term effects of clock drift but applied in a shorter period. In our case, we either changed the cycle time in the TSN switch or the sender interval in the sender node. The hypothesis is explained in further detail in section 6.2.2.

7.2.1. Synthetic negative clock drift

Figure 7-8 shows the results from the first extreme scenario where the hypothesis was that packets would eventually miss their TSN window. In this graph, the cycle time in the TSN switch was set to 990 milliseconds and the sender/receiver schedule was set to one second. The receiver node experiences heavy clock drift in its packet reception schedule. The jumps in the graph show that packets are missing their scheduled delivery times and are instead received in their next window 990 milliseconds later. Since there are ten jumps it is implied that packets have missed their scheduled window ten times.

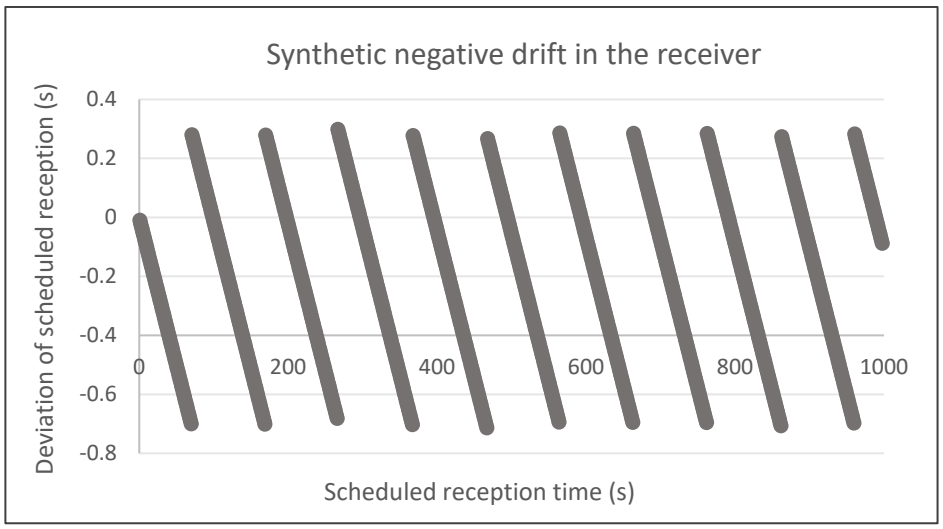

Figure 7-8: *Synthetic negative clock drift in the receiver*

7.2.2. Synthetic positive clock drift

In Figure 7-9 the result from the other extreme scenario is shown. Note that the graph does not display the heavy clock drift experienced in the sender node. A positive clock drift of 500 milliseconds per second exists, but the graph is represented in this way to clearly show that packets were lost. The x-axis shows the expected packet number, and the y-axis shows the deviation of the received packet number.

To force these errors, the TSN switch's cycle time was set to one second and the sender's sending interval was set to 500 milliseconds. After about 450 packets were received, packets started to disappear in irregular patterns which are illustrated in the graph which shows the accumulated lost packets on its y-axis. Packets being lost are further illustrated in Figure 7-10 where packets received are compared to the respective payload they carried, meaning the packet containing payload 454 was lost.

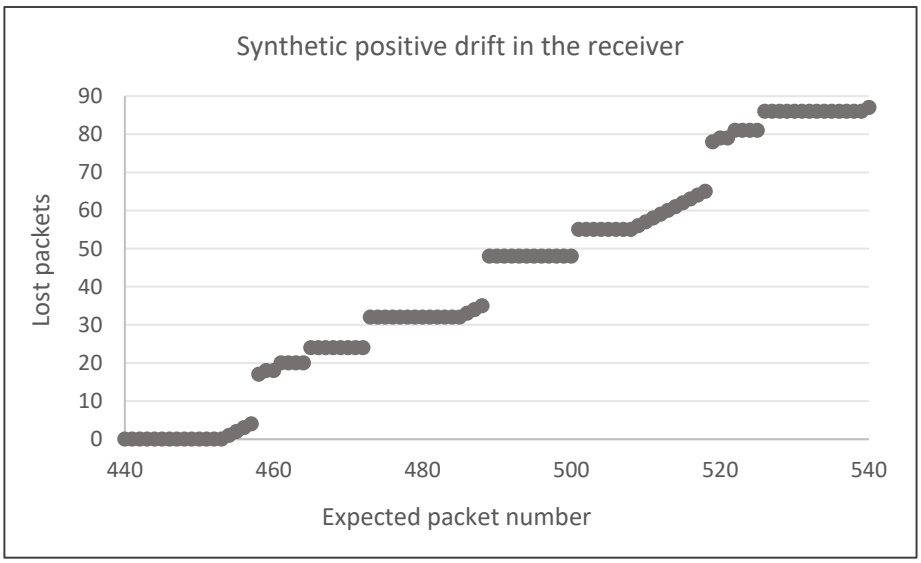

Figure 7-9: *Synthetic positive clock drift in the receiver*

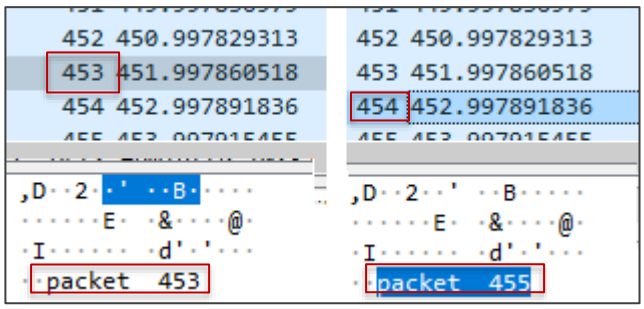

Figure 7-10: *Packets lost in Wireshark*

7.3. Solutions

This section covers the results from the implemented solutions. The solutions are derived from the analysis of the previous experiments.

7.3.1. Static solution

In Figure 7-11 the sender interval is set to one second and the interval in the switch is changed from 990 milliseconds to 995 milliseconds. The change is done to symbolize an improvement by statically trying to match the clock drift measured in the receiver by changing the cycle time in the TSN TAS schedule. In the graph, we can see that compared to Figure 7-8 where ten jumps in the graph indicated ten packets missing their scheduled window, there are now five jumps in this graph after changing the switch cycle time.

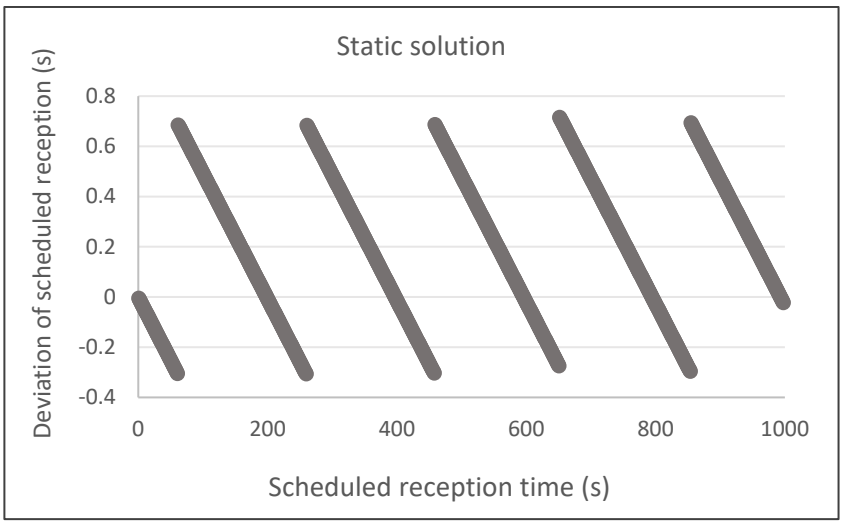

Figure 7-11: *Static solution*

Figure 7-12 shows the result from fully applying the precise static solution. After trying to precisely measure and estimate the real clock drift in the receiver compared to the TSN switch, the switch cycle was set to 1 000 007 962 nanoseconds, or 1.000007962 seconds. In the graph, all packets are arriving in their scheduled window and with almost zero drift. The slope value indicates that there is a drift of two nanoseconds per second which means that over the duration of the experiment (10 000 seconds) the receiver drifted 20 microseconds. This result means that it would take 500 million seconds (approximately 16 years) for the receiver to drift one second.

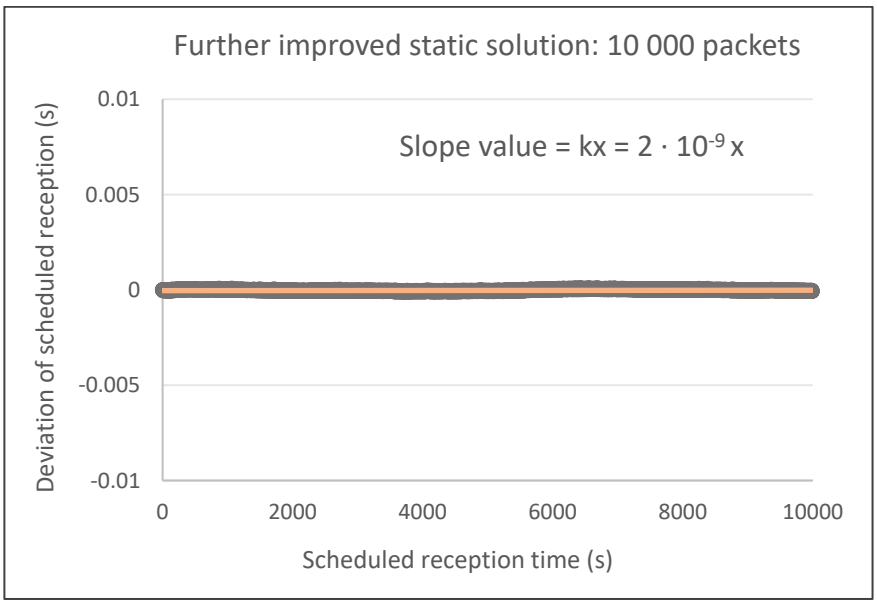

Figure 7-12: *Receiver node, static solution*

7.4. Dynamic solution to extreme scenarios

Figure 7-13 shows the result from the implementation of the dynamic solution on an experiment where the switch cycle was set to 990 milliseconds and the scheduled interval in the endpoints was set to one second. The experiment builds as a solution on the example of synthetic negative clock drift shown in section 7.2.1. In this experiment, the dynamic script was executed during runtime after approximately four minutes to show the effect of running the script. The graph shows that the number of critical errors, i.e., where packets miss their windows in the TSN switch are reduced from one per every 100 packets to zero over 70 000 packets.

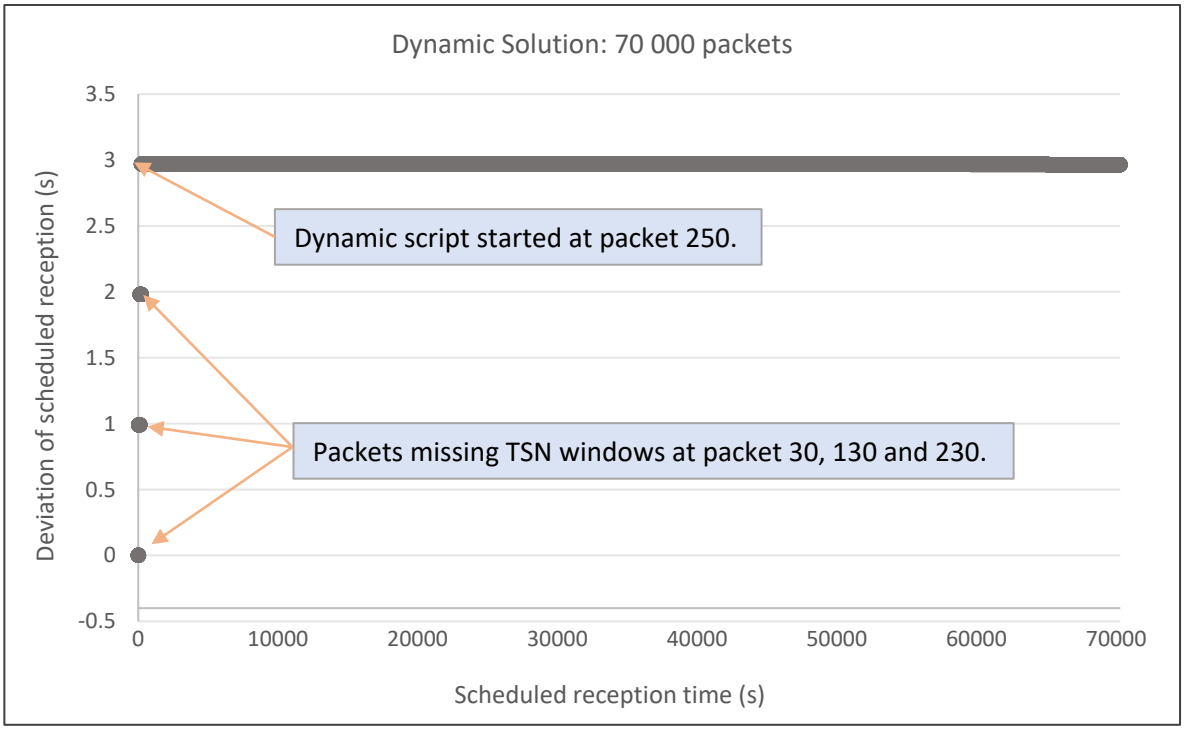

Figure 7-13: *Dynamic solution with average drift accounted for*

8. Discussion

With this work we aimed to analyze the effects of introducing an unsynchronized TSN Switch in a network with already synchronized endpoints, making it a partially synchronized TSN network. Moreover, since TSN seems to require full synchronization to function properly, we wanted to see if we could solve potential issues when the network is partially synchronized. To investigate this, we decided to design small-scale experiments in a closed network environment. We have been measuring the arrival time of packets on the input interfaces. Our measuring tool, Wireshark, allows us to make precise measurements in the order of nanoseconds. Since we also used a closed environment, we benefitted from having zero traffic originating from outside the network, as well as keeping other network traffic to a minimum. As scheduled traffic is isolated in its respective queue, we have no reason to believe that the scheduled traffic used in the experiments would be affected by other traffic in the network.

For the experiments, we first laid a baseline of initial experiments to gather data on the network performance without TSN, both unsynchronized and synchronized. When comparing the results, we noticed that synchronization had no discernible impact on jitter in the network, however, the drift was reduced from 500 nanoseconds per second to -60 nanoseconds per second. The drift is almost zero in the receiver and can be considered a statistical error due to the large amount of jitter in the packet reception. Depending on how well the sender node performs during runtime, the trendline can become skewed one way or the other. However, since we are seeing a substantial improvement in drift, we can assume that the endpoints are properly synchronized.

After gathering data without TSN in the network we introduced the TSN switch and made comparisons. The results showed that the introduction of TSN reduced jitter since the MTSN Soc-E kit has more precise hardware, designed for real-time capabilities, and uses a hardware clock as opposed to the Raspberry Pis. The Raspberry Pis are neither equipped with hardware clocks nor real-time capabilities except for some light functionality to prioritize processes in the Raspberry Pi OS. The Raspberry Pis not having hardware clocks is a limitation in the experiments due to increased jitter and less precise data. Although, the lack of precise end systems can also be considered plausible in real-world scenarios.

The results from the initial experiments partially answered our questions about what happens when we introduce the TSN switch, which is that jitter was reduced at the cost of reception drift being introduced. With this drift in mind, we wanted to see what long-term effects drift in the network could cause. To achieve this, we constructed extreme cases of drift, both positive and negative. The results showed that in the case of negative clock drift, packets missed their TSN window since the TSN switch is keeping a schedule that is sending scheduled traffic at a faster rate than the endpoints. In the other case, with a positive reception drift, packets are completely lost because the TSN switch queue buffers become full, and the switch must empty them. These effects can have dire consequences when occurring in a production setting or a finished product like a car.

With this work, we have iteratively analyzed and designed experiments and made implementations which we then have observed and evaluated. In the last iteration of experiments, we tried to counteract the clock drift by either statically changing the cycle time

in the TSN switch or dynamically changing the sender interval according to the measured clock drift. Our results indicate that we are managing the clock drift in both cases. In the first case, with the static approach, it is easier to measure how well the solution works. As described in section 7.3.1, it should take approximately 16 years before a packet drifts a whole second off schedule. Since the drift is positive, the packet would result in being queued. In the other example, with the dynamic solution, it is harder to estimate how well the solution works since it is dynamically updating the sender node. We have at least proven it worked over 70 000 packets (approximately 19 hours), which is a vast improvement over having packets missing their window every 99 seconds. We expect this solution to function permanently even if the system is subjected to environmental changes, such as temperature or pressure changes that could modify the different clocks involved.

Out of the dynamic and static solution, no solution is better than the other, but rather it depends on the application. In homogenous networks, where the drift is deterministic and shared between all endpoints, both the static and dynamic solution are applicable. The static solution could theoretically work forever if the drift is accurately measured and stable. Although the solution is susceptible to changes in drift, for example when the system is exposed to changes in temperature or pressure. However, server rooms can be considered controlled environments where the static solution could work indefinitely.

If the network is homogenous and experiences a change in drift, then the dynamic solution would still work since it can adapt several endpoints to the same drift. The dynamic solution might instead experience issues in heterogeneous networks where the drift of one set of endpoints changes differently to another set. If the dynamic solution registers these different drifts, and changes the individual periods accordingly, eventually a collision may occur unless the TSN switch also is rescheduled. However, rescheduling in TSN is a known nondeterministic polynomial problem (NP). Rescheduling the switch becomes an increasingly complex problem when the network grows, meaning that it is not feasible to reschedule the switch dynamically. In the example in Figure 8-1 we are sending two different messages that are scheduled within TSN. The periods are first the same but after measurements, they are changing, period one is extended by 25%, and period two is shortened by 25%. After the periods are updated, we can see how the messages collide. If the changes are to be done in the switch it calls for re-scheduling of the TSN network which we already mentioned is not feasible in real implementations.

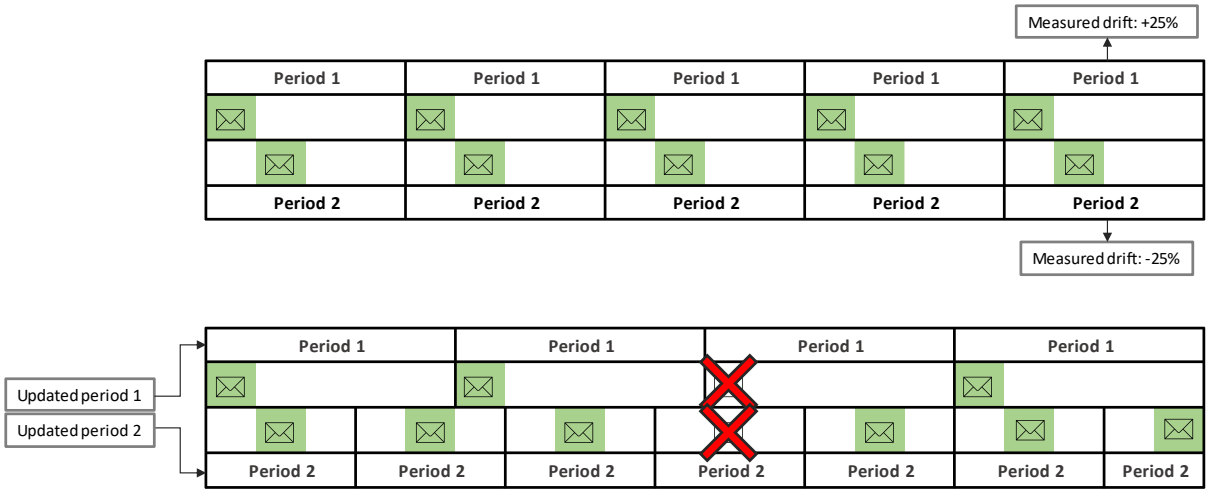

Figure 8-1: *Dynamic solution registers different types of drift*

Regarding the last question in the problem formulation, according to our results, it is possible to counteract the drift we found in our small heterogeneous network, and it is possible without interfering with the switch. However, we see problems in the static solution if there are dynamic changes in drift and problems with the dynamic solution in heterogeneous networks if different sets of endpoints experience different drift.

9. Conclusions

With this work, we aimed to study the effects of keeping a heterogeneous TSN network partially synchronized. This is motivated by cases where legacy systems are not able to be fully integrated with a TSN network due to a lack of clock synchronization support. There are also cases where cheap units like microcontrollers might support clock synchronization between each other but lack compatible synchronization mechanisms with TSN. We wanted to analyze jitter, as well as reception time drift in the end stations when they are used in a TSN network. Moreover, we wanted to investigate if it is possible to counteract any negative effects of keeping the network partially synchronized.

We have analyzed a simple network of two nodes sending time-triggered traffic from a sender to a receiver. First, we started by laying a foundation to show the effects of having the two nodes unsynchronized and synchronized, we could then progress to analyzing the effects of adding a TSN switch in between these nodes. Our results show that jitter in the receiver node is reduced by TSN, but at the cost of synchronization throughout the network since the switch is not synchronized with the endpoints. These results answered our first research question: "*With regard to clock drift and jitter, what are the effects of only synchronizing endpoints through their legacy synchronization mechanisms in a heterogeneous TSN network?"* We showed that there are negative effects of having a partially synchronized TSN network. We also analyzed the long-term effects of letting the network experience positive and negative drift between the TSN switch and the Raspberry Pi endpoints. These results showed that in the case of negative reception drift in the receiver node, after drifting a whole period, packets missed their TAS window. In the case of positive reception drift, packets were instead queued until packet drops occurred from the buffer in the switch. We have not shown exactly when the drop happens, since it is outside of the scope of this thesis, but we have shown that negative effects exist.

Our subquestion was: *"If the effects result in clock drift between devices, in what ways can we manage this and is it possible without interfering with the TSN switch?"*. To answer this question and solve the issues we have designed two solutions. The first solution consists of measuring packet reception drift as accurately as possible in the receiver node and applying this change in the TSN switch. The second solution consists of continuously calculating the drift in intervals by timestamping packets in the receiver node. After calculations, the receiver node updates the sender node of the perceived reception drift which the sender node adjusts to. Through our results, we have shown that our solutions have had positive effects on the reception drift. In the dynamic solution, we showed that it was possible to reduce drift by dynamically and automatically changing the sending period in the sender node. The period used for the experiment drifted circa one millisecond per second and missed its TAS window every 100th packet. When executing the dynamic solution, we showed that another packet did not miss a TAS window for the duration of the experiment, which was running for over 70000 packets, or approximately 19 hours. In the static solution, we measured a real reception drift

of negative eight microseconds per second. Then we showed that we could statically change the cycle time in the switch, and subsequently measure a clock drift of two nanoseconds per second which is 4000 times smaller clock drift.

10. Future work

On one hand, this work lends itself to a plethora of different directions for continuing the work. The work could be improved by writing code to accurately measure the drift of the dynamic solution, which has not been done yet. It is currently only possible to measure the duration the dynamic solution runs an experiment without experiencing critical errors, such as packets getting queued or missing TAS windows. On the other hand, it would be interesting to investigate the effects in heterogeneous networks when introducing different sets of endpoints with different synchronization. It is also possible to build on the dynamic solution and show whether the hypothesis in the discussion section and Figure 8-1 is correct when packets collide.

Currently, in the dynamic solution, the periods are changed in the sender node. We would like to propose a solution where periods instead are changed in the switch, since legacy devices may be less flexible than TSN switches. Another consideration would be to research the possibility of dynamically rescheduling the TSN switch. Since rescheduling is an NP-problem it would be interesting to investigate faster scheduling methods to reduce scheduling time by using heuristic algorithms[7] and how to propagate these changes throughout the TSN network.

[7] A heuristic algorithm aims to simplify problems to solve them faster and more efficient, on the cost of accuracy and optimality [26].

Appendix A. Switch configuration

The TSN switch was configured according to the following steps:

(1) Connection to the switch is done via Ethernet, in this case, we connected switch port 0 to a PC.
(2) By default, the switch is accessible on IP address 192.168.4.64 which can be reached via a web browser. Note that it is necessary for the device connecting to the switch to be in the same subnet as the switch.
(3) To access the TAS scheduling, go to Advanced Network, see Figure A-1.
(4) In Advanced Network the gate was enabled for port 2, see Figure A-2. This port is used by the device which is receiving traffic. The other ports were left unconfigured.
(5) TAS was subsequently configured according to Figure A-3. In the figure, it can be observed that the cycle time was set to one second since the experiment is using periodic traffic in one-second intervals. The list length is set to two. The interval for Q7 which is the queue for the experiment traffic of high priority is set to 7400 ns. The reason for this is explained in section 6.2.3.1 *Adjusting cycle time in the switch*. Other traffic traversing the network is allowed to be sent in Q0 outside the 7400 ns interval.

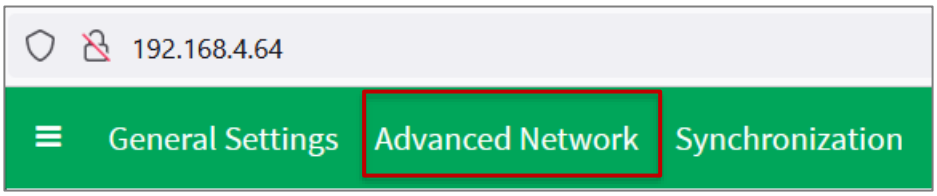

Figure A-1*: Switch menu*

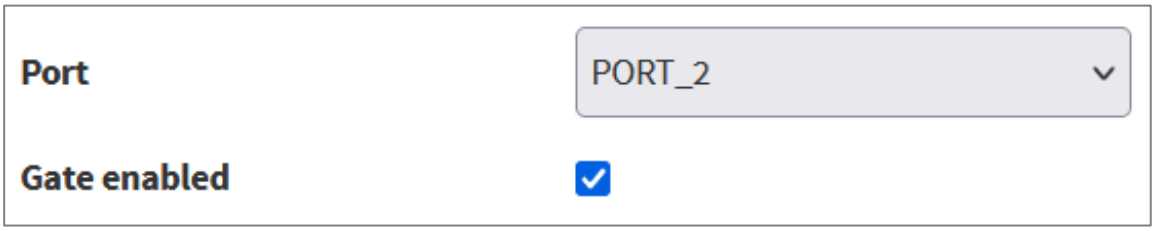

Figure A-2: *Switch port configuration*

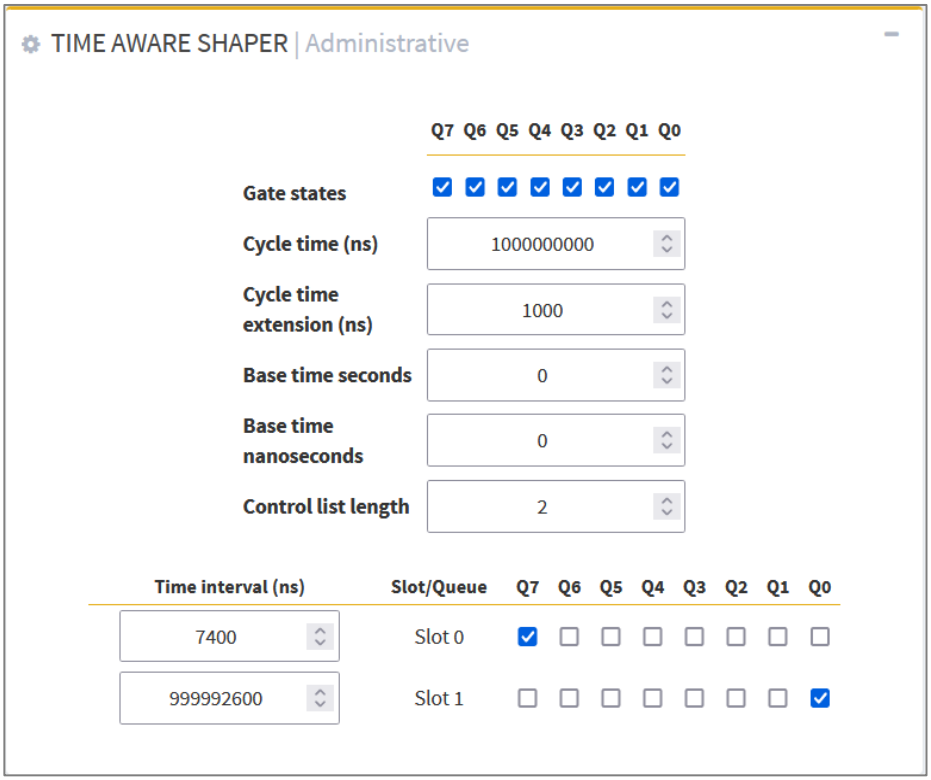

Figure A-3: *TAS configuration.*

Appendix B. Simple sender node code

Table B-1 presents the code excerpt for a python 3 file named traffic.py. This code was used in the implementations where only an Internet Control Message Protocol (ICMP) packet was sent in a periodic interval without any payload or communication back from the receiver node.

The code was created by Nguyen and Nasiri [9] with slight modifications to adapt to the devices used in this thesis. To run the code on Raspberry Pi OS or other Debian-based operating systems (OS) it is necessary to install the package Scapy which is done with the command **`sudo pip3 install scapy`**. The code is modifying the Virtual LAN (VLAN) tag to make the packets high priority. The code is run with the command **`python3 traffic.py.`**

```
1.  # authors: Andreas Johansson <ajn19017@sudent.mdu.se>
2.  #          Built upon the program by:
3.  #          Thien Phong David Nguyen <tnn18006@student.mdu.se>
4.  #          and
5.  #          Zack Nasiri <zni18001@student.mdu.se>
6.  # course.: DVA333 Thesis for the Degree of Bachelor of Science in Computer Network Engineering
7.  # date...: 17 Mar 2022
8.  #
9.  ##### Importing neccessary modules #####
10. import time, sys, os
11. from scapy.all import *
12. from scapy.utils import *
13.
14. #This program modifies an ICMP packets ethernet header with Scapy and sends it periodically to
    a receiver. The period is 1 second.
15. #To use the program: Change IP:s and MAC-addresses as well as interface (iface) to the
    corresponding devices IP/MAC/interface
16. #If the file is renamed the traffic.py needs to be changed as well.
17.
18. ##### Set the priority of the script to real-time and to the highest priority #####
19. sudoPrio = os.popen("ps -aux | grep traffic.py | awk '{print $2}' | head -1").read()
20. prosPrio = os.popen("ps -aux | grep traffic.py | awk '{print $2}' | awk 'NR==2'").read()
21.
22. os.system("sudo chrt -p 99 " + str(sudoPrio))
23. os.system("sudo chrt -p 99 " + str(prosPrio))
24. os.system("sudo renice -20 -p " + str(sudoPrio))
25. os.system("sudo renice -20 -p " + str(prosPrio))
26.
27. ###### Code to manipulate the Ethernet header #####
28. frame = (Ether(dst='2c:44:fd:10:32:a1',src='b8:27:eb:95:42:a2')
29.   / Dot1Q(vlan=1)
30.   / Dot1Q(prio=7)
31.   / IP(dst='192.168.4.100',src='192.168.4.200')
32.   / ICMP())
33.
34. #Main thread, periodic sending of traffic of 1 second
35. q = time.time()
36. p = time.time()
37. i=0
38. try:
39.   while True:
40.             while p < q-k:
41.                       p = time.time()
42.             sendp(frame, iface='eth0')
43.             q+=x
44.             i+=1
```

Table B-1: *Code for periodic traffic*

Appendix C. Sender node code, dynamic solution

This code (Table C-1) is the first part of a two-way communication solution where the receiver node updates the sender node with a perceived clock drift value. The receiver part of the code can be found in Appendix D. To run the code, see the necessary information in Appendix B.

```
1.  import time, sys, os, socket, threading
2.  from scapy.all import *
3.  from scapy.utils import *
4.
5.  #Set up a simple TCP server
6.  server_address = ('192.168.4.200', 1338)
7.  sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
8.  sock.bind(server_address)
9.  sock.listen(1)
10.
11. #Traffic period in seconds = x and slope value = k. As in y=kx+m
12. x=1000000000
13. k=0
14.
15. #Function to run TCP server in separate thread
16. def data_receiver():
17.   global k, x
18.   connected = False
19.   while True:
20.             if connected == False:
21.                       connection, client_address = sock.accept()
22.                       print("Client has connected")
23.                       connected = True
24.             else:
25.                       data = connection.recv(1024)
26. #                     k=int(data.decode())
27. #                     x+=k
28.                       print("New drift received: " + str(k))
29.                       print("New schedule:" + str(x) + "nanoseconds")
30.
31.
32. ##### Set the priority of the script to real-time and to the highest priority #####
33. sudoPrio = os.popen("ps -aux | grep traffic_with_TCP.py | awk '{print $2}' | head -1").read()
34. prosPrio = os.popen("ps -aux | grep traffic_with_TCP.py | awk '{print $2}' | awk
    'NR==2'").read()
35.
36. os.system("sudo chrt -p 99 " + str(sudoPrio))
37. os.system("sudo chrt -p 99 " + str(prosPrio))
38. os.system("sudo renice -20 -p " + str(sudoPrio))
39. os.system("sudo renice -20 -p " + str(prosPrio))
40.
41. #Start data receiver thread
42. thread_0 = threading.Thread(target = data_receiver)
43. thread_0.start()
44.
45. #Forge ethernet frame used for periodic sending
46. frame = (Ether(dst='2c:44:fd:10:32:a1',src='b8:27:eb:95:42:a2')
47.   / Dot1Q(vlan=1)
48.   / Dot1Q(prio=7)
49.   / IP(dst='192.168.4.100',src='192.168.4.200')
50.   / ICMP())
51.
52. #Main thread, periodic sending of traffic in x seconds adjusted with slope value from receiver
53. q = time.time_ns()
54. p = time.time_ns()
55. i=0
56. try:
57.   while True:
58. #           print("p " + str(p) + " q-k " + str(q+k)) #Used to validate that period is
    updating when receiving new slope (k) value from receiver node
```

```
59.             while p < q-k:
60.                     p = time.time_ns()
61.             sendp(frame, iface='eth0')
62.             q+=x
63.             i+=1
64.             print("Sending packet #%d ST: %s" %(i,str(p)))
65.
66. except KeyboardInterrupt:
67.   print('\033[2DBye.')
68.   thread_0.join()
```

Table C-1: *Sender node code, dynamic solution*

Appendix D. Receiver node code, dynamic solution

This code (Table D-1) is the second part of a two-way communication solution where the receiver node updates the sender node with a perceived clock drift value. The sender part of the code can be found in Appendix C. To run the code, see the necessary information in Appendix B.

```
1.  from scapy.all import *
2.  import time, sys, os, socket
3.
4.  pkt_count=21
5.  current_slope=0
6.  old_slope=0
7.
8.  server_address=('192.168.4.200',1338)
9.
10. ##### Set the priority of the script to real-time and to the highest priority #####
11. sudoPrio = os.popen("ps -aux | grep receiver_new_formula.py | awk '{print $2}' | head -
    1").read()
12. prosPrio = os.popen("ps -aux | grep receiver_new_formula.py | awk '{print $2}' | awk
    'NR==2'").read()
13.
14. os.system("sudo chrt -p 99 " + str(sudoPrio))
15. os.system("sudo chrt -p 99 " + str(prosPrio))
16. os.system("sudo renice -20 -p " + str(sudoPrio))
17. os.system("sudo renice -20 -p " + str(prosPrio))
18.
19. try:
20.   client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
21. except socket.error:
22.   print("Socket failed. Exiting..")
23.   sys.exit()
24.
25. print("Socket Created, connecting to " + str(server_address[0]) + " port " +
    str(server_address[1]))
26. try:
27.   client.connect(server_address)
28. except ConnectionRefusedError:
29.   print("Could not connect to server. Exiting..")
30.   sys.exit()
31. print("Connection established.")
32.
33. ## Setup sniff, filtering for IP traffic
34.
35. while True:
36.   pkt = sniff(iface="eth0",filter='vlan 1',count=pkt_count)
37.   file=wrpcap('file.pcap',pkt,nano=True)
38.   nanopacket = rdpcap('file.pcap')
39.
40.   n = pkt_count-1
41.   sum_x=0
42.   sum_y=0
43.   sum_xy=0
44.   sum_squared_x=0
45.   delta = [None] * pkt_count      #Declare an array
46.
47.   for i in range(0, n, 1):        #Save all y-values to the array
48.             delta_y[i] = int((nanopacket[i+1].time-nanopacket[0].time-(i+1))*1000000000)
49.
50.   for i in range(1, n+1, 1):      #Sum x and x^2
51.             sum_x+=i
52.             sum_squared_x+=i*i
53.
54.   for i in range(0, n, 1):        #Sum y and xy
55.             sum_y+=delta_y[i]
56.             sum_xy+=((i+1)*delta_y[i])
57.
```

```
58.    slope = int((n*sum_xy - sum_x*sum_y) / (n*sum_squared_x - sum_x*sum_x)) #This is the
    trendline formula for the slope only
59.    current_slope = slope-old_slope
60.    print("Slope of last 20 packets: ", str(slope))
61.    print("Slope update to sender node: " + str(current_slope) + " nanoseconds per second")
62.
63.    if current_slope == 0:
64.              print("Drift is stable")
65.    else:
66.              print("Sending update to sender node")
67.              client.sendall(str(current_slope).encode())
68.              old_slope+=current_slope
69.    file = open('slopes.txt', 'a')
70.    file.write(str(slope)+"\n")
71.    file.close()
72.
```

Table D-1: *Receiver node code, dynamic solution*

Appendix E. Sender node code with payload

This code (Table E-1) sends periodic User Datagram Protocol (UDP) packets with a payload of "packet x" where x represents which packet is currently being sent. This is done to identify any packet losses in the receiver node. To run the code, see the necessary information in Appendix B.

```
1.  import time, sys, os, socket, threading
2.  from scapy.all import *
3.  from scapy.utils import *
4.
5.  #Set up a simple TCP server
6.  server_address = ('192.168.4.200', 1338)
7.  sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
8.  sock.bind(server_address)
9.  sock.listen(1)
10.
11. #Traffic period in seconds = x and slope value = k. As in y=kx+m
12. x=500000000
13. k=0
14.
15. #Function to run TCP server in separate thread
16.
17. ##### Set the priority of the script to real-time and to the highest priority #####
18. sudoPrio = os.popen("ps -aux | grep traffic_with_payload.py | awk '{print $2}' | head -
    1").read()
19. prosPrio = os.popen("ps -aux | grep traffic_with_payload.py | awk '{print $2}' | awk
    'NR==2'").read()
20.
21. os.system("sudo chrt -p 99 " + str(sudoPrio))
22. os.system("sudo chrt -p 99 " + str(prosPrio))
23. os.system("sudo renice -20 -p " + str(sudoPrio))
24. os.system("sudo renice -20 -p " + str(prosPrio))
25.
26.
27. #Forge ethernet frame used for periodic sending
28. def forge_frame(i):
29.   global frame
30.   frame = (Ether(dst='2c:44:fd:10:32:a1',src='b8:27:eb:95:42:a2')
31.             / Dot1Q(vlan=1)
32.             / Dot1Q(prio=7)
33.             / IP(dst='192.168.4.100',src='192.168.4.200')
34.             / UDP(sport=9999, dport=9999)
35.             / Raw(load="packet " + str(i)))
36. #Main thread, periodic sending of traffic in x seconds adjusted with slope value from receiver
37. q = time.time_ns()
38. p = time.time_ns()
39. i=1
40. try:
41.   while True:
42.             forge_frame(i)
43. #           print("p " + str(p) + " q-k " + str(q+k)) #Used to validate that period is
    updating when receiving new slope (k) value from receiver node
44.             while p < q-k:
45.                       p = time.time_ns()
46.             sendp(frame, iface='eth0')
47.             q+=x
48.             i+=1
49.             print("Sending packet #%d ST: %s" %(i,str(p)))
50.
51. except KeyboardInterrupt:
52.   print('\033[2DBye.')
53.   thread_0.join()
```

Table E-1: *Sender node code with payload*

References

[1] S. Floyd and M. Allman, "Comments on the Usefulness of Simple Best-Effort Traffic," ICSI., Berkeley, CA, USA, Tech. Memo. RFC 5290, Jul. 2008.

[2] J. Wang, Real-Time Embedded Systems (Quantitative Software Engineering Series), New York: John Wiley & Sons Inc , 2017.

[3] F. M. Pozo Pérez, "Methods for Efficient and Adaptive Scheduling of Next-Generation Time-Triggered Networks," PhD dissertation, Mälardalen University, Västerås, Sweden, 2019.

[4] D. Bujosa, I. Álvarez and J. Proenza, "CSRP: An Enhanced Protocol for Consistent Reservation of Resources in AVB/TSN," *IEEE Transactions on Industrial Informatics*, vol. 17, pp. 3640-3650, 2021.

[5] H. Lee, J. Lee, C. Park and S. Park, "Time-aware preemption to enhance the performance of Audio/Video Bridging (AVB) in IEEE 802.1 TSN," in *2016 First IEEE International Conference on Computer Communication and the Internet (ICCCI)*, 2016.

[6] D. B. Mateu, D. Hallmans, M. Ashjaei, A. V. Papadopoulos, J. Proenza and T. Nolte, "Clock Synchronization in Integrated TSN-EtherCAT Networks," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2020.

[7] M. Barzegaran, N. Reusch, L. Zhao, S. Craciunas and P. Pop, *Real-Time Guarantees for Critical Traffic in IEEE 802.1Qbv TSN Networks with Unscheduled and Unsynchronized End-Systems*, 2021.

[8] M. Ashjaei, private communication, Dec 2021.

[9] D. Nguyen and Z. Nasiri, "Performance analysis of a non-synchronized heterogeneous TSN network," BSc thesis, Mälardalen University, Västerås, Sweden, 2022.

[10] C. E. Spurgeon, Ethernet: The Definitive Guide, Sebastopol, CA, USA: O'Reilly Media, 2000.

[11] IEEE 802.3 NGOATH Study Group, "IEEE P802.3bs,", IEEE802. https://www.ieee802.org/3/bs/NGOATH_3bs_Objectives_16_0122.pdf (accessed Jan. 28, 2022).

[12] Cisco Networking Academy, Routing and Switching Essentials v6 Companion Guide, Indianapolis, IND, USA: Cisco Press, 2017.

[13] Z. Wang, Y.-q. Song, J.-m. Chen and Y.-x. Sun, "Real time characteristics of Ethernet and its

improvement," in *Proceedings of the 4th World Congress on Intelligent Control and Automation (Cat. No.02EX527)*, 2002.

[14] P. Meyer, T. Steinbach, F. Korf and T. C. Schmidt, "Extending IEEE 802.1 AVB with time-triggered scheduling: A simulation study of the coexistence of synchronous and asynchronous traffic," in *2013 IEEE Vehicular Networking Conference*, 2013.

[15] L. Zhao, P. Pop, Z. Zheng and Q. Li, "Timing Analysis of AVB Traffic in TSN Networks Using Network Calculus," in *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018.

[16] S. Thangamuthu, N. Concer, P. J. L. Cuijpers and J. J. Lukkien, "Analysis of Ethernet-switch traffic shapers for in-vehicle networking applications," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2015.

[17] C. Le and D. Qiao, "Evaluation of Real-Time Ethernet with Time Synchronization and Time-Aware Shaper Using OMNeT++," in *2019 IEEE 2nd International Conference on Electronics Technology (ICET)*, 2019.

[18] "IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications," *IEEE Std 802.1AS-2020 (Revision of IEEE Std 802.1AS-2011)*, pp. 1-421, 2020.

[19] H.-T. Lim, D. Herrscher, M. J. Waltl and F. Chaari, "Performance Analysis of the IEEE 802.1 Ethernet Audio/Video Bridging Standard," in *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, Brussels, 2012.

[20] L. Zhao, P. Pop and S. Steinhorst, *Quantitative Performance Comparison of Various Traffic Shapers in Time-Sensitive Networking*, Mar 2021, arXiv:2103.13424.

[21] S. Chouksey, H. S. Satheesh and J. Åkerberg, "An Experimental Study of TSN-NonTSN Coexistence," in *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, 2021.

[22] M. Kim, J. Min, D. Hyeon and J. Paek, "TAS Scheduling for Real-Time Forwarding of Emergency Event Traffic in TSN," in *2020 International Conference on Information and Communication Technology Convergence (ICTC)*, 2020.

[23] K. Säfsten and M. Gustavsson, *Forskningsmetodik för ingenjörer och andra problemlösare*, 1:2 ed., Lund, Sweden, Studentlitteratur AB, 2019.

[24] J. F. Nunamaker, M. Chen and T. D. Purdin, "Systems development in information," *Journal of management information systems*, vol. 7, no. 3, pp. 97-101, 1990.

[25] N. Agarwal, "Computer Network | Quality of Service and Multimedia," GeeksforGeeks. https://www.geeksforgeeks.org/computer-network-quality-of-service-and-multimedia/ (accessed Feb. 1, 2022).

[26] K. Vincent, N. Matthew and S. Spencer, "Heuristic algorithms,". Optimization. https://optimization.mccormick.northwestern.edu/index.php/Heuristic_algorithms (accessed May. 6, 2022).