OCORA

Open CCS On-board Reference Architecture

CCS Communication Network

Proof of Concept (PoC)

This OCORA work is licensed under the dual licensing Terms EUPL 1.2 (Commission Implementing Decision (EU) 2017/863 of 18 May 2017) and the terms and condition of the Attributions- ShareAlike 3.0 Unported license or its national version (in particular CC-BY-SA 3.0 DE).

Document ID: OCORA-TWS02-020

Version: 4.00

Date: 02.12.2022

Management Summary

Today the interfaces between CCS components on the vehicle are proprietary. The proprietary interfaces do not allow to exchange CCS components from different suppliers. The OCORA architecture [7] aims for plug and play interchangeability within the CCS domain through the specification of a generic, open and standardized communication backbone, the CCS Communication Network (CCN) formerly called Universal Vital Control and Command Bus (UVCCB). The CCN itself will be modifiable in accordance with future technological evolutions by means of strict separation of the different communication layers (OSI Layers).

This document was jointly elaborated with the CCN evaluation report [8]. The CCN evaluations done in former release phases proposed the CCN to be a TSN Ethernet based network. The CCN re-evaluations done in this release phase have shown that TRDP on top of UDP/IP fulfils all requirements for onboard CCS communication if additional Quality of Service (QoS) mechanisms according to IEEE 802.1Q are applied to standard Ethernet (see Table 21 in chapter 3.3.2). On session layer, TRDP is the right choice for the main cyclic process data used for the business logic of the onboard CCS system. As TRDP is designed and standardized by the railway industry, it is simple to evolve the protocol into the direction the railway sector wants to. This is not the case for the other automation industry or automotive driven session layer protocols. And since most of the ECN networks in TCMS domain will use TRDP, a one common bus for CCS and TCMS domain would become possible with the use of TRDP.

The proposed protocol stack of CCN is listed in the following table.

Layer	Protocol	Standard
(Safety Layer[1])	(SDTv2/v4)	IEC 61375-2-3 and [9]
Session Layer	TRDP	IEC 61375-2-3
Transport Layer	UDP TCP	RFC 768 RFC 793
Network Layer	IPv4	RFC 791
Data Link Layer	Standard Ethernet with QoS	IEEE 802.3 IEEE 802.1Q
Physical Layer	1000BASE-T (optionally 100BASE-TX for end devices)	IEEE 802.3 Clause 40 IEEE 802.3 Clause 25

Table 1: Protocol Stack CCN

This document is reporting the task definitions and results of the proof-of-concept (PoC) elaborated during release phases R1 to R3. With the PoC, it was possible to verify and give a deeper insight into the different TSN-mechanisms. These mechanisms are quite specific and add functionalities to the ethernet protocol that cannot be achieved otherwise, such as seamless redundancy on network level, time scheduling of the streams and the preemption of frames. The different mechanisms were investigated in isolation so that the resulting effects could be clearly seen and attributed to the mechanism under consideration.

The main value of the TSN protocols considered is to improve the predictability of the network. By being able to completely separate different streams in the time domain and by providing a way to pre-empt frames of lower priority while they are being sent, eliminates random delays of messages and makes it possible to define upper bounds on the latency, thus making the network deterministic. However, the random delays eliminated are on the order of the length of a message. For fast networks (> 1 Gbit/s), these delays are rather small and on the order of a few microseconds. Even if compounded over a path with several switches the gains remain small if an overall latency in the millisecond range is aimed at.

Next to the working of the mechanism the relative complexity of the solutions could be noticed. It is not clear how configurations can be generalised and scaled up to a complex network in operation. This is especially valid for the time-scheduling (IEEE 802.1Qbv) and the redundancy protocols (IEEE 802.1CB) as these protocols, need a network wide coordination of the devices. IEEE 802.1Qbu can be configured on a per link basis and thus involves less oversight for it to be implemented.

Moreover, as was seen when integrating TSN into a network stack like TRDP, the latency of the network is small compared to the latencies of the software stacks on the sending and receiving side, even though the stack was executed on a Linux system with a real time patch. Thus, the full potential of the TSN protocols is limited by these factors outside of the networks scope, and the advantage of TSN over usual QoS techniques is lessened. Indeed, tests using usual TRDP Process Data and QoS techniques showed a similar overall

[1] Safety Layer is only applicable for safety-related data traffic.

performance in the test setup under consideration.

Another finding was that the TRDP protocol stack even though it starts to support TSN messages, needs a lot of application dependent configuration and adjustments to fully work. A stack with TSN support ready out of the box does not seem to exist yet.

For current requirements on latencies (10-100 ms, see CCN-07 in the evaluation report [8]), the additional complexity and effort that comes with a TSN network does not seem reasonable for the relatively small gains compared to a well set up priority system using classical QoS techniques. This applies especially to the scheduling (IEEE 802.1Qbv) and seamless redundancy (IEEE 802.1CB). A use of classical QoS with frame preemption (IEEE 802.1Qbu) to reduce the mean time a priority message takes to transfer through a switch could however be considered.

Revision history

Version	Change Description	Initial	Date of change
1.00	Official version for OCORA Delta Release	SSt	30.06.2021
2.00	Official version for OCORA Release R1	SSt	03.12.2021
3.00	Official version for OCORA Release R2	SSt	08.06.2022
4.00	Official version for OCORA Release R3	SSt	02.12.2022

Table of contents

Table of figures

Table of tables

References

Reader's note: please be aware that the numbers in square brackets, e.g. [1], as per the list of referenced documents below, is used throughout this document to indicate the references to external documents. Wherever a reference to a TSI-CCS SUBSET is used, the SUBSET is referenced directly (e.g. SUBSET-026). OCORA always reference to the latest available official version of the SUBSET, unless indicated differently.

[1] OCORA-BWS01-010 – Release Notes

[2] OCORA-BWS01-020 – Glossary

[3] OCORA-BWS01-030 – Question and Answers

[4] OCORA-BWS01-040 – Feedback Form

[5] OCORA-BWS03-010 – Introduction to OCORA

[6] OCORA-BWS04-011 – Problem Statements

[7] OCORA-TWS01-030 – System Architecture

[8] OCORA-TWS02-010 – CCS Communication Network – Evaluation

[9] CTA2-T3.4-T-SIE-019-03 – Safety Analysis SDTv4

[10] User guide, PCIE-0400-TSN, Doc. Rev. 0.16, Doc-ID: 1062-6228, Kontron AG

[11] Slate XNS User Manual, D-TTE-G-11-001, Revision 2.0.6

[12] Linux manual page, tc-etf(8) - https://man7.org/linux/man-pages/man8/tc-etf.8.html

[13] ELC-2018-USA-TSNonLinux.pdf

Introduction

1.1 Purpose of the document

This document summarizes the work done in the workstream CCS Communication Network (CCN) regarding the Proof of Concept (PoC). It documents the foreseen tasks and the results of the PoC.

This document is addressed to experts in the CCS domain and to any other person, interested in the OCORA concepts for on-board CCS. The reader is invited to provide feedback to the OCORA collaboration and can, therefore, engage in shaping OCORA. Feedback to this document and to any other OCORA documentation can be given by using the feedback form [4].

If you are a railway undertaking, you may find useful information to compile tenders for OCORA compliant CCS building blocks, for tendering complete on-board CCS system, or also for on-board CCS replacements for functional upgrades or for life-cycle reasons.

If you are an organization interested in developing on-board CCS building blocks according to the OCORA standard, information provided in this document can be used as input for your development.

1.2 Applicability of the document

The document is currently considered informative but may become a standard at a later stage for OCORA compliant on-board CCS solutions. Subsequent releases of this document may be developed based on a modular and iterative approach, evolving within the progress of the OCORA collaboration.

1.3 Context of the document

This document is published as part of the OCORA Release R3, together with the documents listed in the release notes [1]. Before reading this document, it is recommended to read the Release Notes [1]. If you are interested in the context and the motivation that drives OCORA we recommend to read the Introduction to OCORA [5], and the Problem Statements [6]. The reader should also be aware of the Glossary [2] and the Question and Answers [3].

1.4 Renaming

The CCS Communication Network (CCN) was formerly called Universal Vital Control and Command Bus (UVCCB). The evaluations on different communication layers [8] concluded to use an Ethernet network as communication backbone. Therefore, the UVCC-Bus was renamed to CCS Communication Network.

1.5 Problem Description

Today the interfaces between CCS components on the vehicle are proprietary. The proprietary interfaces do not allow to exchange CCS components from different suppliers. The vendor lock-in created by proprietary interfaces leads to high costs. The existing proprietary interfaces do not allow to add new functions or make use of the available data by other components.

Moreover, these interfaces are implemented using heterogeneous fieldbus technologies. This leads to increased complexity and extensive effort for the operator/maintainer to handle these heterogeneous systems.

1.6 Concept

The OCORA architecture [7] aims for plug and play interchangeability within the CCS on-board domain through isolation of specific functions in combination with the specification of a generic, open and standardized communication backbone, the CCS Communication Network (CCN). In the following figure the final physical view of the OCORA architecture [7] is shown. The CCN connects different components of the future CCS on-board systems as for example:

- European Train Protection On-Board (ETP-OB)
- Train Display System (TDS-OB)
- National Train Control System (NTC) or Specific Transmission Module (STM)
- Safe Computing Platform (SCP)
- Gateway to Train Control Management System Network, Operator Network, Communication Network or Security Network (ECN/ECN Security Gateway)

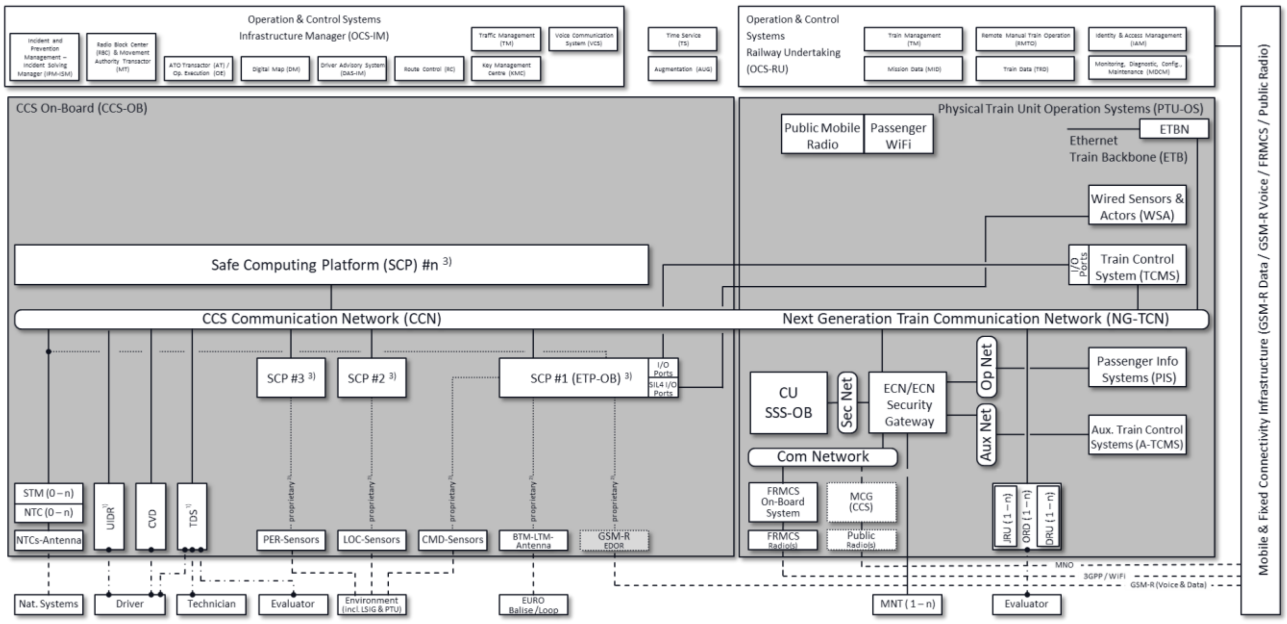

Operation & Control Systems Infrastructure Manager (OCS-IM)
Operation & Control Systems Railway Undertaking (OCS-RU)
CCS On-Board (CCS-OB)
Physical Train Unit Operation Systems (PTU-OS)
Safe Computing Platform (SCP) #n 3)
CCS Communication Network (CCN)
Next Generation Train Communication Network (NG-TCN)
SCP #3 3)
SCP #2 3)
SCP #1 (ETP-OB) 3)
Public Mobile Radio
Passenger WiFi
ETBN
Ethernet Train Backbone (ETB)
Wired Sensors & Actors (WSA)
Train Control System (TCMS)
Passenger Info Systems (PIS)
Aux. Train Control Systems (A-TCMS)
CU SSS-OB
Sec Net
ECN/ECN Security Gateway
Op Net
Aux Net
Com Network
FRMCS On-Board System
MCG (CCS)
STM (0 – n)
NTC (0 – n)
NTCs-Antenna
UIDR
CVD
TDS
PER-Sensors
LOC-Sensors
CMD-Sensors
BTM-LTM-Antenna
GSM-R EDOR
Nat. Systems
Driver
Technician
Evaluator
Environment (incl. LSIG & PTU)
EURO Balise/Loop
MNT (1 – n)
Evaluator
MNO
3GPP / WiFi
GSM-R (Voice & Data)
Mobile & Fixed Connectivity Infrastructure (GSM-R Data / GSM-R Voice / FRMCS / Public Radio)

Figure 1: Technical architecture (final view) from [7]

In the final vision of the system an open and standardized CCN (OSI-Layers 1 to 7 & Safety Layer) ensures the safe data connection between CCS on-board components. The network allows simple upgrades / enhancements of the CCS on-board System by new functions or components. It also enables procurement on a component-based (or building block based) granularity which leads to more flexibility in the life cycle management and optimal components due to larger market size. For the CCN itself modifications due to future technological evolutions are facilitated by the communication layering concept.

The CCN evaluations done in former release phases proposed the CCN to be a TSN Ethernet based network. The CCN re-evaluations done in this release phase have shown that TRDP on top of UDP/IP fulfils all requirements for onboard CCS communication if additional Quality of Service (QoS) mechanisms according to IEEE 802.1Q are applied to standard Ethernet (see Table 21 in chapter 3.3.2). On session layer, TRDP is the right choice for the main cyclic process data used for the business logic of the onboard CCS system. The proposed protocol stack of CCN is listed in the following table.

Layer	Protocol	Standard
(Safety Layer[2])	(SDTv2/v4)	IEC 61375-2-3 and [9]
Session Layer	TRDP	IEC 61375-2-3
Transport Layer	UDP TCP	RFC 768 RFC 793
Network Layer	IPv4	RFC 791
Data Link Layer	Standard Ethernet with QoS	IEEE 802.3 IEEE 802.1Q
Physical Layer	1000BASE-T (optionally 100BASE-TX for end devices)	IEEE 802.3 Clause 40 IEEE 802.3 Clause 25

Table 2: Protocol Stack CCN

1.7 Goal

A Proof of Concept (PoC) was performed to investigate the TSN-Ethernet mechanisms, in order to judge their usefulness for the CCN network. The goal was to demonstrate that a certain amount of bandwidth for process data can be guaranteed by the network with low network latency and jitter even in congested network situations, in order to show the feasibility of a TSN-Ethernet network for the CCN. The performance of the network in congested situations was then compared between the TSN-Ethernet technology and a standard Ethernet technology with standard QoS mechanisms. The setup of the PoC also helps to investigate the technical implementation details, readiness of the ethernet technologies for the CCN. The insights of the PoC enable a better evaluation of the technologies and their usefulness for the CCN.

This document contains the current results of the Proof of Concept (PoC). The evaluation part is documented in [8].

[2] Safety Layer is only applicable for safety-related data traffic.

2 Task definition

2.1 Tasks

2.1.1 Task 1: Small TSN Network Test

A demonstrator shall show the feasibility of the CCN as a TSN-Ethernet network. In a first step a small TSN network shall be set-up. The network topology is shown in the following Figure 2. It contains two TSN capable switches, four standard PCs with standard network interface cards (NIC) and a configuration PC. Between two TSN switches a TSN connection will be established by configuring the switches accordingly through the configuration PC. The connections of the PCs will be best effort standard Ethernet connections, since no TSN MAC layer on PCs with standard network interface cards (NIC) is available. Nevertheless, with this simple network topology with only one TSN connection the TSN features can be demonstrated. The aim of this first task is to show, that the guaranteed traffic (shown in green) cannot be influenced even by a large amount of best effort traffic (shown in red). The bandwidth of the guaranteed traffic will be reserved with the scheduling scheme IEEE 802.1Qbv. To achieve a higher magnitude of bandwidth reservation the frame pre-emption scheme IEEE 802.1Qbu shall also be applied. Therefore, different priorities for the two different traffic classes must be established. This can easily be achieved by separating the traffic in two virtual LANs (VLANs) and suitably setting their priority fields.

The traffic on the network shall be analysed with an appropriate network protocol analyser (e.g. Wireshark) enabling data traffic sniffing on the switch-to-switch connection.

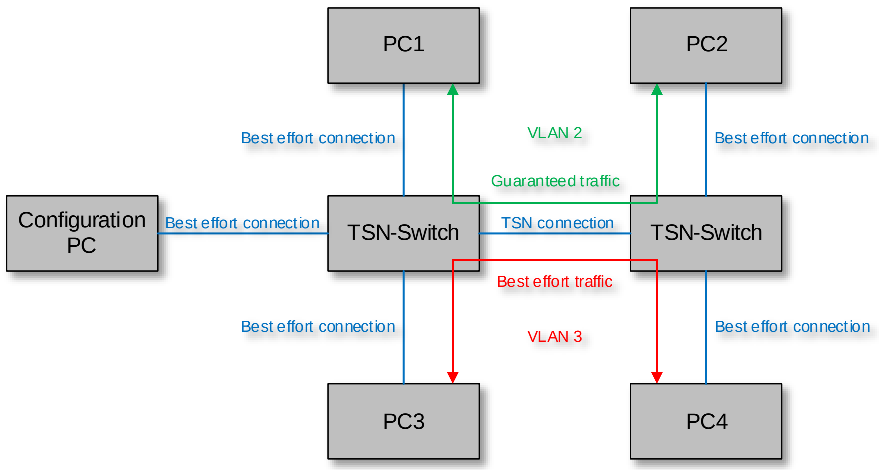


Figure 2: Network topology task 1

In this first task, the transmitted data is standard TCP/IP or UDP/IP data. Only the switch-to-switch connection is covered by TSN technology. The sender and the receiver do not notice anything of the TSN at all.

2.1.2 Task 2: End-to-End TSN Connection Test

2.1.2.1 Part a): Deterministic and Best-effort Traffic on different End Devices

In a second step an end-to-end TSN connection should be established. The end devices PC1 and PC2 should implement a TSN MAC layer to send the data packets synchronously to the network. Therefore, a session layer protocol, which can send TSN-Ethernet-Frames, should be implemented on the end devices for the guaranteed traffic. The session layer protocol shall be TRDP 2.0 or OPC UA PubSub over TSN. DDS/RTPS is not yet an option since the DDS/RTPS over TSN standard is not yet released. The protocol stack therefore corresponds to Table 1.

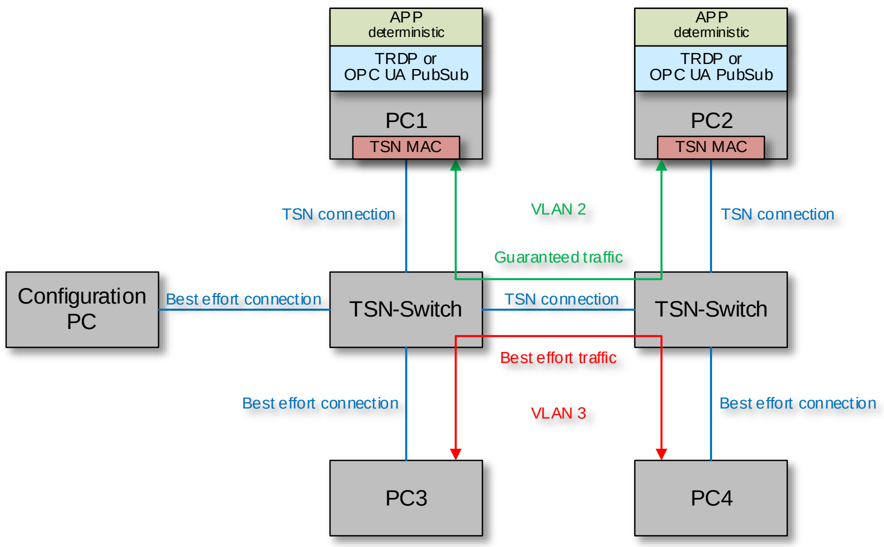


Figure 3: Network topology task 2 a)

The traffic on the network shall be analysed with a suitable network protocol analyser (e.g. Wireshark) enabling data traffic sniffing on the switch-to-switch connection.

2.1.2.2 Part b): Deterministic and Best-effort Traffic on same End Devices with different applications

To show the consolidation of best effort traffic and guaranteed traffic in a single PC, the network is reduced to comprise only PC1 and PC2. Two separate applications will run on each PC where one generates and listens to best effort traffic and the other sends and receives data over the guaranteed traffic channel. This will demonstrate the possibility to differentiate communication priorities within one end-device.

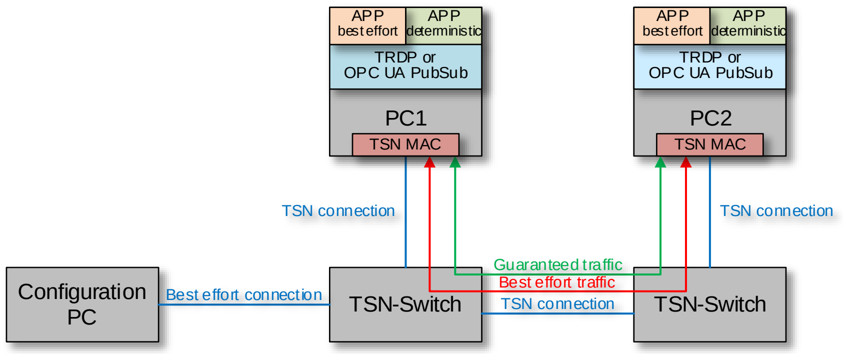


Figure 4: Network topology task 2 b)

2.1.2.3 Part c): Deterministic and Best-effort Traffic on same End Devices and within same applications

This part is similar to part b). It differs in the fact that one application is used to send 'sensitive' data using a guaranteed traffic channel and to send 'non-sensitive' data over a best-effort traffic channel. The aim is to investigate how one can segregate the traffic into different priorities at application level.

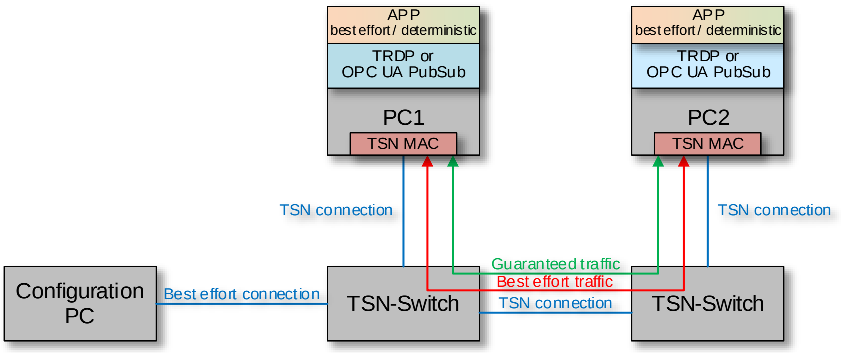


Figure 5: Network topology task 2 c)

2.1.3 Task 3: End-to-End Standard Ethernet Connection with TCNOpen TRDP

As the results of tasks 1 and 2 have shown the complexity of a TSN network and due to the re-evaluation [8] caused by the newly established SS-147 intended to be published as part of the TSI CCS 2022, in this task the performance of standard TRDP PD-packets on standard Ethernet shall also be examined. Therefore, the PD-packet timing transferred on standard ethernet connection is analysed. Additionally, the performance of the network in congested situations shall be compared between the TSN-Ethernet technology and a standard Ethernet technology with standard QoS mechanisms.

3 Task 1: Test results

3.1 Best effort traffic generated locally

3.1.1 Setup Introduction

Task 1 was realised using Kontrons TSN test-kit. The kit consists of two KBox C-102-2 industrial PCs running a Fedora Linux distribution with planet ccrma which, using the real time pre-emption patch, creates a low latency "real time" environment. The computers are fitted with TSN network interface cards providing 4 ports next to the PCIe connection to the KBox. The tests run in this section are based on the demonstration setup provided by Kontron with its TSN test-kit. To find more details please refer to the user guide ([10]) and Figure 6.

The 4 ports of the network interface card from Kontron constitute a stand-alone switch, thus these can also be used to either connect different devices or to forward a packet from one port to another.

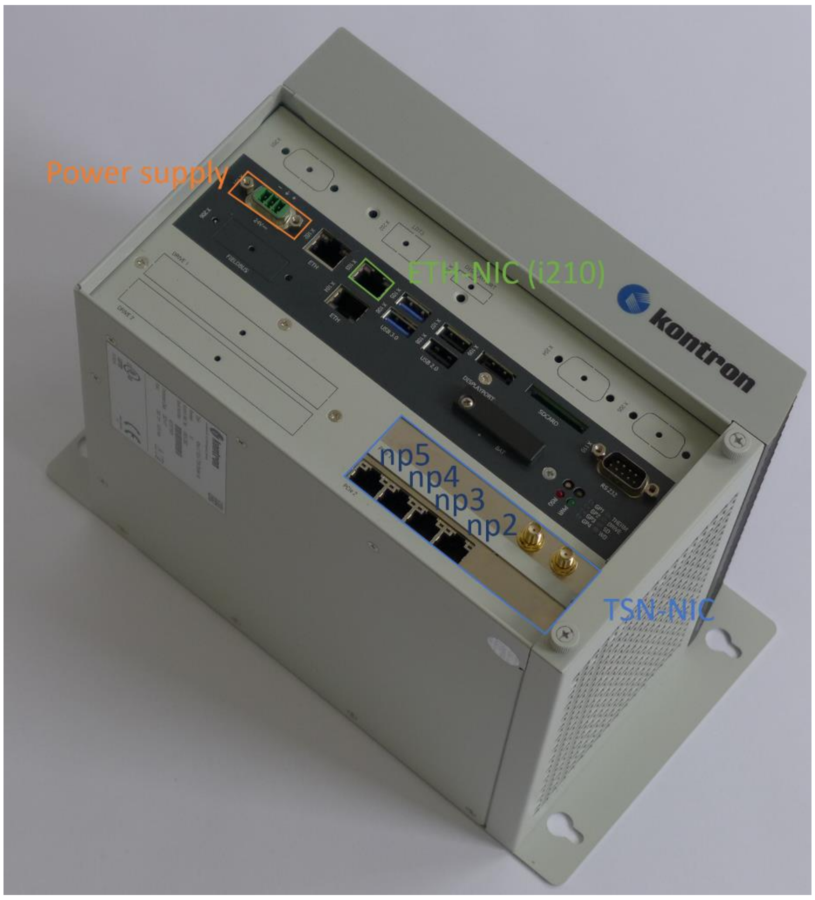


Figure 6: KBox C-102-2 with TSN interface card (TSN-NIC). The ports of the TSN-NIC as well as the other connections used for the setup are annotated.

To test the TSN functionality according to Task 1 (chapter 2.1.1), the subsequently described tests are performed with the two KBox computers and the latency as well as the jitter of the scheduled data flow is measured. During the tests the following data load was produced:

- One-way scheduled data.
- Additionally, one-way best effort data with a high volume is sent.

In a first step the best effort traffic was generated by the same computers also generating the scheduled traffic, and not from separate computer entities as defined in Task 1 (chapter 2.1.1). This is how the test-kit is built up and provides a good insight into TSN functionalities. In a second step, the best effort traffic generation is generated separately.

For the tests to be successful, the computer's hardware clocks need to be synchronised. This is done via the gPTP protocol and is supported over TSN (IEEE 802.1AS generalized Precision Time Protocol).

In TSN, the traffic can be scheduled according to IEEE 802.Qbv, using a time aware shaper. With such a configuration in place, a switch can give exclusive use of the ethernet transmission medium to a defined priority class (or set of priority classes). By creating adequate schedules, transmissions can thus be made deterministically, in the sense that an upper limit to the latency can be defined and that jitter can be controlled.

The computers are set up according to Figure 7. A photo of the setup can be seen in Figure 8. The traffic is generated at PC 1 and received at PC 2. At the receiving PC 2, the traffic is routed over the TSN NIC that is used as a switch, to another non-TSN i210-interface. This interface is used as it supports the hardware timestamping used by the application of the starter kit to determine the latency and jitter of the communication. The VLAN IDs and priority code points (PCP) configured for the different traffic types are also shown in Figure 7.

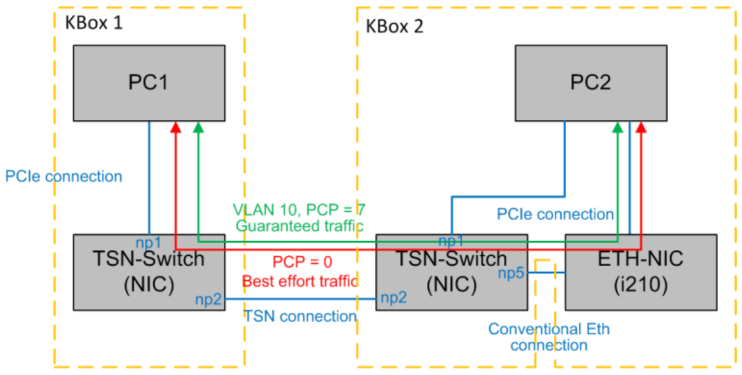


Figure 7: Connections of the computers, basic setup.

The traffic is routed via the TSN-NIC used as a switch to another Ethernet port of the receiving PC. This is done as the Ethernet port can handle hardware timestamping so that the latency of the connection can be measured without introducing time delays caused by the receiving application.

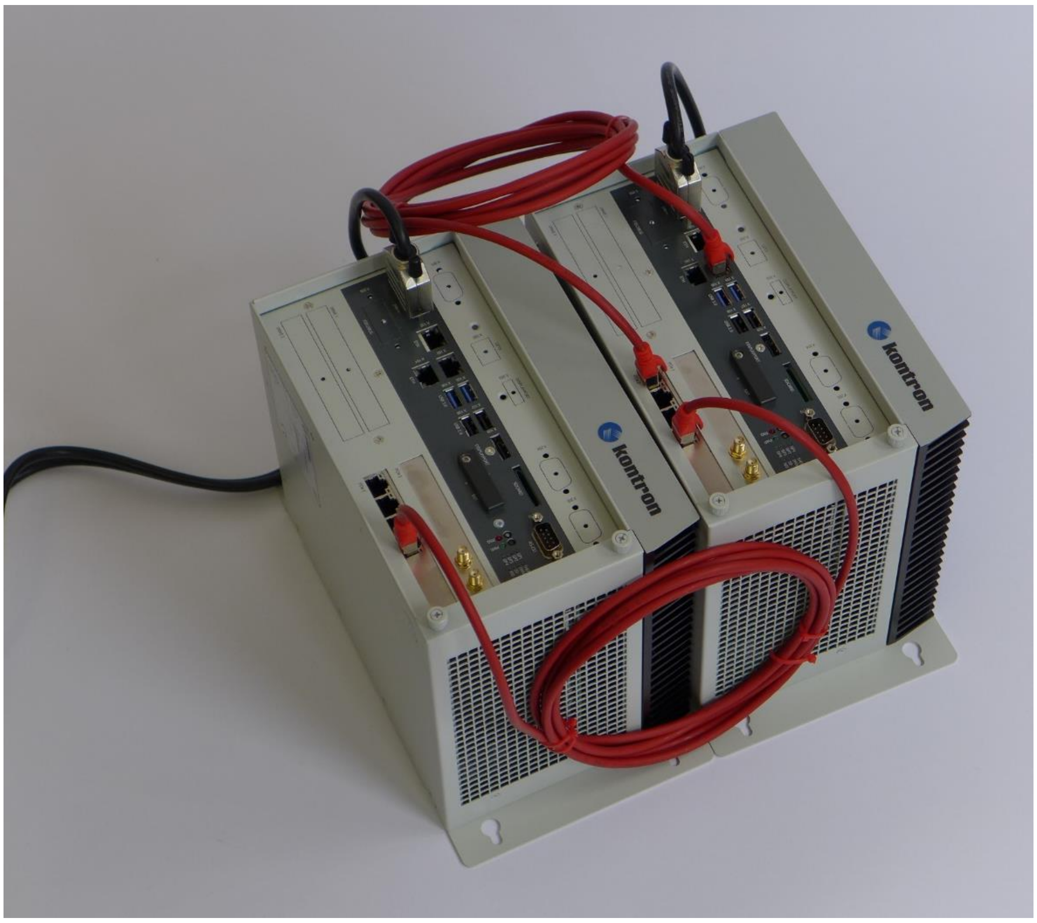


Figure 8: Photo of the test setup shown in Figure 7. (See also Figure 6 for details on the interfaces)

To measure the latency and jitter of the connection the netlatency toolset (written by Kontron) was used. The scheduled traffic is generated using the nl-tx script which periodically sends a message. The packets are received by the nl-rx script which receives the messages, extracts the timestamps of the message acquired along the way. Following timestamps are accessible and shown schematically in Figure 9:

- interval-start: start of the interval. The interval starts with the beginning of a TSN cycle.
- tx-wakeup: wake-up of the nl-tx program which prepares the message to be sent.
- tx-program: timestamp of the moment the nl-tx program calls the send function.
- tx-kernel-netsched: Linux kernel timestamp SOF_TIMESTAMPING_TX_SCHED timestamp prior to entering the packet scheduler in the Linux kernel.
- tx-kernel-hardware: Linux kernel timestamp SOF_TIMESTAMPING_TX_HARDWARE, timestamp generated by the network adapter when sending the message (prior to switch entity of TSN NIC)
- rx-hardware: Linux kernel timestamp SOF_TIMESTAMPING_RX_HARDWARE, timestamp generated by the receiving network adapter
- rx-program: timestamp when the packet is handled in the nl-rx program.

The nl-rx program outputs a file containing these timestamps for each message it received. This log file can then be analysed to extract characteristic information about the connection. For more information on the kernel timestamps, see the documentation of the Linux kernel[3].

The latency is measured by the difference in time between rx-hardware and interval-start, thus measuring the time after the beginning of the cycle when the network interface of the receiving computer receives the message. The jitter is defined by the difference between the latency of one message and the mean latency.

Latency = rx-hardware – interval-start

[3] https://www.kernel.org/doc/Documentation/networking/timestamping.txt

Jitter = Latency – mean(Latency)

The application latency is measured by the difference in time between tx-program and interval-start, thus measuring the latency within the application nl-tx.

RT Application Latency = tx-program – interval-start

If the Linux system uses the real time pre-emption patch, it is expected that this value is rather low and similar between the different messages. This latency can show if there are problems with the real-time patch of the Linux system but should not affect the network as long as the application can send the package before it is scheduled to leave on the TSN port.

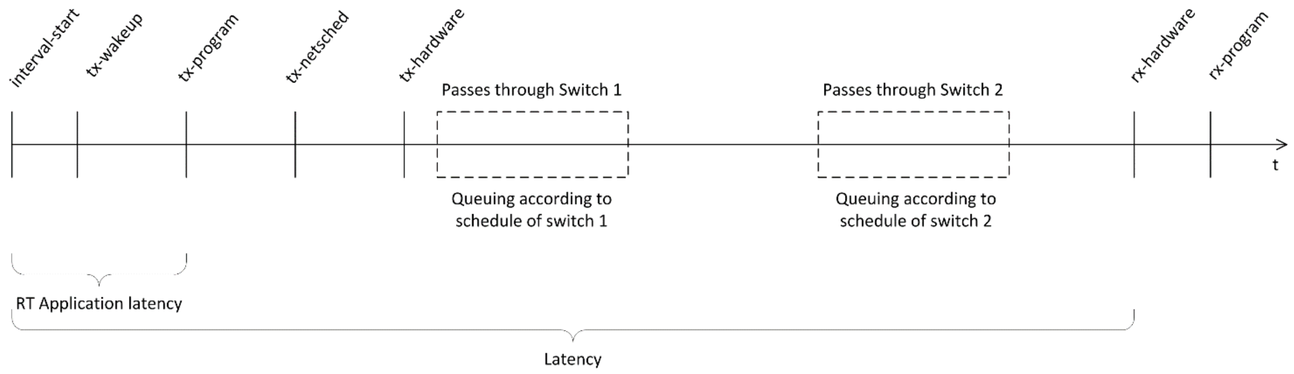


Figure 9: Message path, with time-stamp events. Time differences are not to scale.

Before running the test programs, the switches need to be configured. This configuration will define the schedule according to IEEE 801.2Qbv and thus the performance of the network. The configuration is defined in the Sched_Config file. The configuration is loaded using the tsntool toolset. For the demonstration, Kontron provided scripts that load the configuration onto its own network interface card.

The used schedule is shown below:

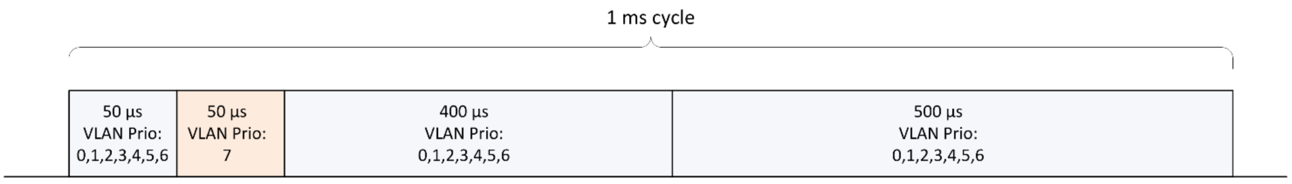

1 ms cycle
50 µs VLAN Prio: 0,1,2,3,4,5,6
50 µs VLAN Prio: 7
400 µs VLAN Prio: 0,1,2,3,4,5,6
500 µs VLAN Prio: 0,1,2,3,4,5,6

Figure 10: Schedule example used to execute the test cases. The last 900 µs of the schedule are configured in two slices as there is a maximum slice length that needs to be respected for the definition of the schedule configuration.

The scheduled data is sent with VLAN priority 7 and will be sent between 50 µs and 100 µs after the beginning of the cycle. The rest of the time, best effort data which is configured to use the VLAN priority 0 will be sent. This guarantees a minimum bandwidth for the scheduled traffic as well as it determines when the scheduled traffic will be sent out. The beginning of the cycle triggers the application which will then send out the scheduled data somewhere between the beginning of the cycle and 50 µs (usually much faster as the application latency was usually around 5 µs during our tests). The message will then be queued until the VLAN priority 7 window is opened at 50 µs. This means that the message will take longer for being sent than if it would have been sent right away. On the other hand, it will result in very stable latencies of 50 µs plus the time it takes the message to get through the network. As the time the message takes to get through the network is quite stable, the jitter of the transmission will be very small.

The schedule also guarantees a certain bandwidth for non-scheduled traffic as the scheduled traffic, with highest priority, will only be sent during the allotted time window.

3.1.2 Time synchronization between application and network

To adjust the application data sending with the network cycles it is not only necessary that the application knows the network cycle length but also when a given network cycle starts. Otherwise, latencies lower than a cycle length cannot be guaranteed. Hereinafter we explain how the netlatency tool nl-tx proceeds to synchronize with the network cycle, illustrating the different mechanisms that can be used to perform this synchronization.

Before going into the details of the synchronization of the application and schedules, it is necessary to introduce the relevant clocks of the KBox. Relevant is the PTP hardware clock (PHC), situated physically on the network interface card (NIC). It is controlled over the NIC driver by the ptp4l script which reads the PTP messages and adjusts the PHC according to the gPTP protocol to be synchronized with the PTP Master. This is done for example by changing the frequency of the clock. The second relevant clock is the system clock. This is the clock that the applications will use to know the time and to timely adjust their functionality. The POSIX clock CLOCK_REALTIME is used by the applications of the test setup as the system clock. The system clock is synchronized to the PHC by the phy2sys script also from the linuxptp package.

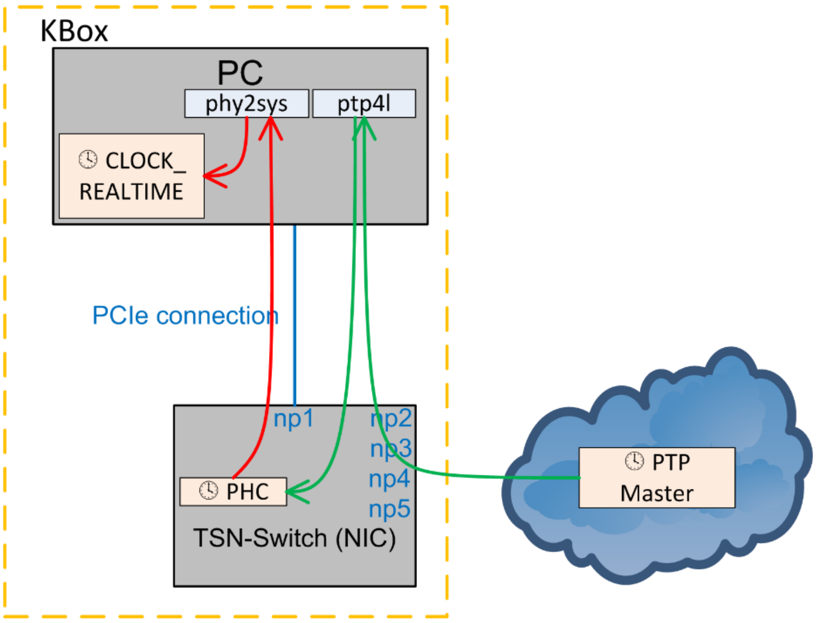


Figure 11: Clock synchronization on the KBox. The ptp4l and phy2sys scripts of the linuxptp package synchronize the PTP hardware clock (PHC) of the NIC and the system clock (CLOCK_REALTIME) to the PTP Master clock which is supposed to be in the network.

When a device (switch or end-device) is configured for scheduling according to IEEE 802.1Qbv, in addition to the cycle time, schedule timing and gate status a base time is given. The device will then start a cycle at

CycleStart = BaseTime + N*CycleTime

where N is an integer. The base time is expressed in the PTP timescale as the number of seconds, nanoseconds and fractional nanoseconds that have elapsed since 1 January 1970 00:00:00 TAI and is a variable that can be freely set during the configuration of the network device. Thus, the beginning of each cycle is defined by the configuration. The clock of the network device remains synchronized with the network through the gPTP protocol according to IEEE 802.1AS. The clock synchronization ensures that devices being configured with the same base and cycle time start their cycles synchronously.

For the application to synchronize to the network cycles, it needs to be able to access the time of the network interface card (synchronized with the network) or another clock that is synchronized with the clock of the network interface card and thus an extension of the network. As explained above the system clock of the KBox is synchronized to the network via the ptp4l and phc2sys packages and can thus be used by the application to time itself with the configured schedule.

To do this, the application must be configured with the same cycle time and base time as the relevant switch or NIC. If that is the case, it can calculate the next cycle start and act accordingly.

The netlatency tool takes the shortcut of supposing that the base time is 1 January 1970 00:00:00 TAI. It thus only needs to be configured with the cycle time in order to synchronize itself with the cycles of the network traffic. Moreover it uses a simplified timer implementation that restricts the possible cycle times. Entering a cycle time that is a non-integer number of milliseconds results in an error of the netlatency tool program.

3.1.3 Tests with IEEE 802.1Qbv

Figure 12 shows the results for a connection with only scheduled traffic. Unsurprisingly the network performs well with low jitter of less than 1 µs. The overall latency is at 52 µs to 53 µs. The results were produced with 10'000 messages sent every 1 ms.

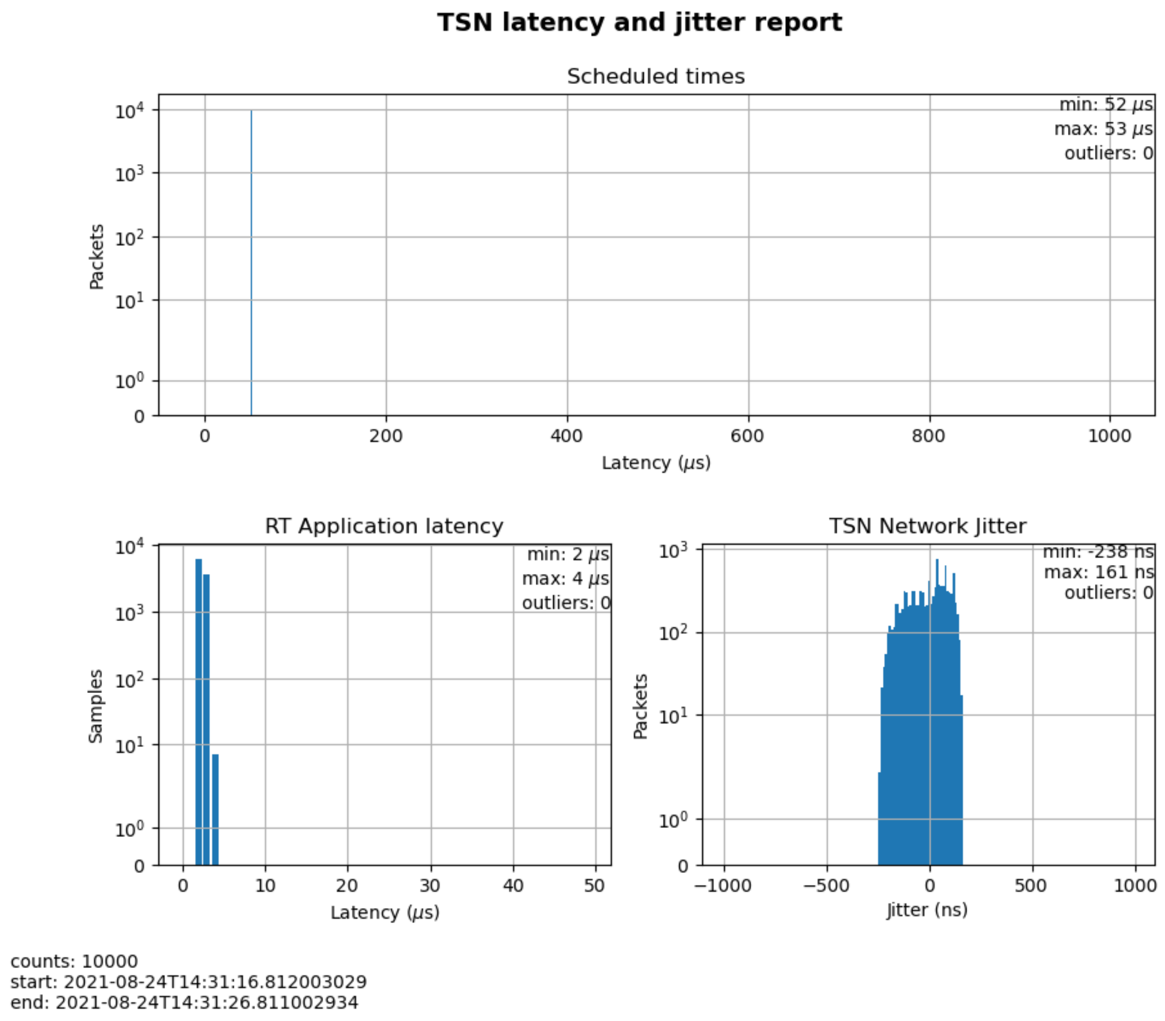


Figure 12: Histograms of latency, application latency and network jitter for scheduled traffic alone.

Figure 13 shows the results for a connection with the same scheduled traffic as before, but with high volume best effort traffic sent alongside. The results were produced with 10'000 scheduled messages sent every 1 ms. Only the scheduled messages are considered as they are relevant to determine the performance of the network. The application latency went up from below 5 µs to below 12 µs, probably due to the CPU cores being also busy generating the best effort traffic. However, the application times are, thanks to the real time patch, still quite small. The network performance remains almost unchanged, with a mean overall latency for scheduled messages of 52 µs to 53 µs and similarly small jitter.

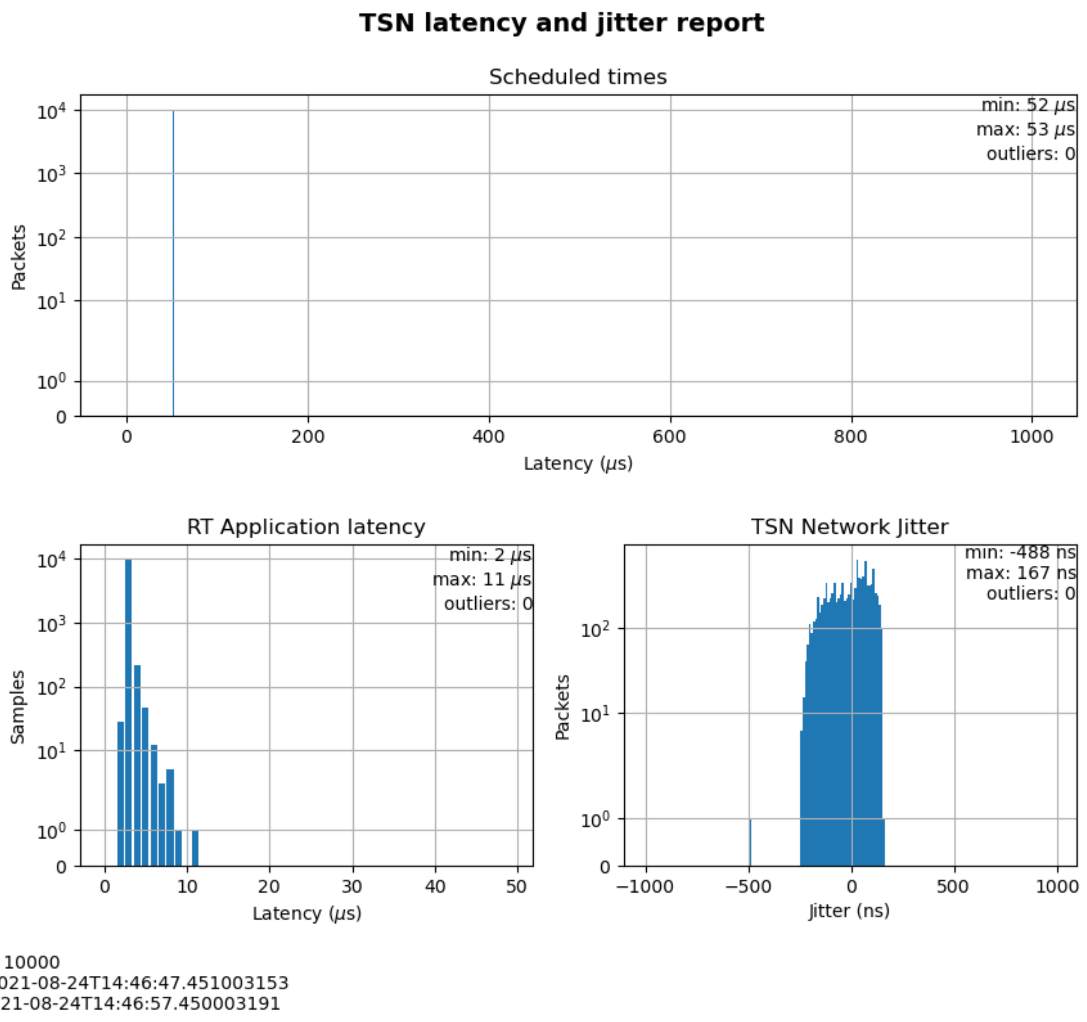
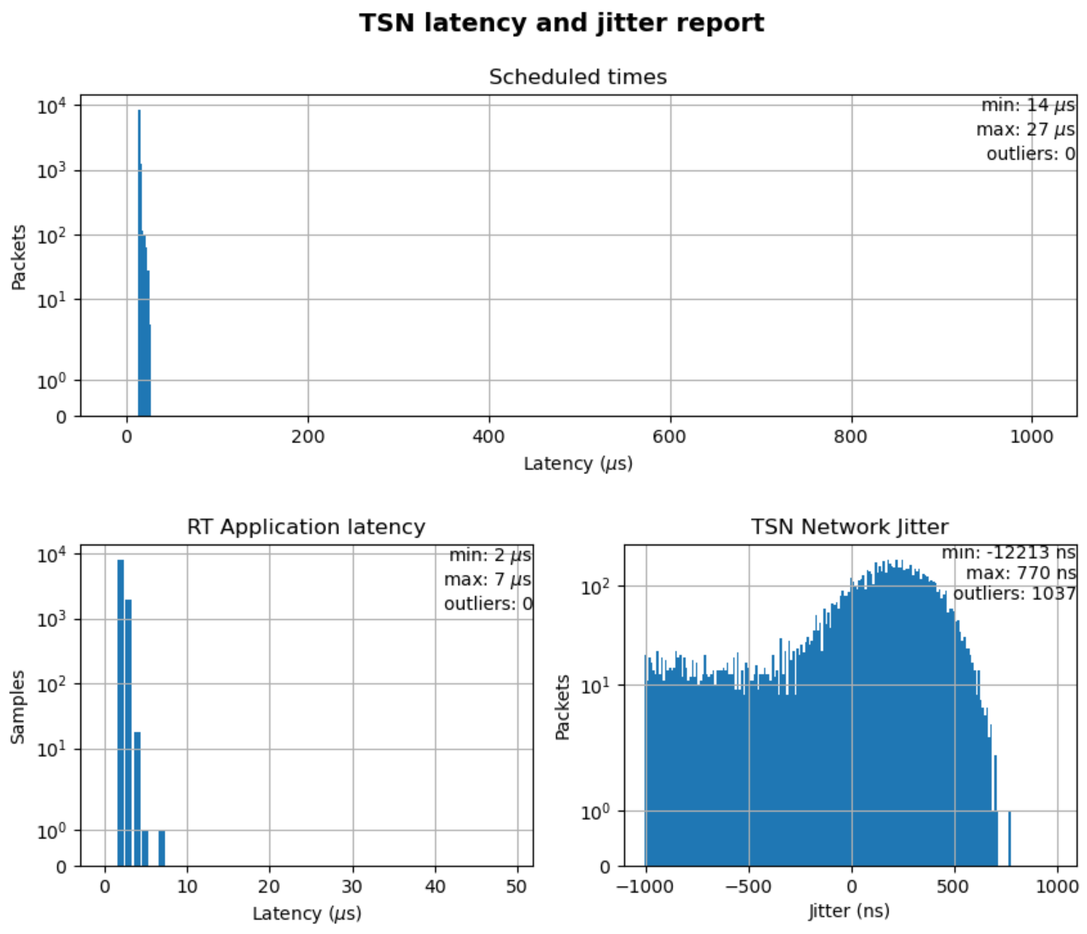


Figure 13: Histograms of latency, application latency and network jitter for scheduled traffic alongside high-volume best effort traffic.

To compare these results with the non-scheduled case, the priority of the previously scheduled traffic was changed from 7 to 2, so that the scheduled traffic is intermixed with the best effort traffic (VLAN priority 0), but still keeps a higher priority than the best effort traffic. The traffic is then routed using usual priority techniques. This test was done in both cases, first without any best effort traffic, then with high volume best effort traffic present.

The results of the first case, without best effort traffic, are shown in Figure 14. We see that the overall latency is reduced compared to the scheduled case (see Figure 12). This is because the packages are not retained by the network card until the gates are opened at 50 µs according to the schedule of Figure 10 but are directly sent onto the network when ready. The latency is thus greatly reduced and lies between 14 and 27 µs. However, this means that part of the jitter is due to the spread in the application latency which is then transferred onto the network. The network however adds jitter as the spread in latency is larger than the one arising from the application.

The results of the second case, with high volume of best effort traffic alongside are shown in Figure 15. First and most importantly we notice that in this case adding the high-volume best effort traffic impacts the network for the VLAN priority 2 traffic where the latency is measured. In the previous case, with the TSN scheduling, one could see that the scheduled traffic was not affected by adding the best effort traffic. Without the TSN scheduling the overall latencies are increased compared to the previous case and the jitter is worsened, for the network as well as for the application. Most of the jitter points lie outside of the plotted range of ±1000 ns.



```
counts: 10000
start: 2021-08-25T13:38:54.418002928
end: 2021-08-25T13:39:04.417003012
```

Figure 14: Histograms of latency, application latency and network jitter for scheduled traffic with VLAN priority 2 without best effort traffic.

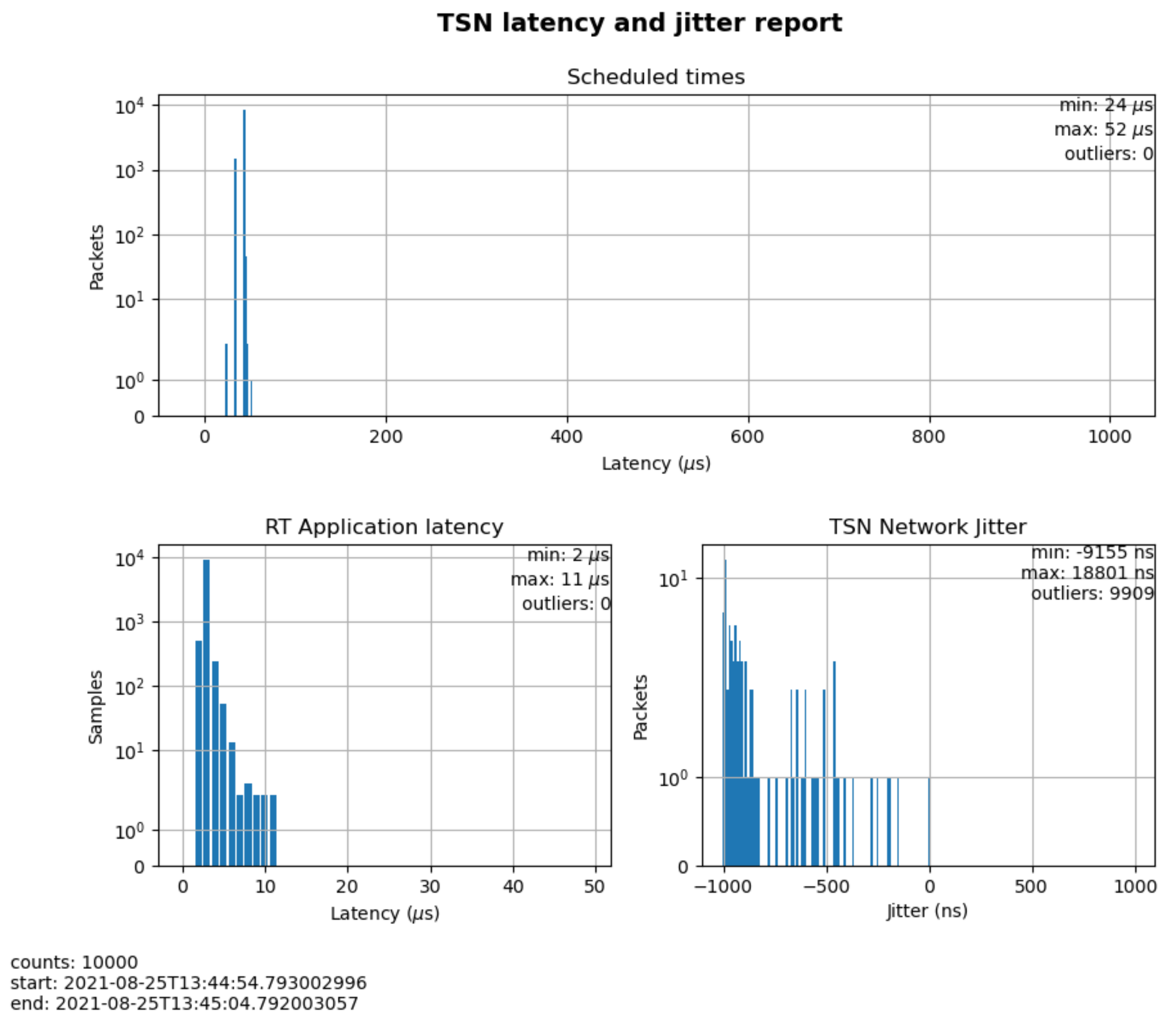


Figure 15: Histograms of latency, application latency and network jitter for scheduled traffic with VLAN priority 2 alongside high-volume best effort traffic.

Another comparison was done where the scheduled traffic was kept at priority 7 however the schedule was changed to the test schedule 2, as shown in Figure 16. This schedule has the advantage of reducing the overall latency to a minimum while reserving a certain amount of bandwidth for the scheduled traffic and keeping the scheduled traffic separated from the best effort traffic thus avoiding interference on the path. However, in this setup, the jitter at arrival can't be controlled anymore as the jitter from the application will be transferred to the network.

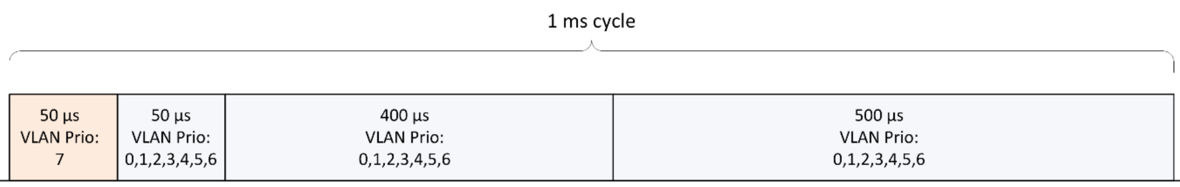

1 ms cycle
50 µs VLAN Prio: 7
50 µs VLAN Prio: 0,1,2,3,4,5,6
400 µs VLAN Prio: 0,1,2,3,4,5,6
500 µs VLAN Prio: 0,1,2,3,4,5,6

Figure 16: Schedule example 2 to send messages with a latency as small as possible.

Figure 17 shows the result without best effort traffic and Figure 18 shows the result with best effort traffic alongside. If no high-volume best effort traffic is present, this situation is similar to the one discussed previously where the priority of the scheduled traffic was reduced, thus changing the time when the traffic can be sent out (see above and compare with Figure 14). With the best effort traffic however, we can observe a different reaction to the one seen previously (Figure 15). As the traffic classes should remain separated with this schedule scheme one would expect the result to remain basically unchanged compared to the case where there is no best effort traffic. However, comparing the results shown in Figure 17 and Figure 18 we do see a degradation of the jitter and the latencies. The degradation is not due to the network but because of the computation load needed on the computer to also generate the high-volume best effort traffic. This results in an increase in the values as well as a spread of the application latencies. The overall latencies spread from about 19 µs to 47 µs which is due to the increase in application latency as well as an expected increase in the latency of the Linux kernel (time between tx-program and tx-kernel-hardware). The latter, however, is not a consequence for the poor performance of the TSN Network but due to computation load on the computer and one could expect a better result if the CPU was not tasked to fully load the ethernet connection with high-volume best effort traffic.

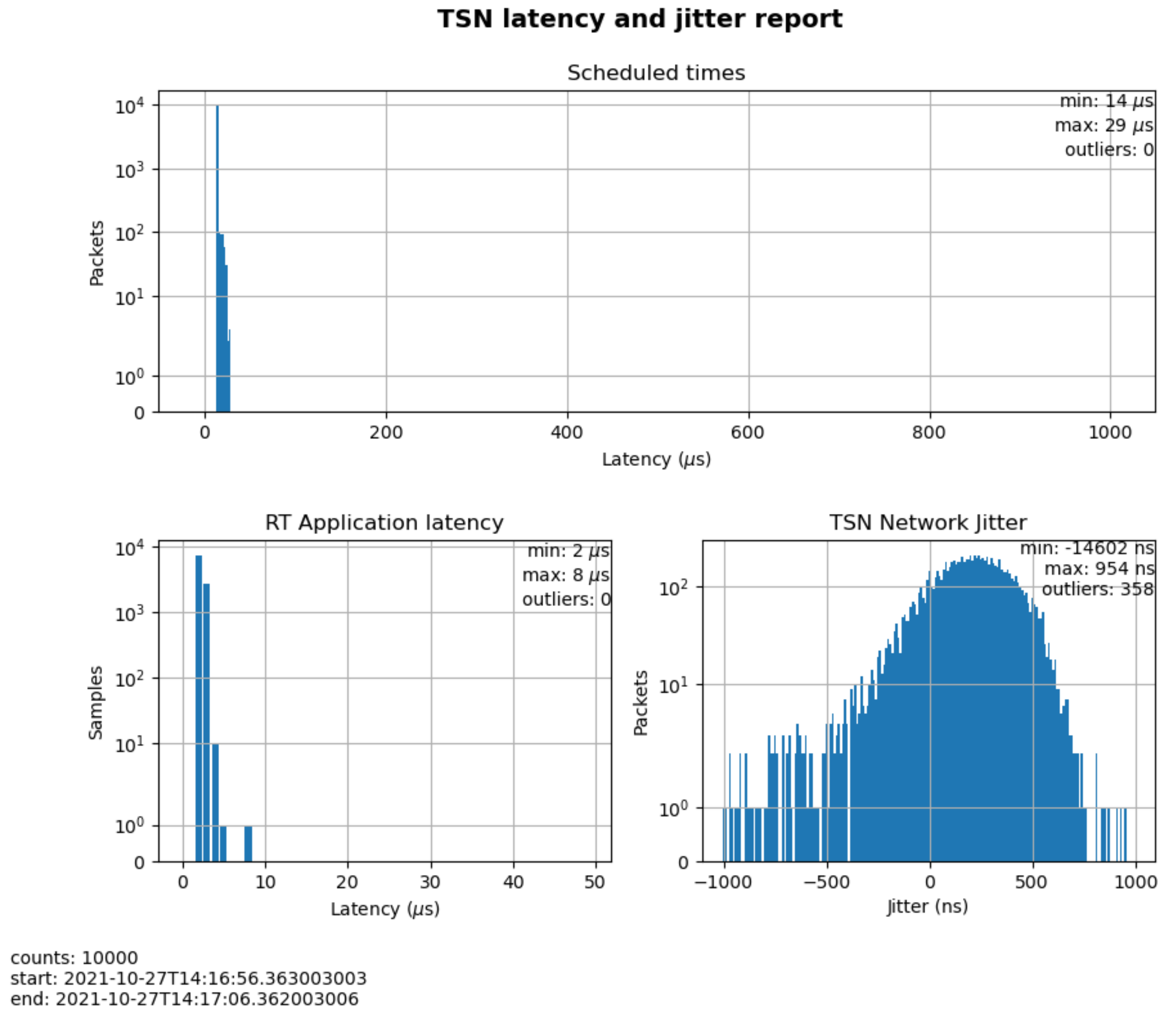


Figure 17: Histograms of latency, application latency and network jitter for traffic with VLAN priority 7 without best effort traffic. Schedule 2 from Figure 16 was used.

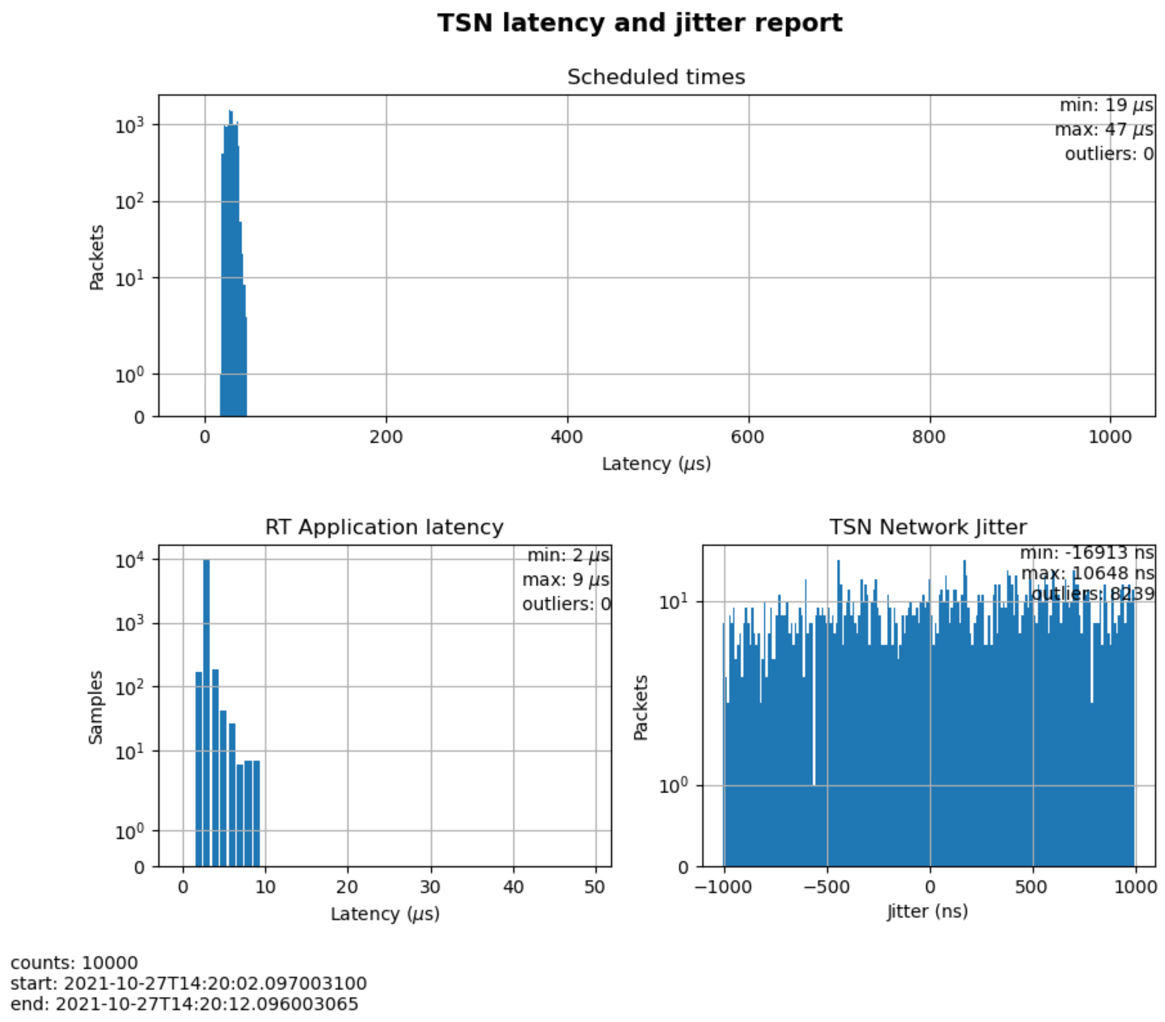


Figure 18: Histograms of latency, application latency and network jitter for traffic with VLAN priority 7 alongside high-volume of best effort traffic. Schedule 2 from Figure 16 was used.

3.2 Best effort traffic generated externally

3.2.1 Tests with IEEE 802.1Qbv

Looking at the results in section 3.1 we see that generating the bulk traffic in high volume next to the scheduled traffic can have an impact on the latency at application level and thus affect the results even though the network performed normally. As a consequence, in the next step we separated the best effort traffic generation from the scheduled traffic generation by adding two computers to the network. The computers were Raspberry Pi 4 computer (RPi) that generated traffic between them using iperf3. Iperf3 is a tool for network performance measurements and generates traffic between a server and a client while measuring the throughput speed. To set this up, the two RPis were connected to the two TSN-NIC which act as switches. They are thus connected, as far as the network is concerned, independently from the KBox computers. There is only one connection between the TSN-NIC which must carry the scheduled traffic as well as the best effort traffic and presents the bottleneck of the network under test. The setup is shown in Figure 19 schematically, a photo can be seen in Figure 20.

During this test, the same schedule as in Figure 10 was configured onto the TSN-NIC. The connection between the two RPis was set up in a separate VLAN and with a priority (PCP) of 1 acting thus in the same way as the previous best effort traffic for the two schedules considered (Figure 10 and Figure 16).

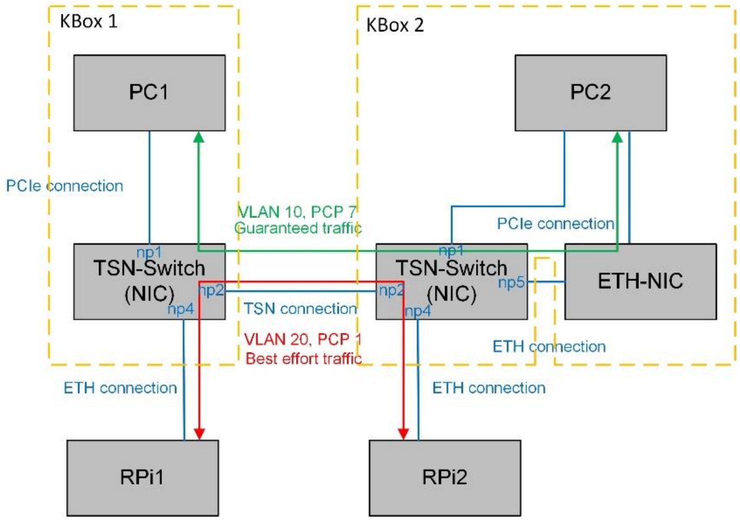


Figure 19: Connections of the computers, setup with dedicated load generation computers.

The scheduled traffic is routed like before (see Figure 7) via the TSN-NIC used as a switch to another Ethernet port of the receiving PC. This is done as the Ethernet port can handle hardware timestamping so that the latency of the connection can be measured without introducing time delays caused by the receiving application. The high-volume best effort traffic is generated and received using two RPi computers. They are connected through the TSN-NIC of the KBox computers which act as a switch. Thus, the connection between the two TSN-NIC carries the best effort traffic as well as the scheduled traffic.

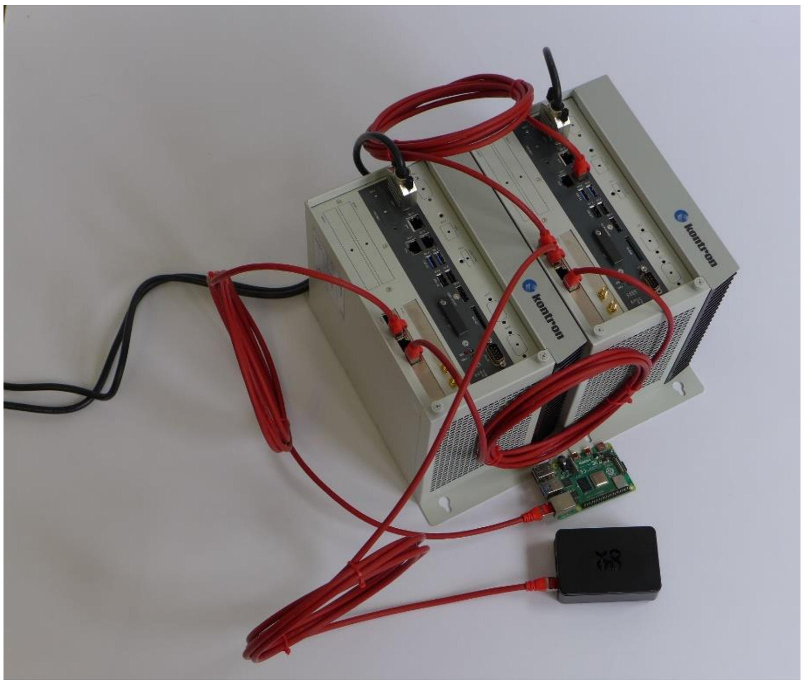


Figure 20: Photo of the test setup shown in Figure 19. (See also Figure 6 for details on the interfaces)

The setup was first tested without the iperf3 traffic between the two RPis enabled giving the baseline to compare to. The result is shown in Figure 21. As expected, the results are comparable to the ones from Figure 12, which is essentially the same situation.

When generating as much traffic as possible using iperf3 on the RPis, the results remain largely unchanged. This situation is shown in Figure 22. The result is very similar in this case albeit the jitter seems to have increased slightly. This could stem from the fact that the firmware of the switch behaves slightly differently in this case as it needs to take care of 3 ports instead of 2 thus additional processes in the FPGA of the switch could lead to additional jitter.

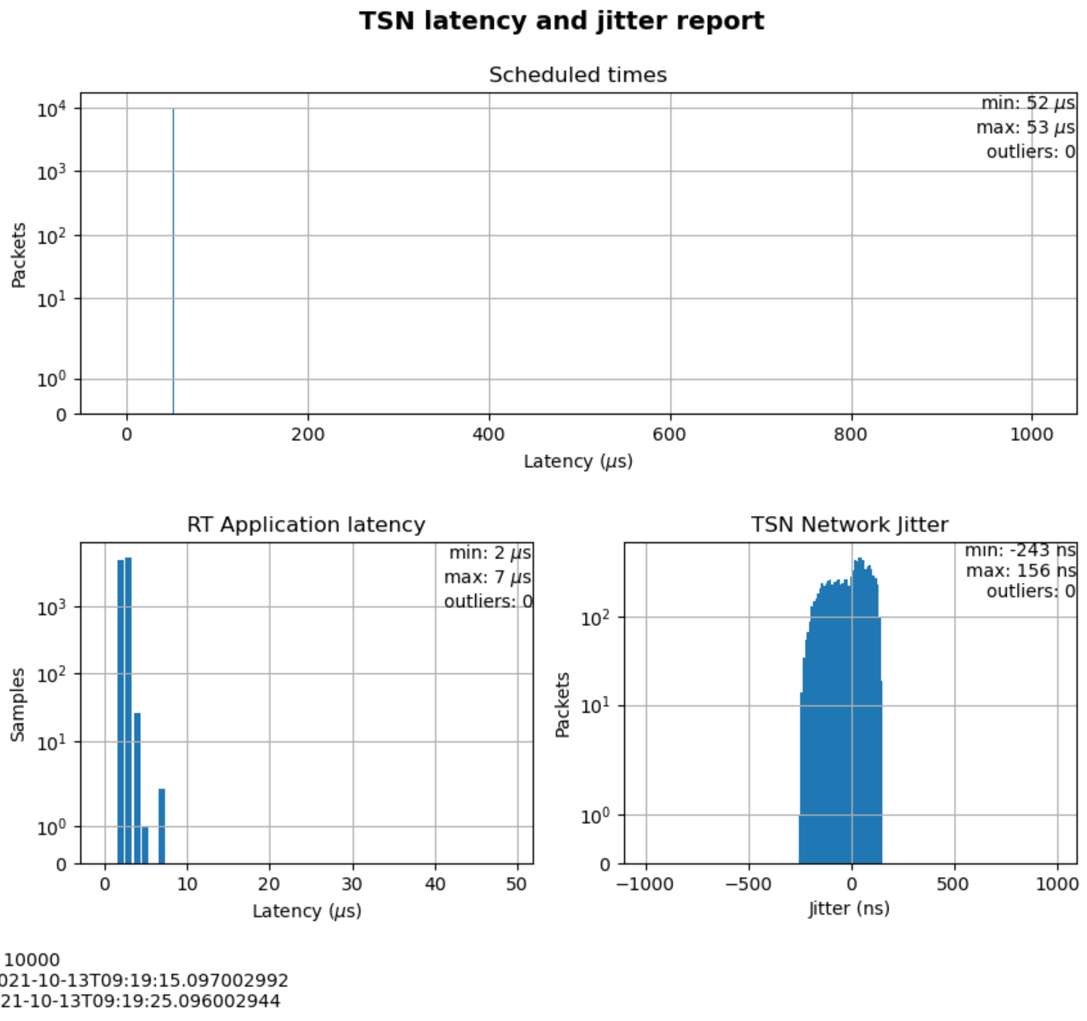


Figure 21: Histograms of latency, application latency and network jitter for scheduled traffic alone with the RPi computers connected but not generating traffic. This is essentially the same case as in Figure 12 and gives comparable results.

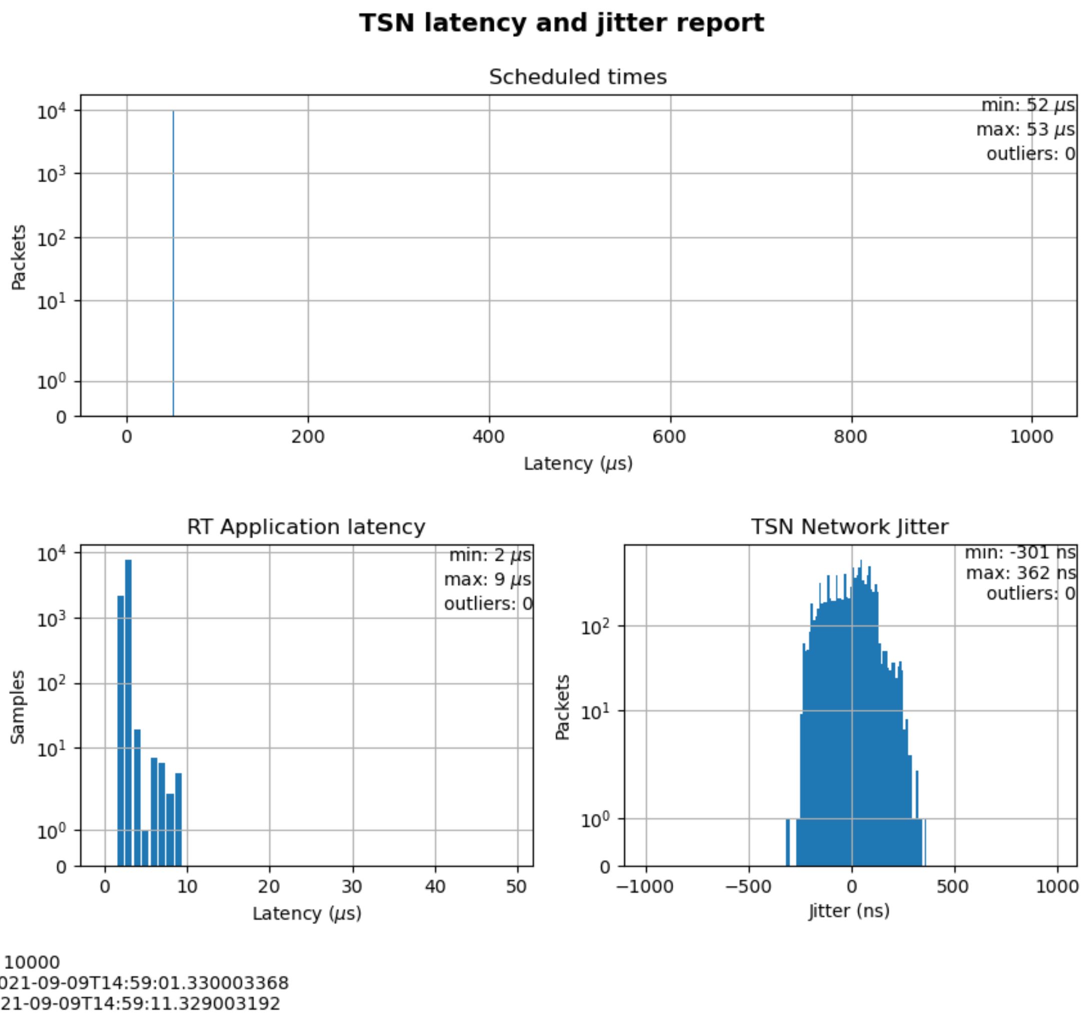


Figure 22: Histograms of latency, application latency and network jitter for scheduled traffic with the RPi computers connected and generating high-volume best effort traffic.

Using schedule 2 (see Figure 16) with the RPis, where the scheduled traffic is sent out as fast as possible during the first 50 µs, we obtain the results shown in Figure 23 (without best effort traffic) and Figure 24 (with best effort traffic alongside). As expected, the result in the case without best effort traffic is very similar to the previous case (Figure 17). When the high-volume best effort traffic generates load however, the result is not affected by the best effort traffic on the network (compare with Figure 18). Previously this was not the case as the KBox computers were also used to generate the best effort traffic. The latter impacted the application side latency of the test and thus altered the result. In this test scenario we see that the network performs well also for this test case as the adverse effect is removed when the generation of the best effort traffic is externalized to the RPis.

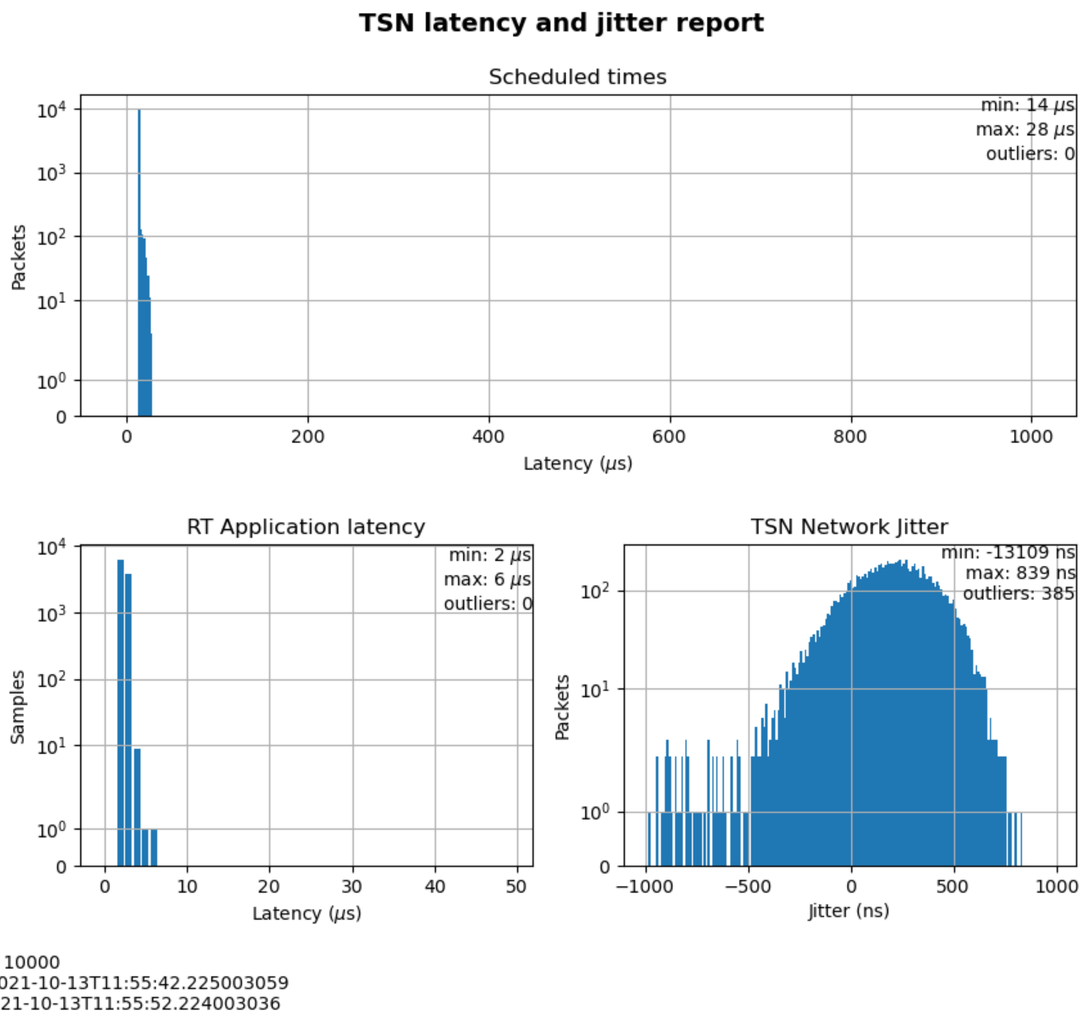


Figure 23: Histograms of latency, application latency and network jitter for traffic with VLAN priority 7 without best effort traffic. Schedule 2 from Figure 16 was used.

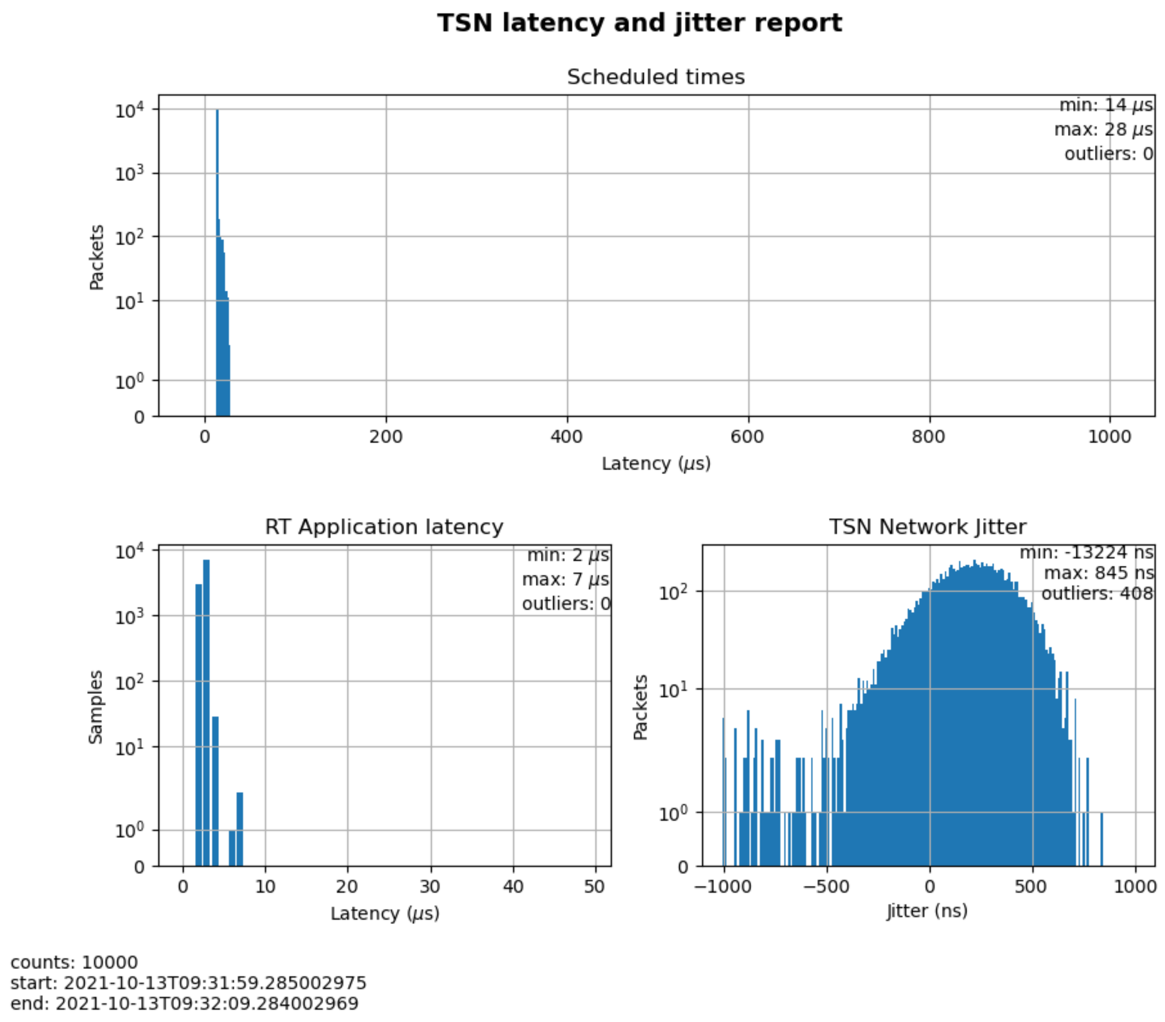


Figure 24: Histograms of latency, application latency and network jitter for traffic with VLAN priority 7 with high-volume best effort traffic between the two RPis. Schedule 2 from Figure 16 was used.

3.2.2 Tests with IEEE 802.1Qbu

Another mechanism for traffic shaping is the preemption of non-priority frames as defined by IEEE 802.1Qbu. This mechanism allows a frame that is currently being sent by a network device to be interrupted in order to send a high-priority frame. The interrupted frame is neither lost nor totally retransmitted but can be continued once the high-priority frame is sent. The two parts of the frame are joined again in the next network device. The mechanism works on the MAC-layer (layer 2) of the network and needs to be configured for and supported by the transmitting and receiving port of a given connection between two devices.

IEEE 802.1Qbu only defines two types of traffic, preemptable traffic and express traffic, where express traffic can pre-empt the preemptable traffic. The PCP field of the VLAN header is used to discriminate between the two traffic classes. Each of the 8 priorities is assigned a frame preemption status (with a value of preemptable or express) in the frame preemption status table. Frames with the same preemption status are handled as if no preemption were configured. In the setup discussed here a strict priority scheme is applied by the network devices.

The fragments of a preempted frame need to be at least 64 octet long. Thus, even with preemption enabled an express frame will have to wait usually below 64 octets time, but at worst up to 123 octets time before it can be sent. Without preemption enabled however an express frame needs to wait for the entire frame which can be in the worst case 1522 octets time, the typical value depends on the kind of traffic sent over the network. For gigabit ethernet connections (1 Gbit/s) latencies can thus be reduced by up to 12.2 µs per connection between two devices. For fast ethernet connections (100 Mbit/s) this time increases to 122 µs per connection between two devices.

In train networks where the use of fast ethernet connections is still common, this mechanism can produce a significant reduction in network jitter and latency. It works best however, if the volume of express traffic in the system is expected to be small.

Using frame preemption in conjunction with traffic shaping such as the scheduling defined in IEEE 802.1Qbv, the bandwidth of the network can be used more effectively than with scheduling or preemption in isolation. Usually using a schedule, a guard band is present to stop the transmission of frames which could extend into the next time slot and are not included in the next time slots open gates. Thus, each transition affects the bandwidth of a certain number of traffic classes.

To test and verify the preemption mechanism, the setup explained in chapter 3.2.1 is taken and IEEE 802.1Qbu is configured for the connection between the two TSN-NIC (np2-np2). For each np2 port the priority PCP 7 was set to express, all other priorities were set to preemptable. Thus, the scheduled traffic from before, that is sent once per cycle from one KBox to the other is now considered as express traffic when IEEE 802.1Qbu is enabled. This traffic will be called high-priority or express traffic in the following paragraphs.

The connection between the KBoxes was tested with and without best effort traffic present as well as with and without preemption enabled on the two np2 ports in order to see the effect of the preemption mechanism on the network. For the tests a dummy schedule was configured that had only one time slot during which every priority could be sent. The scheduling could thus not have any effect on the transmission.

Figure 25 shows the overall latency and jitter between the KBoxes for the base case, where there is no best effort traffic and preemption was disabled. As the dummy schedule sends the high priority traffic throughout the cycle, the traffic is not delayed before being sent to the network. We get a similar situation as in Figure 23. If preemption is enabled at this point, there is no notable effect as there is no preemptable best effort traffic that delays the express messages, see Figure 26.

In the case where best effort traffic between the two RPi computers is enabled and preemption is disabled, the overall latency is spread to higher values. Due to the best effort traffic, the network becomes saturated. It is thus likely that a high priority frame arrives at the switch while another frame is being transmitted and is delayed. Only the trunk connection is however affected by this. Thus, the delay will be somewhere between 0 µs (no frame is blocking the high priority frame) and 12.2 µs (a best effort frame just started transmission).[4] We expect the additional delay to be uniformly distributed between these two values. This situation is shown in Figure 27. The uniform broadening of the peak can be seen as well as an increase in maximum latency of about 14 µs. The fact that this additional delay is larger than the expected 12.2 µs, can be caused by a longer processing time in the switch, which has to handle a lot more traffic than before, or might be due to other reasons specific to this test.

When enabling IEEE 802.1Qbu again, while maintaining the best effort data transmission, the effect of preemption can finally be recognized. This situation, shown in Figure 28, demonstrates the latencies of the express traffic being improved compared to the previous case (Figure 27) and recovering to a similar performance as the case without best effort traffic (Figure 25). Here the best effort frames can be interrupted, and the express frames are sent through the network without being influenced by the best effort traffic. The small additional delay needed to interrupt the frame and to respect the minimal frame sizes for the frame fragments is not noticeable in this evaluation. However, by inspecting the preemption status output of the driver of the network card, one can see a fragment counter which counts the number of times a packet was fragmented/needed to be merged when sent/received. When the IEEE 802.1Qbu is enabled, this counter increases in the presence of high priority traffic, confirming that the IEEE 802.1Qbu mechanism is indeed acting on the traffic.

During the tests it was observed that about 800 – 850 best effort frames were interrupted per second with an express frame being sent every ms, or in other words, about 80 % - 85 % of the express frames led to the preemption of a best effort frame. This is about what can be expected, as the best-effort traffic has frames of 1522 bytes. As the individual frames cannot be smaller than 64 bytes, only during transmission of 1458 bytes of the 1522 bytes frame, the frame can be interrupted, which limits the number of preempted frames to 96 %. This is the theoretical limit, which is further reduced due to inter-frame time, imperfect loading of the line by the best effort traffic, administrative messages etc. to the 80 % - 85 % rates determined during the tests.

[4] The best effort traffic between the RPi computers is generated by iperf3 in such a way that each frame has the maximum size of 1522 octets.

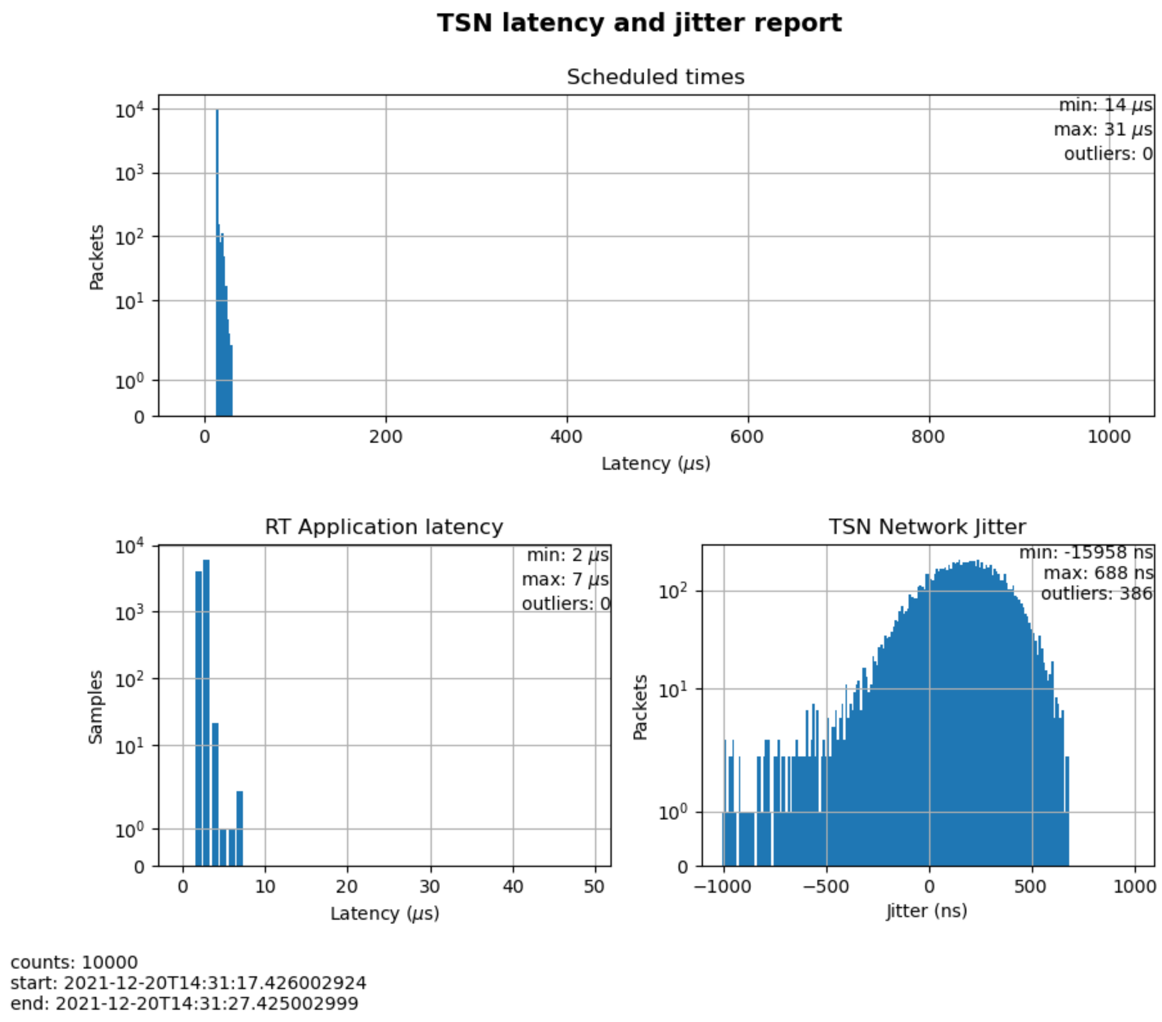


Figure 25: Histograms of latency, application latency and network jitter for high express traffic (PCP 7) alone with the RPi computers connected but not generating traffic and IEEE 802.1Qbu disabled. A dummy schedule where all the priorities can be sent throughout the 1 ms cycle was used.

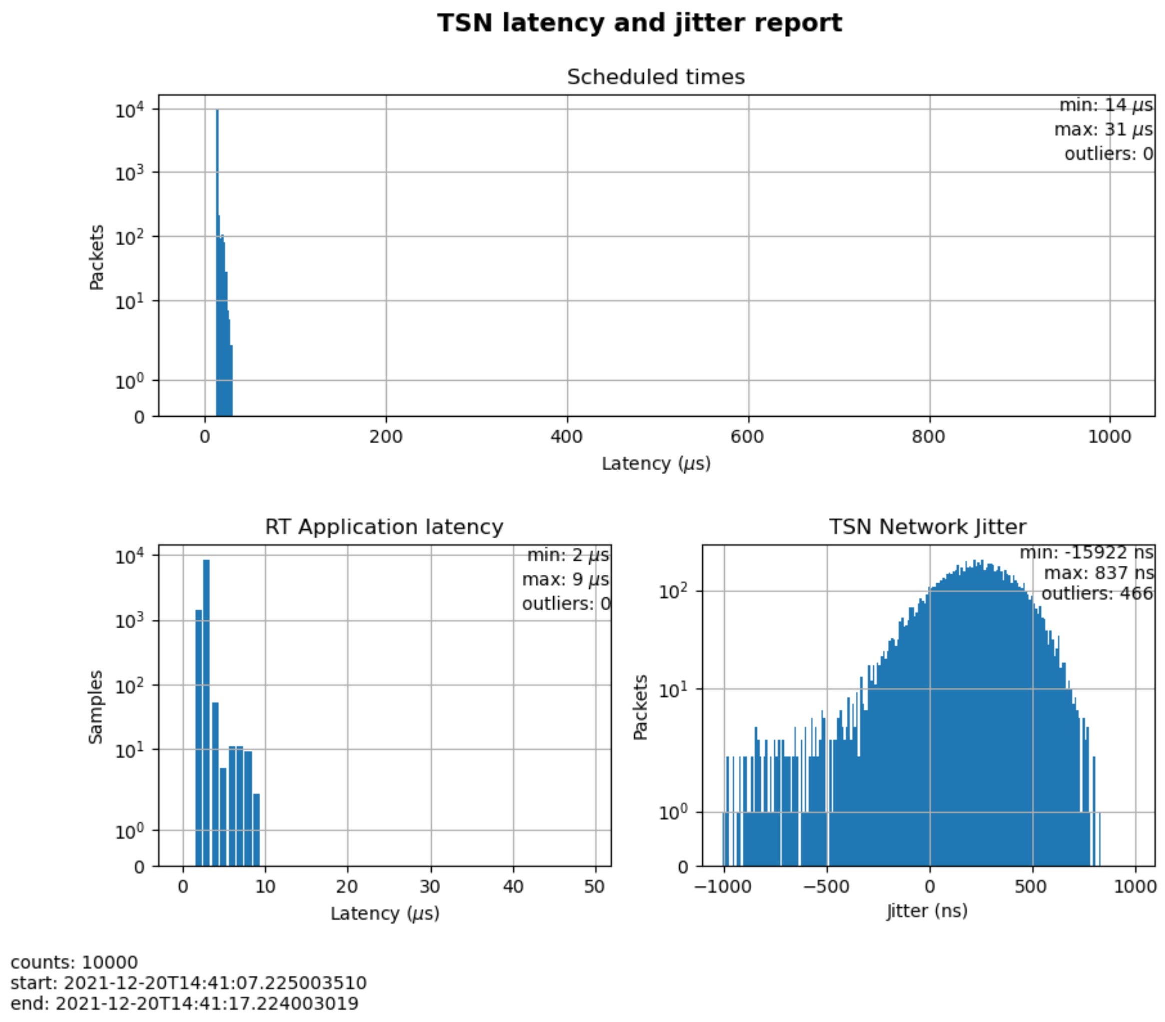


Figure 26: Histograms of latency, application latency and network jitter for high express traffic (PCP 7) alone with the RPi computers connected but not generating traffic and IEEE 802.1Qbu enabled. A dummy schedule where all the priorities can be sent throughout the 1 ms cycle was used.

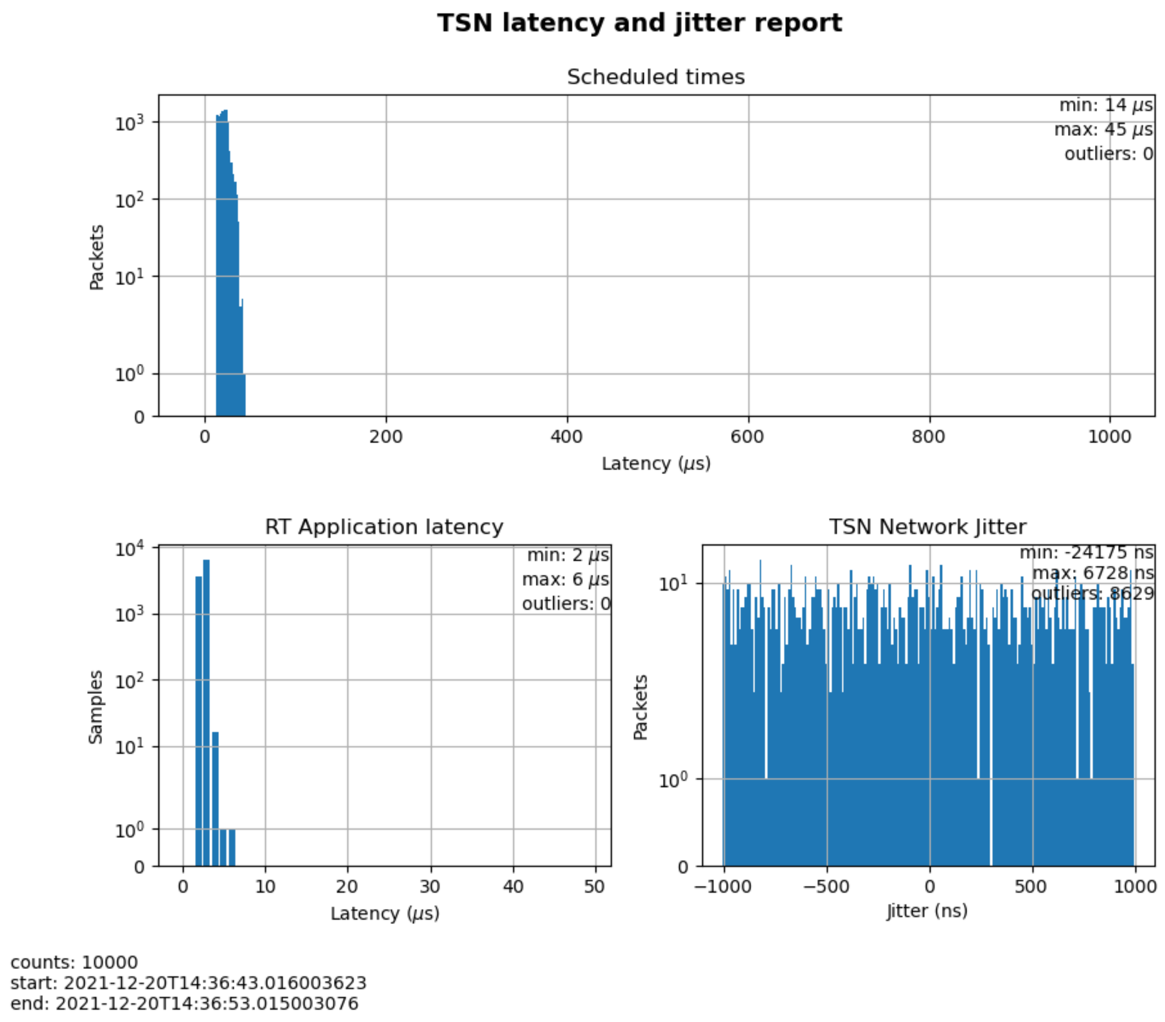


Figure 27: Histograms of latency, application latency and network jitter for high express traffic (PCP 7) with the RPi computers connected and generating traffic and IEEE 802.1Qbu disabled. A dummy schedule where all the priorities can be sent throughout the 1 ms cycle was used.

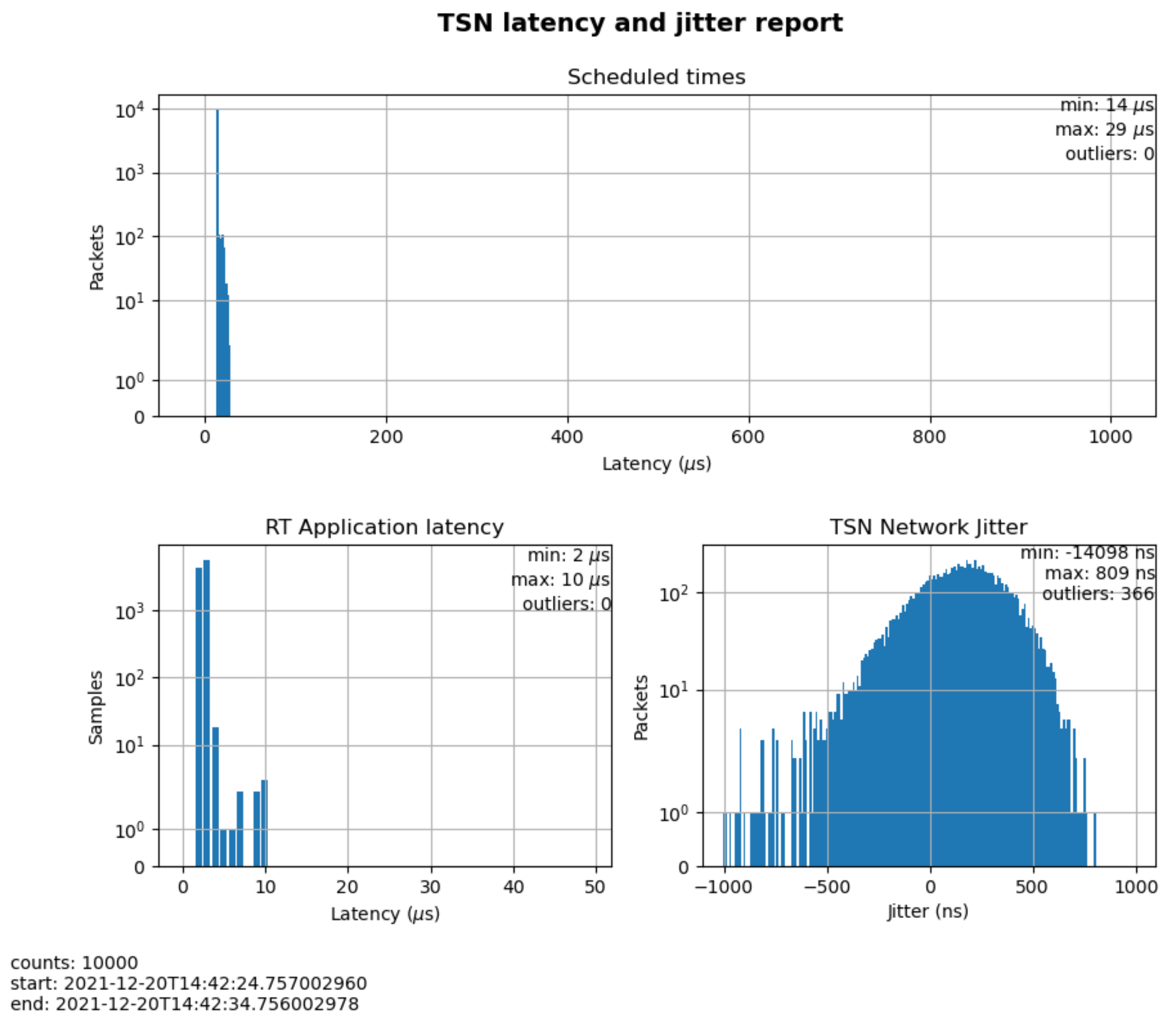


Figure 28: Histograms of latency, application latency and network jitter for high express traffic (PCP 7) with the RPi computers connected and generating traffic and IEEE 802.1Qbu enabled. A dummy schedule where all the priorities can be sent throughout the 1 ms cycle was used.

3.3 Remark Task 1

Here we discuss a situation that occurred several times during the test case with using schedule 2 (Figure 16) and with bulk traffic (as in Figure 18). From time to time a message would take longer than 50 µs to arrive at the second switch and would then be withheld by the schedule, only to be sent out during the next cycle. An example of a test run with at least one of these outliers, having a latency of 1 ms,[5] is shown in Figure 29. Such outliers were not uncommon in this test case as the reserved time for the schedule of 50 µs was close to the upper limit of time needed for a KBox to prepare the message.

Possible delays in the application or in the network should be considered when scheduling the traffic. Ideally an application that is synchronised with the network should prepare the message ahead of the schedule corresponding to it, so that there is no transfer of the application jitter onto the latency of the messages and that the time buffer is sufficient so that one can expect all the messages to be sent within the one cycle. Moreover, it is also necessary to consider added latency due to the network when configuring switches along the message path. For example, the second reached switch would optimally have its schedule for the message shifted by the expected time it takes for the message to reach it from the first switch. In our example the involved time is very short so not taking this into account does not affect the results too much, but in a larger network this should be considered to optimise the bandwidth use of the network. Schedule planning programs

[5] The datapoint corresponding to this message is plotted in the latency graph of Figure 29 at a latency of 0 µs. This is because the latencies are plotted modulo the cycle time (1000 µs) to be able to show all points in the graph. The maximum value shown in the annotation of the graph, however, shows the (rounded) actual value of 1000 µs. As the non-rounded value is just slightly bigger than 1000 µs, taking the modulo gives a value just above 0 µs.

like Slate XNS from TTTech, which will be tested at a later point in time, are programmed to take these shifts into account, thus simplifying the scheduling process.

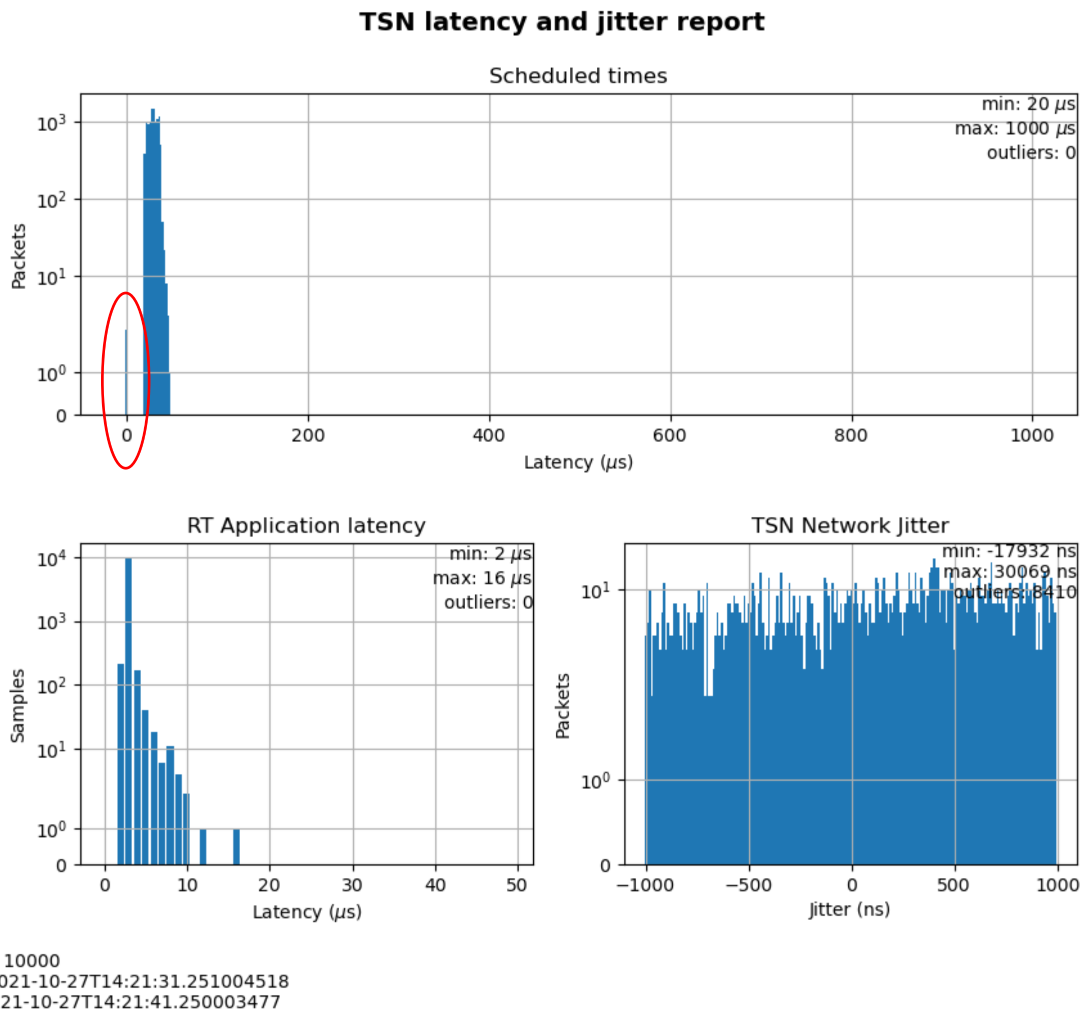


Figure 29: Histograms of latency, application latency and network jitter for traffic with VLAN priority 7 with best effort traffic alongside. Schedule 2 from Figure 16 was used. Two messages did not manage to make it through the network on time with the schedule and were thus withheld until the next cycle (see red mark in latency plot).

3.4 IEEE 802.1CB Frame replication and elimination for reliability (FRER)

3.4.1 Introduction

IEEE 802.1CB also belongs to the TSN set of standards. It can be used to send duplicate copies of each frame over multiple disjoint paths, and provide proactive, seamless redundancy for applications that cannot tolerate packet losses, during the time needed by the network to recover from equipment failures.

The mechanism works by identifying packets according to their MAC destination address and their VLAN. The packets identified to belong to a stream are then duplicated and sent on different ports of specific devices and merged or eliminated at other devices in such a way that if one of the paths fails, the other remains operational and no packet is lost. This is a proactive approach where the streams are constantly duplicated so that in case of failure, there is no downtime for reconfiguration (for example the time needed by the multiple spanning tree protocol MSTP reconfiguration). When duplicating a frame, a sequence number and an R-tag are assigned to each frame. These are also used to recognise a frame that has already passed a point of the network where duplicates are discarded. The means by which the multiple paths are created is not part of the IEEE 802.1CB standard but is the subject of other standards, including IEEE 802.1Q, IEEE 802.1Qca, and IEEE 802.1Qcc.

3.4.2 Tests

Using a reduced setup, the IEEE 802.1CB mechanism could be tested and evaluated. The setup involved just two KBox computers that are connected by two links in a redundant way as shown in Figure 30. The IEEE 802.1CB mechanism needs to be configured to duplicate the high-priority traffic on the links np4-np4 and np5-np5 and to eliminate a copy at receiving side. This however is not sufficient as this configuration only tells the switch to duplicate the frame and specifies the ports. In addition, through the system configuration, we need to make sure that both paths are available and active. Indeed, to avoid the loop created in the network, the usual spanning tree protocol (MSTP IEEE 802.1Q) will disable one of the links for a given VLAN and send all the corresponding traffic over the other link. If a stream of packages, is sent between the two PCs and the active link is interrupted, the MSTP will make sure that the second link is activated, and the stream continued. However, the MSTP protocol takes some time to recalculate the spanning tree and activate the redundant link. During this time packets are lost and cannot be recovered. The configuration, where only MSTP protocol is active while CB mechanism is deactivated, will be designated the ”without CB” in the following sections.

In order to prevent the MSTP to disable one of the links, VLAN 10 is added to a traffic engineered MST instance (TE-MSTID i.e., 4094) for the tests with the IEEE 802.1CB mechanism enabled. The learning for the VLAN 10 is thus disabled and manual/user defined traffic engineering can be performed by setting up static entries in the forwarding database (FDB) for the used destination MAC addresses. The configuration in KBox 1 is defined to forward these messages, with the specified VLANID and destination MAC address, to np4 and np5, at the receiving side (KBox 2) to forward these messages to np1.

By configuring a stream identification function assigned to port np1, a sequence generation function and a sequence encoding function, the duplication of the stream messages can be set up for the sending side (KBox 1).

A stream identification function on ports np4 and np5, a sequence decoding function and stream recovery function needs to be configured on the receiving side (KBox2) for the elimination of the duplicated stream messages.

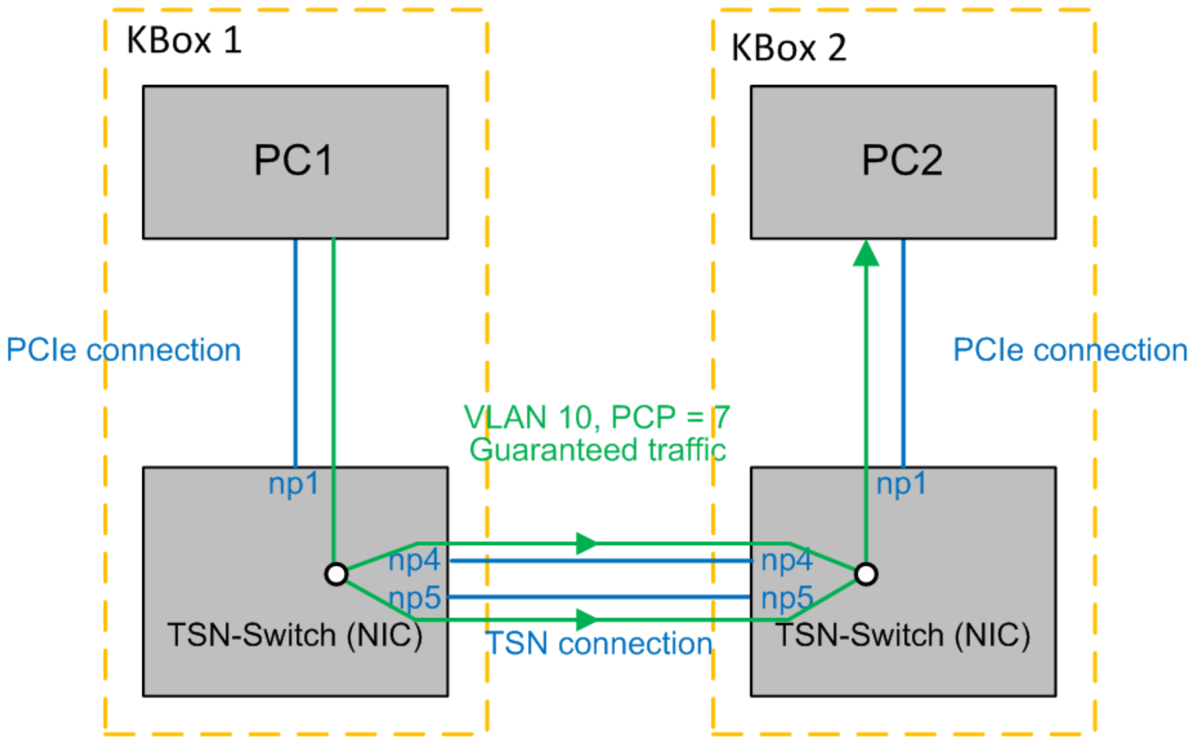


Figure 30: Setup of the computers for the seamless redundancy test.

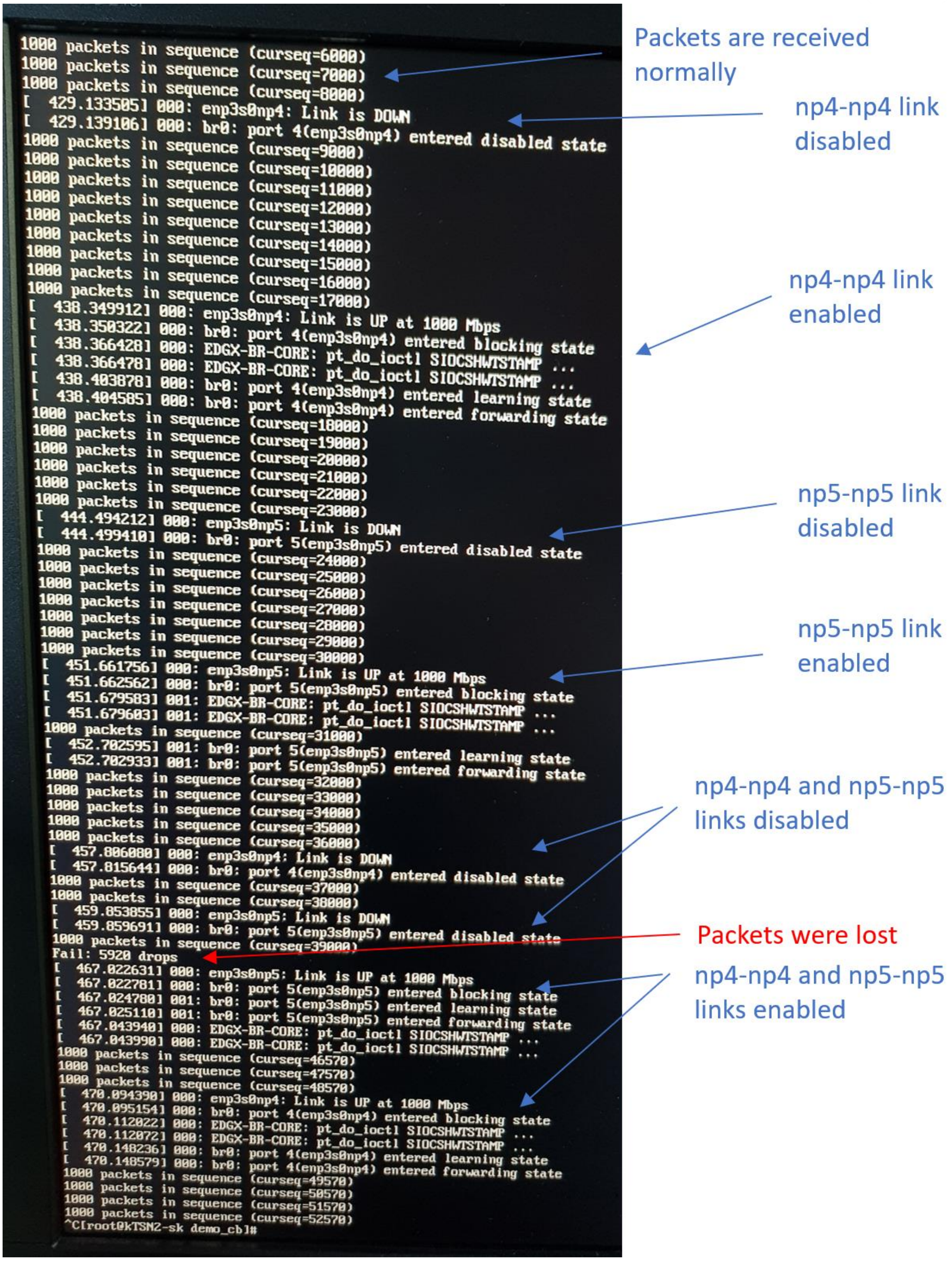
The system was set up once without CB mechanism activated and once with CB configuration active. A test program sends messages with an identifying number (a monotonical number counter) from PC1 to PC2. The reception program checks if the identifying numbers are in sequence and can thus recognize if packets are lost or were received out of order. The test programs for the sending and receiving side are provided by Kontron with the evaluation kit.

The outputs of the test program for a test with CB and without CB are shown in Figure 31 and Figure 32 respectively. To test the behaviour in case of link failure, one or both links between the TSN-Switches were disabled by unplugging the corresponding cable. The different outputs of the program make the different behaviours of the two configurations evident.

With CB enabled, there is no packet lost when the np4-np4 or the np5-np5 link is disconnected, thus showing the seamless redundancy of the setup. If both are disconnected however, the packets are lost as there is no path left between the sender and receiver.

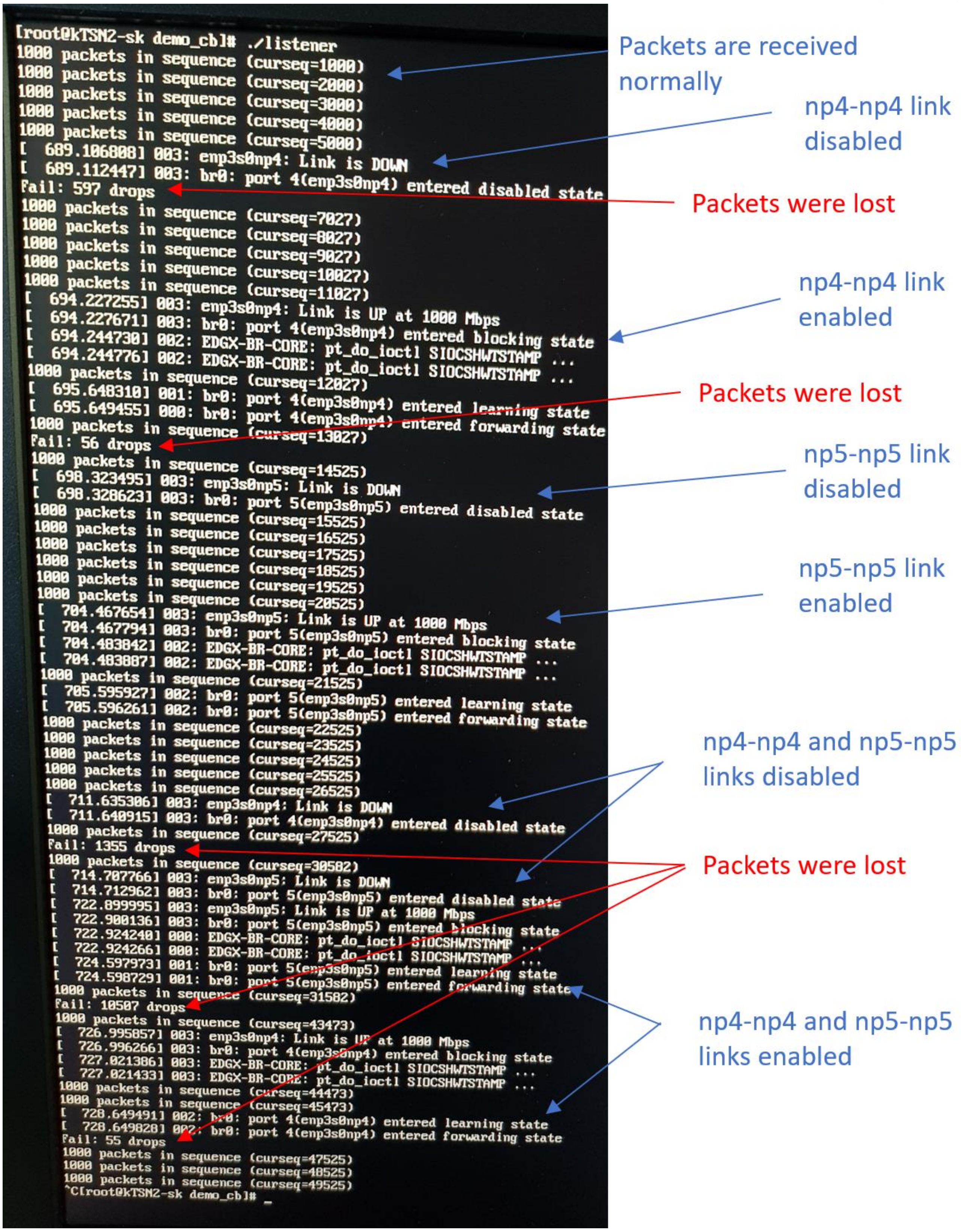
Without CB the story is slightly different. When the np4-np4 link is disabled, a small number of packets is lost as the MSTP reconfigures the tree to use the np5-np5 link. When the np4-np4 link is connected again, there is again a small loss of packets. This is because the MSTP prefers the np4-np4 link. While the np4-np4 link is disconnected, the traffic goes over the np5-np5 link. However, when the np4-np4 link is connected again, the protocol switches the traffic back to this link. In this process a few packets get lost again. When the np5-np5 link is disconnected and reconnected no packets are lost, as it was already disabled by the MSTP. When both links are disconnected there is, of course, a loss of packets as there is no path left between the sender and receiver. The redundancy of this setup manages to ensure the connection between the two PCs, however the switching from one link to another is not done seamlessly. This behaviour can be an issue if the packets sent are time important (cannot be sent again) or time critical (there is no need to send the packet again as the information has deprecated). The behaviour is however not problematic if the packet can be sent a second time, for example if TCP is used for the transmission.

Thus, by configuring the different network devices correctly seamless redundancy can be achieved thanks to IEEE 802.1CB. However, as the example shows, the configuration requires a thorough engineering of the network if end-to-end redundancy should be achieved seamlessly. A simple enable/disable procedure is not sufficient. Nevertheless, the mechanism provides a flexible way to integrate seamless redundancy in networks or parts of a network.



```
1000 packets in sequence (curseq=6000)
1000 packets in sequence (curseq=7000)
1000 packets in sequence (curseq=8000)
[  429.133505] 000: enp3s0np4: Link is DOWN
[  429.139106] 000: br0: port 4(enp3s0np4) entered disabled state
1000 packets in sequence (curseq=9000)
1000 packets in sequence (curseq=10000)
1000 packets in sequence (curseq=11000)
1000 packets in sequence (curseq=12000)
1000 packets in sequence (curseq=13000)
1000 packets in sequence (curseq=14000)
1000 packets in sequence (curseq=15000)
1000 packets in sequence (curseq=16000)
1000 packets in sequence (curseq=17000)
[  438.349912] 000: enp3s0np4: Link is UP at 1000 Mbps
[  438.350322] 000: br0: port 4(enp3s0np4) entered blocking state
[  438.366428] 000: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
[  438.366478] 000: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
[  438.403878] 000: br0: port 4(enp3s0np4) entered learning state
[  438.404585] 000: br0: port 4(enp3s0np4) entered forwarding state
1000 packets in sequence (curseq=18000)
1000 packets in sequence (curseq=19000)
1000 packets in sequence (curseq=20000)
1000 packets in sequence (curseq=21000)
1000 packets in sequence (curseq=22000)
1000 packets in sequence (curseq=23000)
[  444.494212] 000: enp3s0np5: Link is DOWN
[  444.499410] 000: br0: port 5(enp3s0np5) entered disabled state
1000 packets in sequence (curseq=24000)
1000 packets in sequence (curseq=25000)
1000 packets in sequence (curseq=26000)
1000 packets in sequence (curseq=27000)
1000 packets in sequence (curseq=28000)
1000 packets in sequence (curseq=29000)
1000 packets in sequence (curseq=30000)
[  451.661756] 000: enp3s0np5: Link is UP at 1000 Mbps
[  451.662562] 000: br0: port 5(enp3s0np5) entered blocking state
[  451.679583] 001: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
[  451.679603] 001: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
1000 packets in sequence (curseq=31000)
[  452.702595] 001: br0: port 5(enp3s0np5) entered learning state
[  452.702933] 001: br0: port 5(enp3s0np5) entered forwarding state
1000 packets in sequence (curseq=32000)
1000 packets in sequence (curseq=33000)
1000 packets in sequence (curseq=34000)
1000 packets in sequence (curseq=35000)
1000 packets in sequence (curseq=36000)
[  457.806080] 000: enp3s0np4: Link is DOWN
[  457.815644] 000: br0: port 4(enp3s0np4) entered disabled state
1000 packets in sequence (curseq=37000)
1000 packets in sequence (curseq=38000)
[  459.853855] 000: enp3s0np5: Link is DOWN
[  459.859691] 000: br0: port 5(enp3s0np5) entered disabled state
1000 packets in sequence (curseq=39000)
Fail: 5920 drops
[  467.022631] 000: enp3s0np5: Link is UP at 1000 Mbps
[  467.022781] 000: br0: port 5(enp3s0np5) entered blocking state
[  467.024780] 001: br0: port 5(enp3s0np5) entered learning state
[  467.025110] 001: br0: port 5(enp3s0np5) entered forwarding state
[  467.043940] 000: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
[  467.043990] 000: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
1000 packets in sequence (curseq=46570)
1000 packets in sequence (curseq=47570)
1000 packets in sequence (curseq=48570)
[  470.094390] 000: enp3s0np4: Link is UP at 1000 Mbps
[  470.095154] 000: br0: port 4(enp3s0np4) entered blocking state
[  470.112022] 000: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
[  470.112072] 000: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
[  470.148236] 000: br0: port 4(enp3s0np4) entered learning state
[  470.148579] 000: br0: port 4(enp3s0np4) entered forwarding state
1000 packets in sequence (curseq=49570)
1000 packets in sequence (curseq=50570)
1000 packets in sequence (curseq=51570)
1000 packets in sequence (curseq=52570)
^C[root@kTSN2-sk demo_cb]#
```

Figure 31: Output of the test program on receiving side when CB mechanism is enabled. No packets are lost if one of the links is disabled. Packets are lost if both links are disabled.



```
[root@kTSN2-sk demo_cb]# ./listener
1000 packets in sequence (curseq=1000)
1000 packets in sequence (curseq=2000)
1000 packets in sequence (curseq=3000)
1000 packets in sequence (curseq=4000)
1000 packets in sequence (curseq=5000)
[  689.106808] 003: enp3s0np4: Link is DOWN
[  689.112447] 003: br0: port 4(enp3s0np4) entered disabled state
Fail: 597 drops
1000 packets in sequence (curseq=7027)
1000 packets in sequence (curseq=8027)
1000 packets in sequence (curseq=9027)
1000 packets in sequence (curseq=10027)
1000 packets in sequence (curseq=11027)
[  694.227255] 003: enp3s0np4: Link is UP at 1000 Mbps
[  694.227671] 003: br0: port 4(enp3s0np4) entered blocking state
[  694.244730] 002: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
[  694.244776] 002: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
1000 packets in sequence (curseq=12027)
[  695.648310] 001: br0: port 4(enp3s0np4) entered learning state
[  695.649455] 000: br0: port 4(enp3s0np4) entered forwarding state
1000 packets in sequence (curseq=13027)
Fail: 56 drops
1000 packets in sequence (curseq=14525)
[  698.323495] 003: enp3s0np5: Link is DOWN
[  698.328623] 003: br0: port 5(enp3s0np5) entered disabled state
1000 packets in sequence (curseq=15525)
1000 packets in sequence (curseq=16525)
1000 packets in sequence (curseq=17525)
1000 packets in sequence (curseq=18525)
1000 packets in sequence (curseq=19525)
1000 packets in sequence (curseq=20525)
[  704.467654] 003: enp3s0np5: Link is UP at 1000 Mbps
[  704.467794] 003: br0: port 5(enp3s0np5) entered blocking state
[  704.483842] 002: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
[  704.483887] 002: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
1000 packets in sequence (curseq=21525)
[  705.595927] 002: br0: port 5(enp3s0np5) entered learning state
[  705.596261] 002: br0: port 5(enp3s0np5) entered forwarding state
1000 packets in sequence (curseq=22525)
1000 packets in sequence (curseq=23525)
1000 packets in sequence (curseq=24525)
1000 packets in sequence (curseq=25525)
1000 packets in sequence (curseq=26525)
[  711.635306] 003: enp3s0np4: Link is DOWN
[  711.640915] 003: br0: port 4(enp3s0np4) entered disabled state
1000 packets in sequence (curseq=27525)
Fail: 1355 drops
1000 packets in sequence (curseq=30582)
[  714.707766] 003: enp3s0np5: Link is DOWN
[  714.712962] 003: br0: port 5(enp3s0np5) entered disabled state
[  722.899995] 003: enp3s0np5: Link is UP at 1000 Mbps
[  722.900136] 003: br0: port 5(enp3s0np5) entered blocking state
[  722.924240] 000: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
[  722.924266] 000: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
[  724.597973] 001: br0: port 5(enp3s0np5) entered learning state
[  724.598729] 001: br0: port 5(enp3s0np5) entered forwarding state
1000 packets in sequence (curseq=31582)
Fail: 10507 drops
1000 packets in sequence (curseq=43473)
[  726.995857] 003: enp3s0np4: Link is UP at 1000 Mbps
[  726.996266] 003: br0: port 4(enp3s0np4) entered blocking state
[  727.021386] 003: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
[  727.021433] 003: EDGX-BR-CORE: pt_do_ioctl SIOCSHWTSTAMP ...
1000 packets in sequence (curseq=44473)
1000 packets in sequence (curseq=45473)
[  728.649491] 002: br0: port 4(enp3s0np4) entered learning state
[  728.649828] 002: br0: port 4(enp3s0np4) entered forwarding state
Fail: 55 drops
1000 packets in sequence (curseq=47525)
1000 packets in sequence (curseq=48525)
1000 packets in sequence (curseq=49525)
^C[root@kTSN2-sk demo_cb]# _
```

Figure 32: Output of the test program on receiving side, without CB mechanism. Packets are lost if the active link (in this example np4-np4) is disconnected and at switchover upon reconnection. Packets are not lost if the disabled link is disconnected (in this case np5-np5 if np4-np4 is connected).

3.5 Configuration of networks

The configurations for the tests described in the previous chapters were done using the command line toolset tsntool. Each configuration thus had to be done locally on the machine in question. This method gives great control over the configuration of the network but is however very impractical if the network size increases or if many networks need to be configured.

In this section we briefly present another configuration method based on the slate XNS software from TTTech more akin to what the case for a commercial implementation of TSN networks would be. The slate XNS software is in development by TTTech and had, at the time of testing, a reduced set of functionalities. It can be used to:

- Plan a communication schedule for IEEE 802.1Qbv based on a description of the network and the streams of interest, where scheduled streams will be assigned with the priority field PCP 7.
- Deploy the configuration for the schedule to the different devices of the network over NETCONF from a management computer, or save the configuration files for later deployment using NETCONF

It cannot however, at the time of testing, be used to:

- Configure frame preemption according to IEEE 802.1Qbu
- Configure redundancy using IEEE 802.1CB

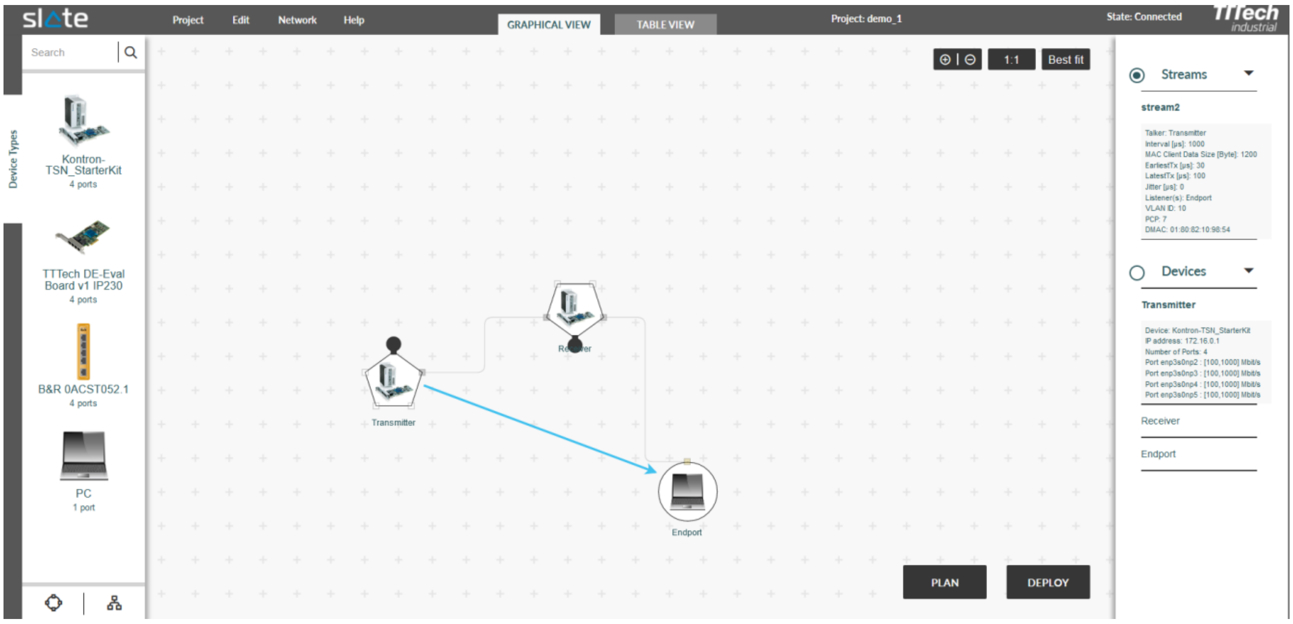
These functionalities are expected to be added in the future once the TSN Networks and planning tools become more widespread.

The network is described using a graphical user interface. The different devices are specified with their IP addresses. An example is shown in Figure 33 where the setup of the basic setup from Figure 7 is represented, with the ETH-NIC and the TSN-NIC represented as separate devices (each with an IP address) as the TSN-NIC acts as a switch and the ETH-NIC as the endpoint. A stream was also defined from the transmitter to the ETH-NIC. A set of parameters can be set to define the connections and properties of the stream and will be used to plan the communication schedule. If several streams are specified the planning will make sure that within the parameters given, there will be no collisions and the messages can arrive in time at their destinations. More information on the different options and parameters can be found in the slate XNS manual ([11]). The schedule for our example which aimed to recreate the configuration for the setup from Figure 7 is shown in Figure 34.

The schedules shown in Figure 34 were deployed over the network using NETCONF from an external computer which was running the slate XNS program and connected on one of the available ports of the TSN-NIC of the transmitting side. Only a basic configuration had to be done on the machines prior to the deployment. This involved mainly setting up IP addresses so that the management computer could communicate with all the machines.

The configuration was tested using the scheduled traffic only generated and evaluated as in the tests for the IEEE 802.1Qbv mechanism. The results are shown in Figure 35. The latency obtained is about 70 µs, indeed this corresponds to the schedule configured on the receiving side which opens a window for transmission of the scheduled traffic (PCP 7) from 70 µs to 90 µs after the cycle start and is different from the schedule at the transmitting side (50 µs to 70 µs). This was chosen by the program during planning of the communication schedule and considers different parameters such as the transmission time in the cables, the maximum allowed latency for the messages, the maximum message size etc. that were specified. As the generated configuration has two stages of transmission windows with different start and end times, the jitter becomes very low as it is only sensitive to the jitter on the last link. The results correspond thus to the deployed schedule.

If developed further this tool and similar ones can be very useful to plan and deploy more complex networks than the ones tested in this report. The fact that the parameters can be defined based on the needs of each stream make the planning of the networks in an automated manner a lot easier. The same configuration can then be easily deployed on the networks of all the vehicles in question. Adding and removing devices and streams can also be done easily, might however change the outcome of the planned schedule for other streams and devices as well. Configuring the devices centrally over the network therefore eases the initial configuration. For maintenance of the network, it must be considered that the central configuration does not conflict with security requirements. Therefore, it might not be the preferred method in operation. However, the planning of the schedules and creation of configuration files for the different devices remain powerful functionalities for the management of the network.



stream2

Talker: Transmitter
Interval [µs]: 1000
MAC Client Data Size [Byte]: 1200
EarliestTx [µs]: 30
LatestTx [µs]: 100
Jitter [µs]: 0
Listener(s): Endport
VLAN ID: 10
PCP: 7
DMAC: 01:80:82:10:98:54

Transmitter

Device: Kontron-TSN_StarterKit
IP address: 172.16.0.1
Number of Ports: 4
Port enp3s0np2 : [100,1000] Mbit/s
Port enp3s0np3 : [100,1000] Mbit/s
Port enp3s0np4 : [100,1000] Mbit/s
Port enp3s0np5 : [100,1000] Mbit/s

Receiver

Endport

Figure 33: Example of a network description in the graphical user interface of the slate XNS tool. The network described corresponds to the setup shown in Figure 7.

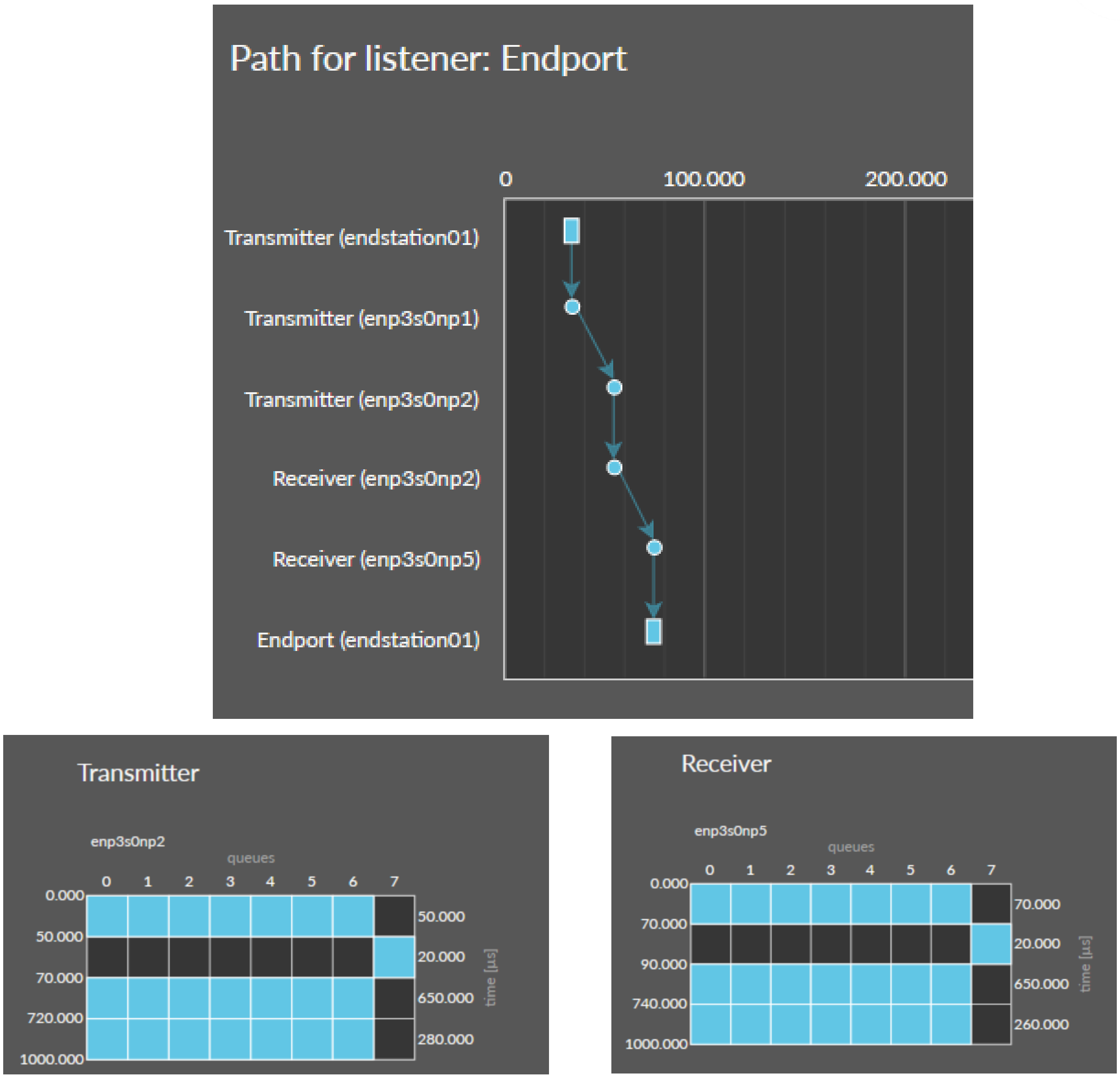


Figure 34: Schedule for the stream from the transmitter to the end port (ETH-NIC) for the setup of Figure 7. The top shows the path for the messages, below are shown the schedules for the gates at the switches on the transmitter-side and receiver-side (TSN-NIC).

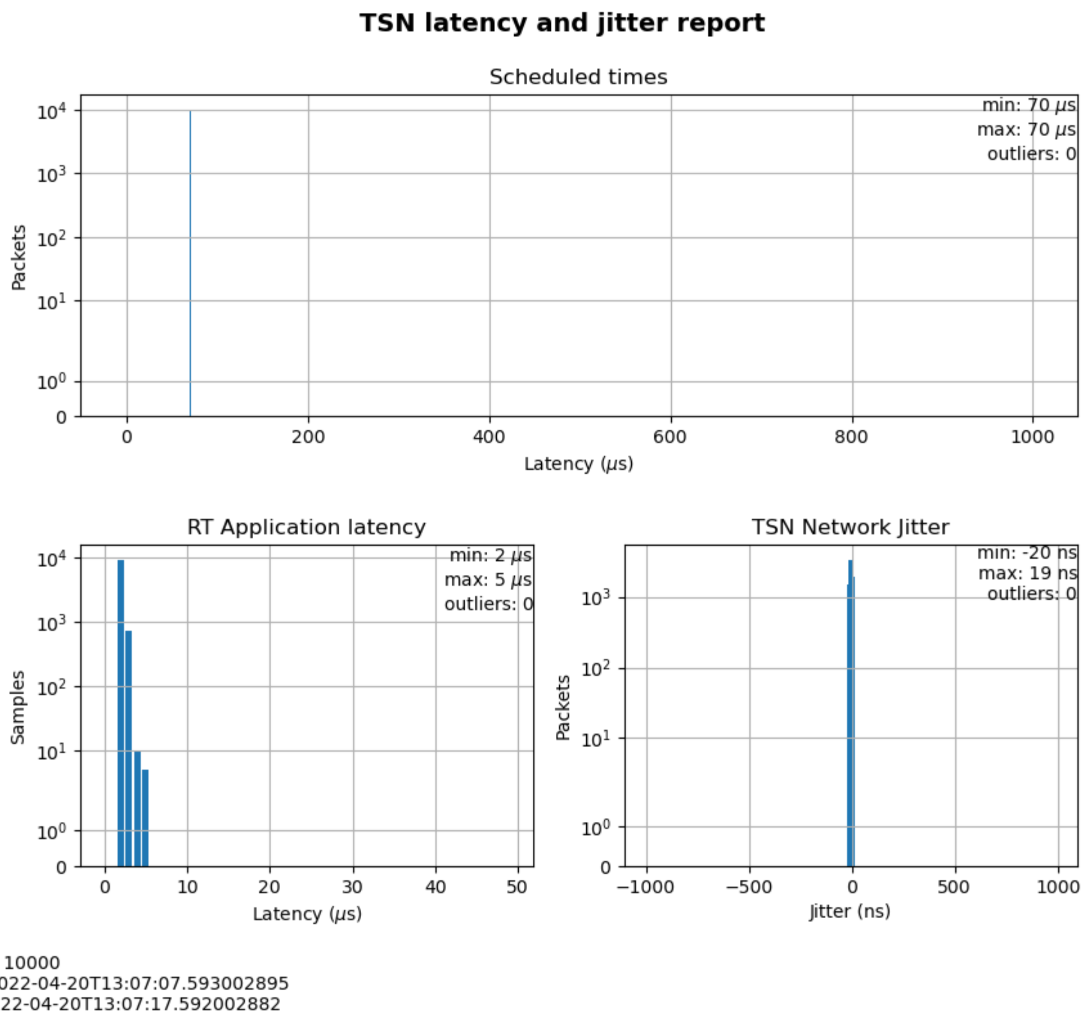


Figure 35: Histograms of latency, application latency and network jitter for scheduled traffic alone with the configuration according to Figure 34.

4 Task 2: End-to-End TSN Connection with TCNOpen TRDP

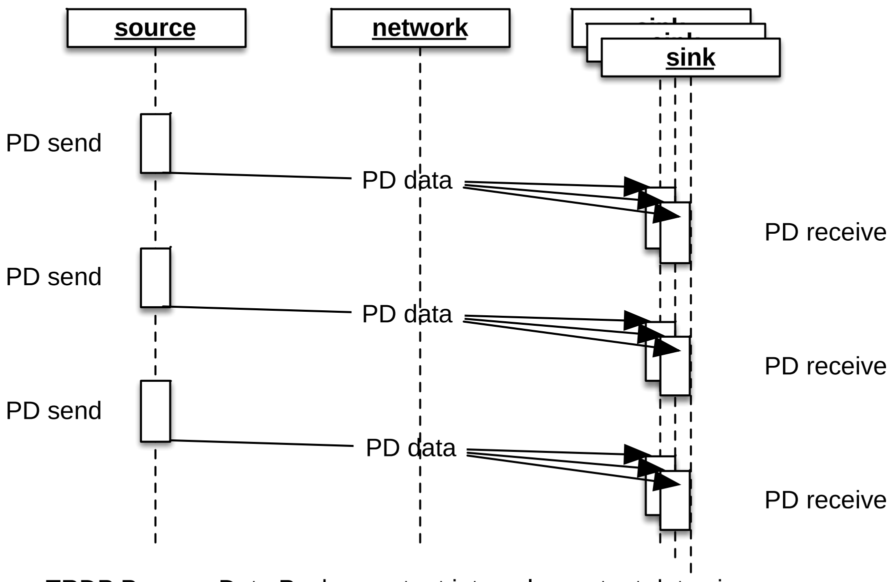
4.1 TCNOpen TRDP stack with TSN – inside

Task 2 was mainly done by NewTec GmbH who has deep knowledge of the TRDP protocol stack as they were involved implementing some parts of TRDP for the TCNopen project. Therefore, TRDP was chosen to be used as session layer protocol for task 2. The currently standardized TRDP protocol (IEC 61375-2-3; Annex A) in version 1.x offers several features, which will not be useful for hard real time applications (TSN) and will thus not be supported for use with TSN. For task 2 TRDP version 2.1 was chosen to be used as session layer protocol, as this is the first version supporting TSN features of the data link layer.

4.1.1 Communication Patterns

TRDP supports several communication patterns, which provide applications with reliable or fast communication within the TCN. While message data (MD) transmission can be compared to a client/server scheme to allow request/reply and request/reply/confirm communication sessions, the process data (PD) part offers interval-driven, cyclic data transmissions (push-pattern) of relatively small data fitting into one network frame. The standard also provides a ‘pull’-pattern, where a requester can force a publisher to source its data.

- Only the PD-Push pattern will offer support for TSN



TRDP Process Data Push: constant interval, constant data size, point to multipoint

Figure 36: Push Pattern

Applications using PD communication must describe the telegrams to the TRDP layer first by calling a function called ‘PD.publish’. With this call, the application prepares the TRDP stack’s internal protocol handling by supplying

- the ComId, data and size	-> what to send
- the destination address	-> where to send to
- the cycle / interval time	-> how often to send
- the communication parameters	-> how to send (**New[6]: vlan, TSN**)

- management parameters (session/internal handle, callback, topography counters)

[6] New in TRDP version 2.x

The TRDP layer will start transmitting if all necessary data was provided; the application can update the payload by calling the function PD.putData later at any time.

On the receiver side, the subscriber provides the stack with similar information:

-	the ComId and expected size	-> what to receive
-	a source and/or destination addresses	-> filter parameters
-	the time-out	-> sink time supervision
-	**New[6]: the communication parameters**	**-> how to receive (vlan, TSN)**
-	management parameters	-> use callback or polling etc.

Incoming data is either polled by calling PD.poll or by providing a callback function (labelled PD.indicate in the standard).

4.1.2 Addressing Scheme

TRDP PD communication uses as transport UDP/IP datagrams – the address range for intra-consist communication is 10.0.0.0/9 whereas 10.128.0.0/18 is the defined network range for the ETB (train-wide or inter-consist communication).

Addresses are mapped via NAT and R-NAT when communicating between separate consists. Additionally, IP multicast addressing is used both on ECN and ETB. To account for changes in train-wide IP addresses (→inauguration), topography counters are maintained and sent with every TRDP telegram to validate the addresses.

TRDP address parameters:

- ECN-local unicast IPv4 address
- ECN-local multicast IPv4 address
- ETB-train-wide unicast IPv4 address
- ETB-train-wide multicast IPv4 address
- etbTopoCnt
- opTrnTopoCnt
- VLAN ID (optional)

TSN communication, however, acts on OSI Layer 2 and its addressing is defined by:

- Destination MAC address (multicast)
- VLAN ID
- Priority

4.1.3 Framing

For TSN PD-PDU a reduced preliminary header format is used:

0 7	8 15	16 23	24 31
SequenceCounter			
Version	MsgType	DatasetLength	
ComId			
HeaderFCS			
… Dataset [0...1458Bytes] …			

Table 3: TRDP TSN PD-PDU

The fields of TRDP TSN PD-PDU are:

- SequenceCounter: For non-safe payloads, to detect redundant frames, UINT32
- Version: 0x02 (Header version of TCNOpen TRDP)
- MsgType: 0x01 (TSN non-safe PD), 0x02 (TSN-PD secured with SDT-Layer)
- DatasetLength: Size of payload, UINT16 [0...1458Bytes]
- ComId: Unique Identifier for payload
- HeaderFCS: CRC32

The major version number is at the same location as in the standard TRDP header – thus it will be recognized and discarded by all current standard TRDP stacks (version 1.x). The header size sums up to a total of 16 Bytes.

Compared to the Version 1 header, TRDP TSN has no need for topography counters, because it is not IP-routable. The TRDP stack must be able to discriminate between the two header formats automatically.

4.1.4 Ordinary best-effort TRDP

The implementation of the TCNOpen TRDP stack for "ordinary" best-effort data transmission aims to ease the usage of TRDP on several platforms and target systems. A VOS called layer (Virtual Operating System) abstracts all target system dependent calls. VOS implementations are currently available for several POSIX-compliant systems (Linux, QNX, Darwin) and also for VxWorks, Windows and FreeRTOS/lwIP.

Applications for TRDP can be written using VOS and stdclib functions only. With these functions an operating system (OS) independent API is given.

Figure 37 shows the hierarchy and layers of a TRDP application. The naming of the TRDP Light API[7] functions reflect the main units of the protocol stack:

- tlp_... **T**RDP **L**ight **P**rocess Data functions
- tlm_... **T**RDP **L**ight **M**essage Data functions
- tlc_... **T**RDP **L**ight **C**ommon functions
- tau_... **T**RDP **A**pplication **U**tilities functions
- vos_... Virtual Operating System functions

The VOS functions cover most requirements of applications, and of course, handle all resource requirements of the TRDP protocol stack (memory & networking). Threading is supported, but except for some timer functions, it is not actively used by the protocol stack. Only the TRDP utilities supporting URIs (DNR) and inauguration (TTI) use separate threads and utilities like semaphores. These utilities have to be used by the application or the TRDP High Layer API.

[7] To be fully compliant to the ECN requirements demanded for in the IEC 65375 2-3 standard, the application or a dedicated layer (e.g. TRDP High Layer) must provide train inauguration awareness, thread handling and possibly XML based configuration. To realize these functionalities the **T**RDP **A**pplication **U**tility functions (tau_...) can be used.

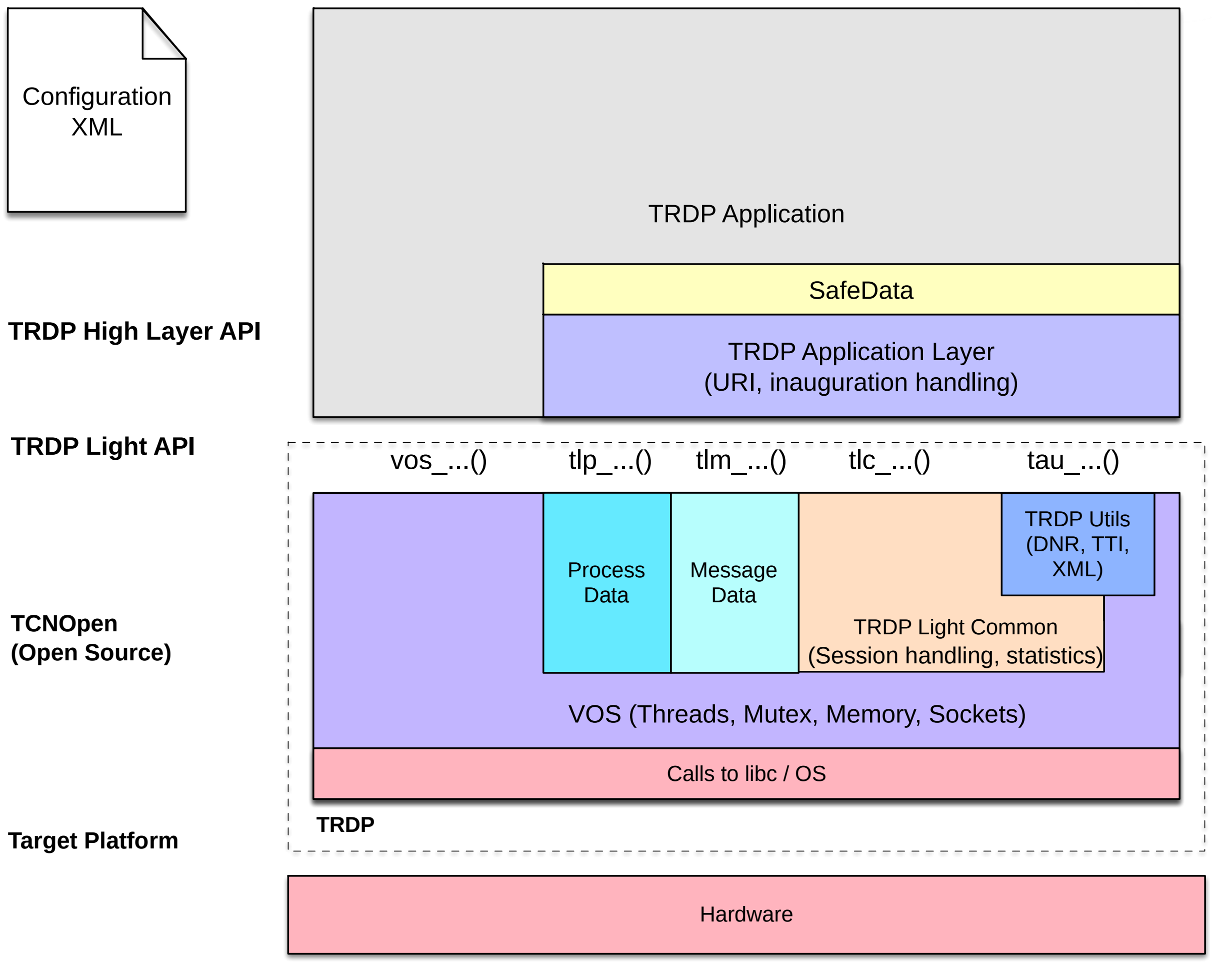


Figure 37: Hierarchy of the TCNOpen TRDP stack

4.1.4.1 Sequence and Control Flow

Sending and receiving of PD telegrams is normally handled inside the `tlc_process()` function, which must be called regularly by the application (or from a communication thread). The application can synchronously (via callback) or asynchronously (`tlp_put()` / `tlp_get()`) provide or read data.

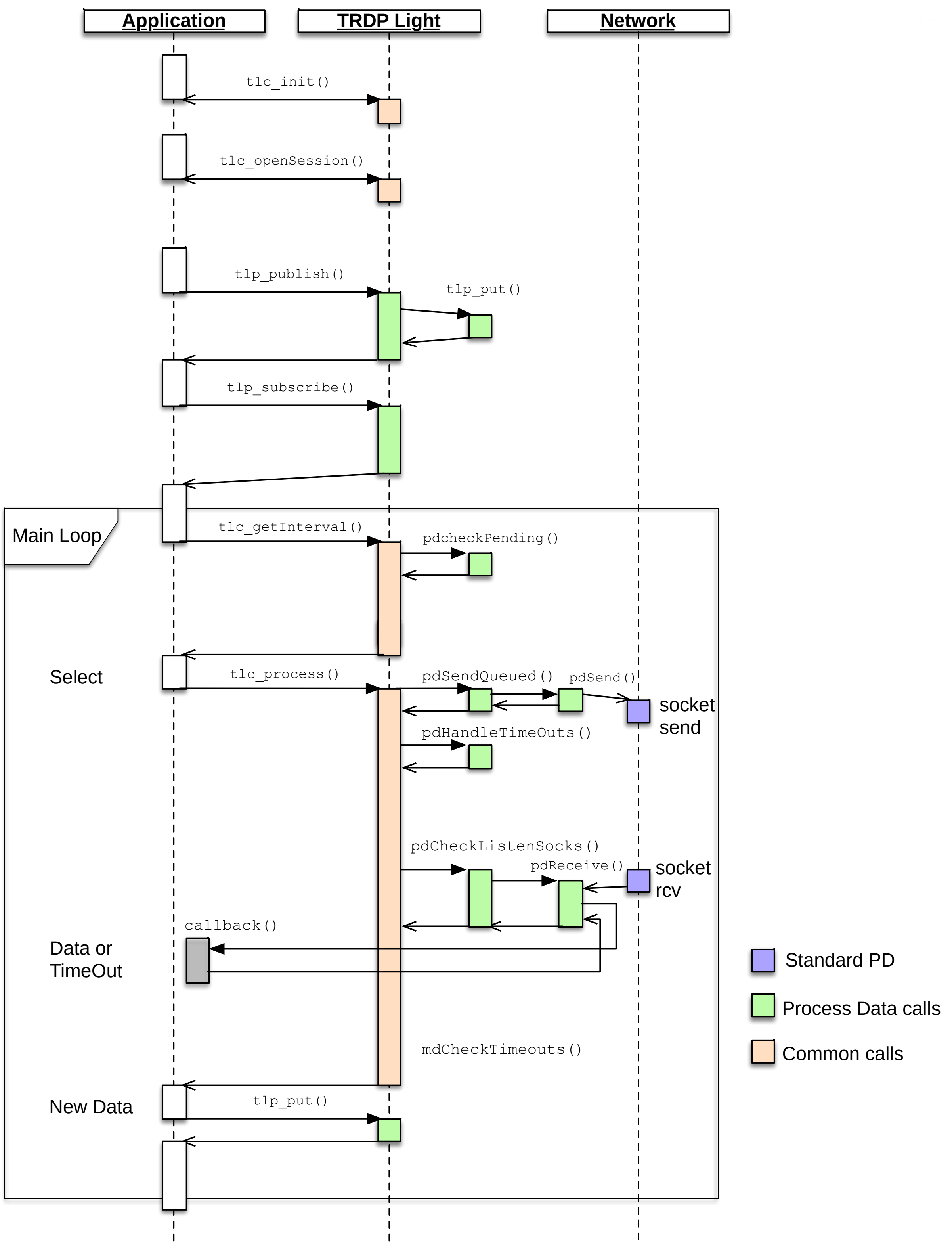


Figure 38: Legacy PD handling

Note: *"TRDP Light" is the synonym for the API of the TCNOpen TRDP-Stack. When integrating the TCNOpen TRDP-Stack into customized devices, often a dedicated layer (e.g. TRDP High Layer) is introduced towards the application (to handle train inauguration, URI/IP address resolution etc.).*

The sequence diagram in Figure 38 shows that sending and receiving is handled inside the context of `tlc_process()` – in this example received telegrams (or timeouts thereof) trigger an appropriate callback (grey box) into the application. Because sending of standard PD is done within that loop, callback handling should be as fast as possible to mitigate jitter or protocol violation.

4.1.5 TRDP-TSN

For TRDP over TSN, the TCNOpen TRDP stack (as of V1.4) is extended by

- the capability to configure TSN Stream Ids (VLAN, Multicast, Priority)
- one additional API function to push TRDP-TSN packets immediately
- adding and recognizing a reduced header format (TRDP TSN PD-PDU)
- adding TSN related networking functions to the VOS layer
- supporting RAW IP sockets (planned to be changed to RAW Ethernet sockets)

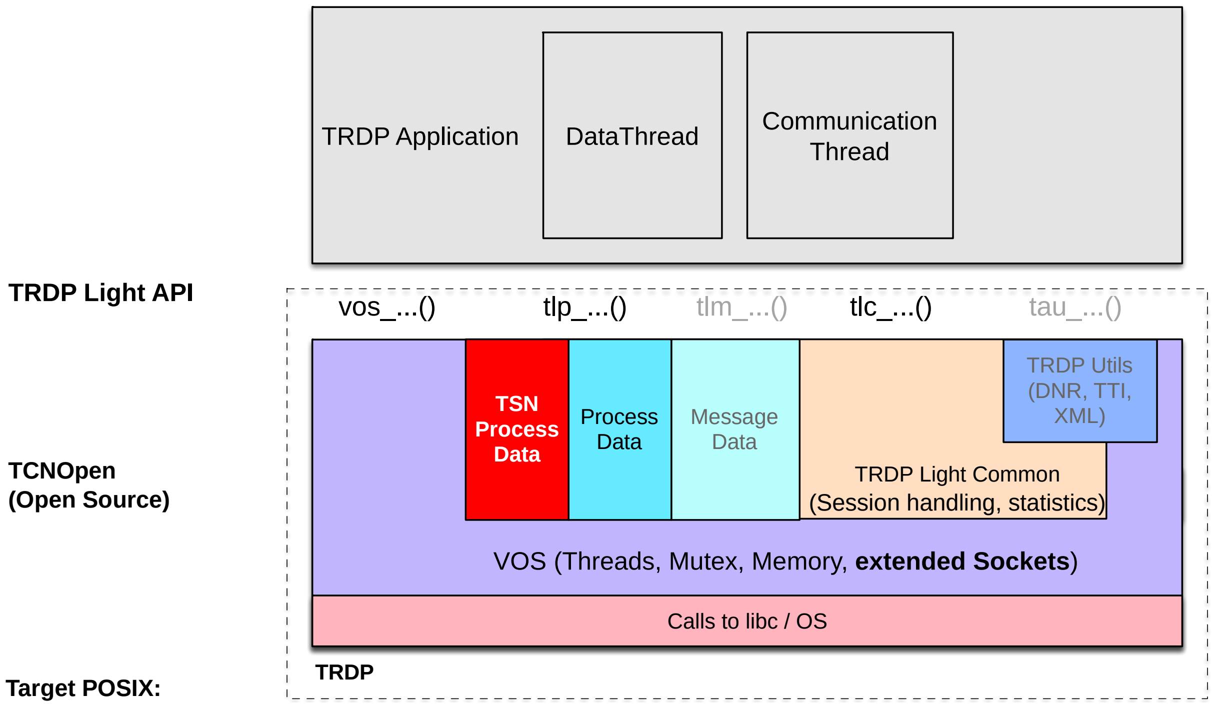


Figure 39: TRDP with TSN

The TSN Process Data in Figure 39 marked in red is actually not a separate module, or unit, but is integrated into and depicts a special behaviour of existing functionality. Parts which are not used in the PoC are greyed out (TRDP Utils, Message Data).

4.1.5.1 Sequence and Control Flow

When handling TSN telegrams, the timely transmission of PDs is moved into the application's responsibility (Figure 40).

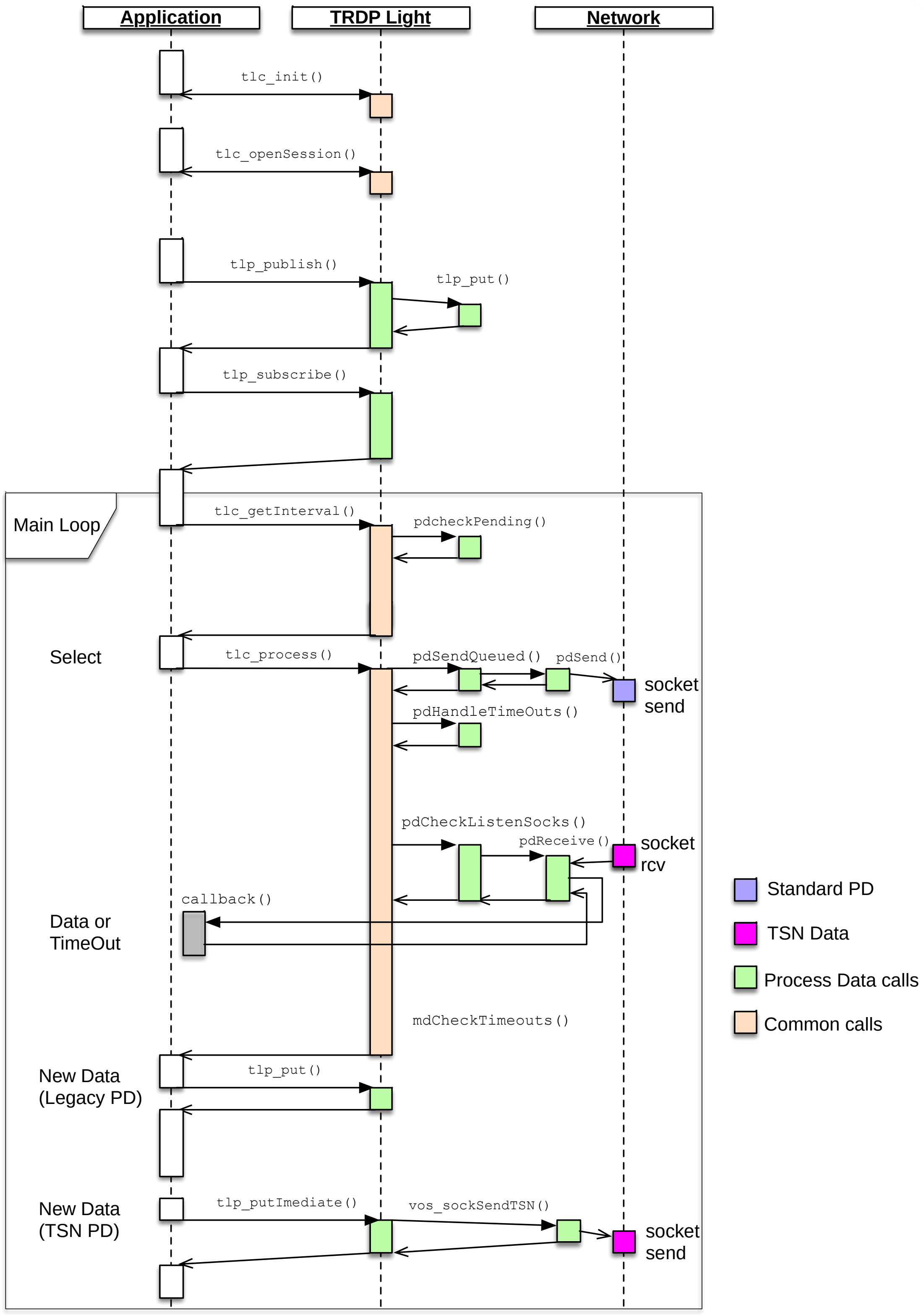


Figure 40: Sending TSN PD-PDU

Note: *"TRDP Light" is the synonym for the API of the TCNOpen TRDP-Stack. When integrating the TCNOpen TRDP-Stack into customized devices, often a dedicated layer (e.g. TRDP High Layer) is introduced towards the application (to handle train inauguration, URI/IP address resolution etc.).*

4.1.5.2 Publish and Subscribe

Applications using TRDP consist of an initialization phase, where publisher and subscribers are prepared and set up, and a work loop, where data is processed – usually by I/O operations.

These calls take as parameters source and destination addresses for the defined ComIds and need additional identifiers for the TSN parameters:

`tlp_publish()`:

- The communication parameter ('ComPar' from EN 61375-2-3:2017-02 Annex C.6) additionally provides the VLAN Id and a TSN indicator (PCP value) for this process data.
- Additional packet flags, `TRDP_FLAGS_TSN`, `TRDP_FLAGS_TSN_SDT` and `TRDP_FLAGS_TSN_MSDT` define the new PD message types for the TRDP TSN PD-PDU (see Table 3).

  If one of these flags is set, the interval and topography counter parameters will be ignored, the telegram will never be sent automatically by the TRDP stack (`tlc_process()`) and the shorter TSN PD-PDU will be prepared.

`tlp_subscribe()`:

- A communication parameter ('ComPar' from EN 61375-2-3:2017-02 Annex C.6) additionally providing the VLAN Id and a TSN indicator is added to the parameter list.
- The additional packet flag `TRDP_FLAGS_TSN` is used to select or create a TSN capable socket and to define the header message type further.

  This new parameter allows the correct receiver to be created by supplying the VLAN Id.

  Providing a callback is the preferred method.

4.1.5.3 Socket handling

Because telegram definitions (published ComIds) can have different communication parameters like `QoS`, `TTL`, `MaxNoOfRetries`, the non-TSN implementation already creates or selects separate sockets with according socket options for each communication class.

Managing required sockets is extended for TSN in trdp_requestSocket().

4.1.5.4 Put Immediate

For best effort PD `tlp_put()` updates the payload in the internal network buffers and does not send it directly. For use with TSN the application (or higher layers) must be able to trigger transmission to support synchronous operation. This is realized with:

`tlp_putImmediate()`:

- Parameters as `tlp_put()`, plus
- an additional time value TxTime.

  TxTime can be set to the absolute transmission time in ns for that packet. If set, it is provided via `vos_sockSendTSN()` to the socket's `sendmsg()` function. If the driver/NIC supports this control command (`SCM_TXTIME`) this can increase timing precision and reduce jitter related to socket delays.

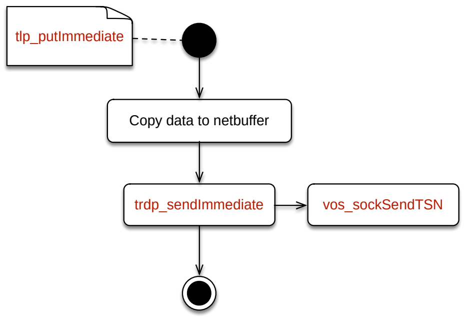


Figure 41: tlp_putImmediate

Note: *The non-TSN i210-interface of the Kontron-Kbox supports the hardware timestamping with `SCM_TXTIME`.*

4.1.5.5 VOS-Layer

Thread & Timing Functions

To enable cyclic and synchronous thread execution for TSN `vos_threadCreate()` is enhanced to support cyclic execution and a `startTime` parameter is added to synchronize to the send schedule.

Time-related functions of the TRDP protocol engine use monotonic time to avoid interruption of standard PD transmission in case of setting the system clock. The monotonic clock will not be adjusted by PTP and thus cannot be used as time base for TSN. An additional function to return the real time (which should be synchronized inside the system via gPTP) is necessary:

`vos_getRealTime():`

- returns `CLOCK_REALTIME`

Network (Socket) Functions

An application using the TCNOpen TRDP stack needs to open a session with the stack, which binds all traffic to an interface IP address. If we want to be able to have one application session taking care over PD, MD and TSN-PD, each communication method has its own parameters.

Sockets are handled per application and are re-used where possible. Sockets in TRDP were either UDP/IP or TCP/IP.

With TSN PD a Layer 2 communication must be set up, which uses a quasi-UDP/IP header to avoid defining a new Ethertype. From a socket point-of-view, there will be a separate virtual interface, which defines the VLAN ID for ingressing and egressing traffic.

The TSN socket needs to bind to this VLAN interface – and this is OS dependent. For the PoC implementation, for each requested VLAN ID, the list of available interfaces is traversed and, if a match is found, the requested socket will bind to its IP address.

`vos_sockOpenTSN()`

- Create an adequate socket (Raw socket preferred) to use with egressing TSN PDs.

The function `trdp_requestSocket()` as part of `trdp_utils.c` calls these functions when publishing or subscribing to TSN process data.

`vos_sockSendTSN()`

- Send TSN over UDP data. Create header manually.
- Optional: Provide the transmit time via the socket control message (`SCM_TXTIME`)

`vos_sockReceiveTSN()`

- Wrapper for receiving standard UDP.

Note: RAW IP sockets cannot be used for receiving - a standard `SOCK_DGRAM` socket is used.

4.2 Sample code of PoC

To show the basic functionality of the TRDP TSN extension, existing sample applications of the TCNOpen TRDP stack are ported and adapted for TSN:

- `sendTSN` (from `sendHello`)
- `receiveTSN` (from `receiveHello`)

These two samples resemble a simplified multi-threaded application.

As the names suggest, `sendTSN` acts as a source of timed data and `receiveTSN` acts as a sink. The basic behaviour is fixed, but TSN related parameters can be changed using command line parameters. Invoking the applications with `–h` displays all configurable parameters (e.g. TSN Stream Id).

4.2.1 Data Exchange

Two ComIds/datasets are transmitted:

Traffic class	cycle	ComId	Priority	VLAN
TRDP TSN PD	10ms or 1ms	1000	PCP7	10
TRDP ordinary best effort PD	100ms	0	PCP0	

Table 4: Sent TRDP Datasets

4.2.2 sendTSN

The C-unit `sendTSN.c` consists of 4 functions (excluding `main()`):

-	`dbgOut`:	print error and info messages of the TRDP stack to the console
-	`usage`:	output the help/usage message to the console
-	`comThread`:	communication thread handling the TRDP
-	`dataAppThread`:	input/output application thread updating TSN data
-	`main`:	Initialize the framework/stack and then update TRDP data

The `dbgOut` and `usage` functions are trivial and self-explanatory and are not covered here. For an overview, please refer to the activity diagram in Figure 42.

Main

On invocation, the main function evaluates the command line arguments, initializes the TRDP stack and opens an application session on the selected or default interface. All incoming or outgoing standard TRDP traffic will be routed through that interface. Next, a communication thread is created, the TSN ComId 1000 and the standard ComId 0 are published, and the TSN data producer thread is spawned.

Note: The comThread can be created before any publisher, where the TSN data producer needs the publisher handle for the right ComId.

The remaining part of the main function updates the ComId 0 packet every 200 ms with a new counter value.

Communication Task

The `comThread` uses the POSIX select function to wait for any ready socket descriptor or timer events and calls the `tlc_process` function which in turn handles all protocol and ordinary TRDP traffic related events – except for sending TSN PD-PDU. This thread usually runs until system shutdown.

Data Producer Task

The data producer task is created as an interval thread with the specified cycle and start time for the producer function. Syncing with the network is currently done manually using delays but is foreseen to be enhanced with support of timer functions or real time tasks (refer to `vos_runCyclicThread`).

The `dataAppThread` function gets the current time (now) and writes it as `TIMEVAL64` to the dataset. Additionally, an ASCII representation is appended, as well.

The dataset is then sent directly to the network stack via `tlp_putImmediate()`.

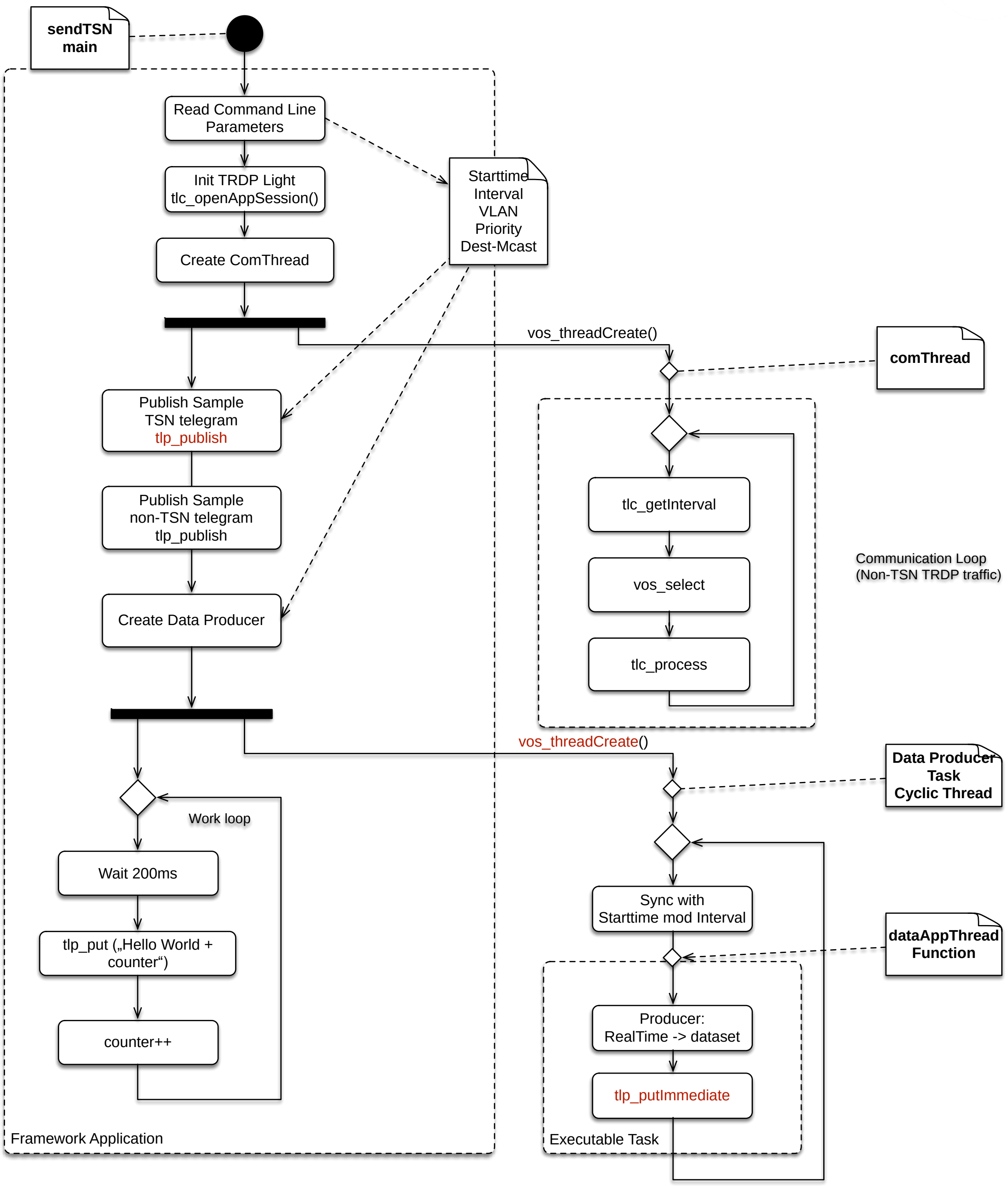

sendTSN main
Read Command Line Parameters
Init TRDP Light tlc_openAppSession()
Create ComThread
Starttime Interval VLAN Priority Dest-Mcast
vos_threadCreate()
comThread
Publish Sample TSN telegram tlp_publish
Publish Sample non-TSN telegram tlp_publish
Create Data Producer
tlc_getInterval
vos_select
tlc_process
Communication Loop (Non-TSN TRDP traffic)
vos_threadCreate()
Data Producer Task Cyclic Thread
Work loop
Wait 200ms
tlp_put („Hello World + counter“)
counter++
Sync with Starttime mod Interval
dataAppThread Function
Producer: RealTime -> dataset
tlp_putImmediate
Framework Application
Executable Task

Figure 42: Activity diagram of sendTSN

Note: In Figure 42 and Figure 43, functions of TCNopen TRDP stack, which have been extended for TSN or added for TSN, are marked red.

4.2.3 receiveTSN

The C-unit `receiveTSN.c` consists of 4 functions (excluding `main()`):

- `dbgOut`:	print error and info messages of the TRDP stack to the console
- `usage`:	output the help/usage message to the console
- `comThread`:	communication thread handling the TRDP
- `myPDCallback`:	event handler called from comThread (tlc_process())
- `main`:	Initialize the framework/stack and then waiting for TRDP data

The `dbgOut` and `usage` functions are trivial and self-explanatory and are not covered here. For an overview, please refer to the activity diagram in Figure 43.

Main

On invocation, the main function evaluates the command line arguments, initializes the TRDP stack and opens an application session on the selected or default interface. All incoming or outgoing standard TRDP traffic will be routed through that interface. Next, a communication thread is created, the TSN ComId 1000 and the standard ComId 0 are subscribed to and the application main thread enters an idle loop.

Communication Task

The `comThread` uses the POSIX select function to wait for any ready socket descriptor or timer events and calls the `tlc_process` function which in turn handles all protocol and TRDP traffic related events – except for sending TSN PD-PDU. This thread usually runs until system shutdown.

Data Consumer

The data consumer is the callback function which was provided by the subscription calls. The callback function is called whenever a valid packet is received on that subscription.

The reception of TSN frames is triggered by an external event of incoming frame from the NIC as it is the case for non-TSN frames. . In the callback function, the delay of the received packet and an average jitter are computed and displayed. In the case that the latency turns negative – this can happen if talker and listener are not in sync – the message "…coming from the future…" is displayed.

When having TSN aware switches, the maximum latency and jitter coming from the listener depend on:

- task priority of the communication thread (switch-over time)
- time between reception and `select` signalling the ready descriptor
- time between calling `receiveMsg`, checking validity of the headers, finding the subscription and invoking the associated callback

The dataset is interpreted directly (no de-serialization or further validation…).

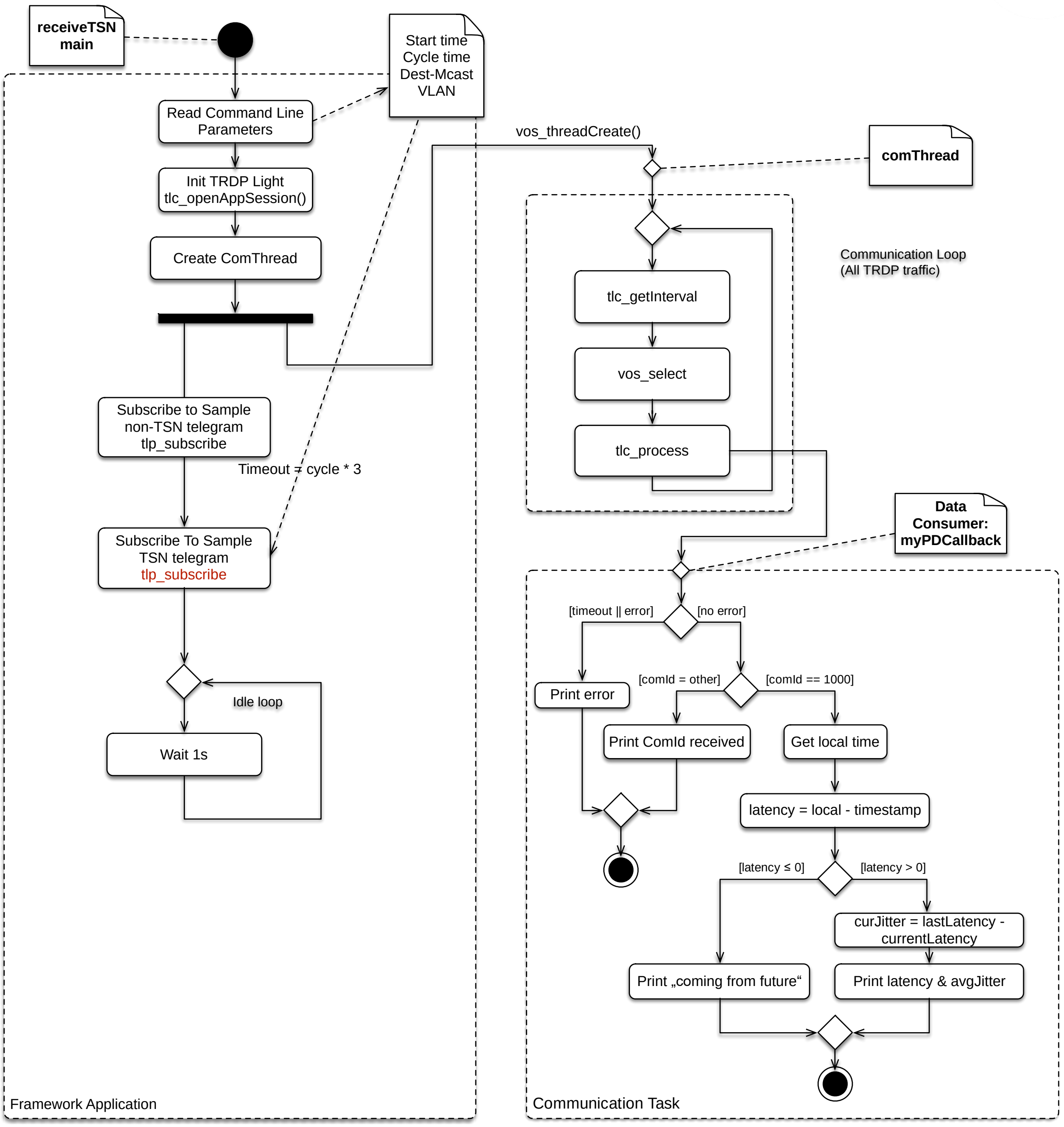

receiveTSN main
Start time
Cycle time
Dest-Mcast
VLAN
Read Command Line Parameters
Init TRDP Light tlc_openAppSession()
Create ComThread
Subscribe to Sample non-TSN telegram tlp_subscribe
Timeout = cycle * 3
Subscribe To Sample TSN telegram tlp_subscribe
Idle loop
Wait 1s
Framework Application
vos_threadCreate()
comThread
Communication Loop (All TRDP traffic)
tlc_getInterval
vos_select
tlc_process
Data Consumer: myPDCallback
[timeout || error]
[no error]
Print error
[comId = other]
[comId == 1000]
Print ComId received
Get local time
latency = local - timestamp
[latency ≤ 0]
[latency > 0]
curJitter = lastLatency - currentLatency
Print „coming from future“
Print latency & avgJitter
Communication Task

Figure 43: Activity diagram of receiveTSN

4.3 Test Set-up

The following picture is a recap of the test set-up of task 1 which uses as basis the Kontron “demo 1” with the best effort traffic generated locally.

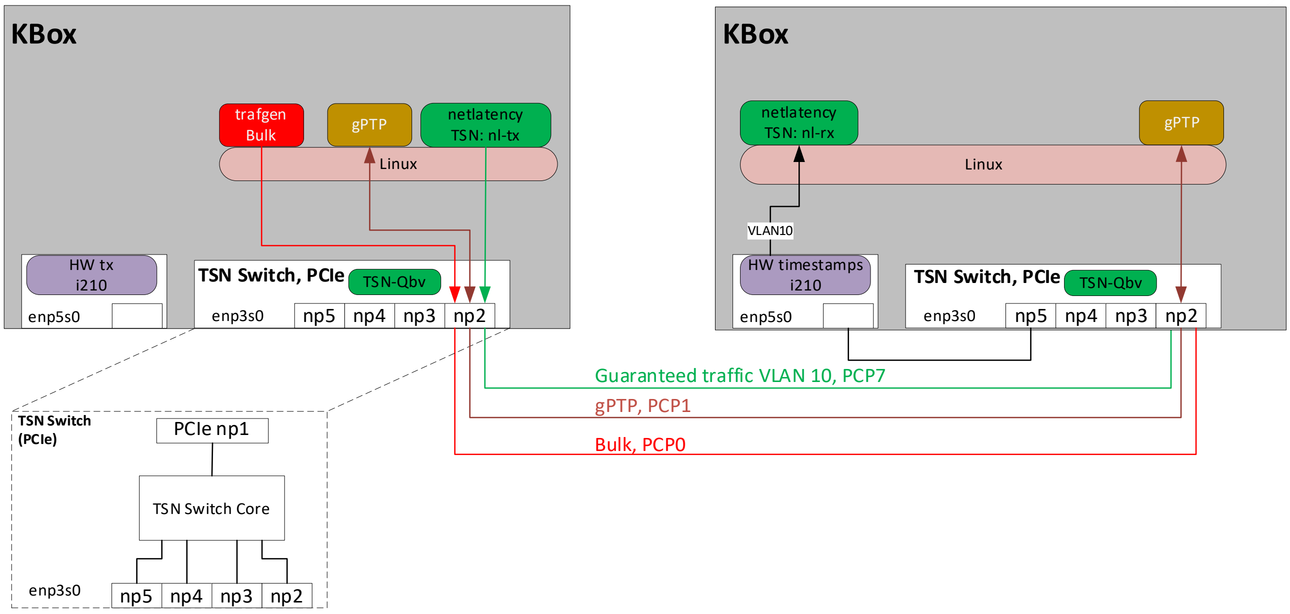


Figure 44: Recap of Set-up Task 1

The test set-up for task 2 is depicted in the next picture showing bulk traffic (best effort high volume) generated locally.

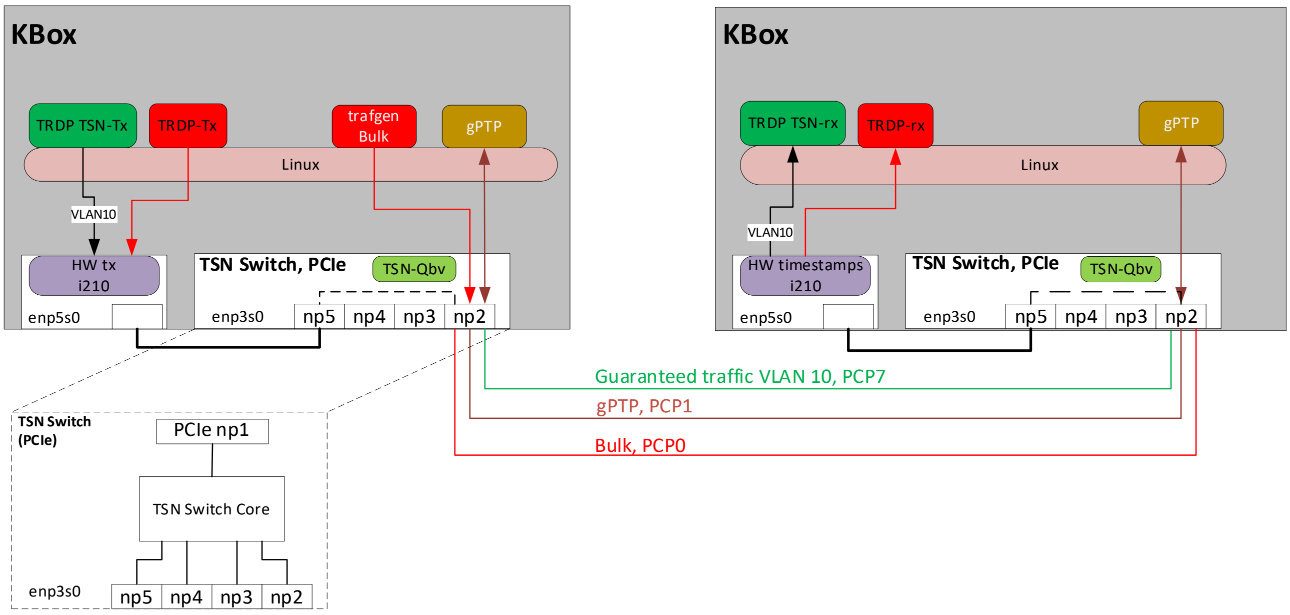


Figure 45: Test Set-up Task 2, bulk generated locally

Terminology	
TRDP TSN-tx/rx	TRDP TSN PD
TRDP tx-rx	TRDP ordinary best effort PD

Table 5: Terminology TRDP functions of Figure 45

5 Task 2: Test Implementation and Results

5.1 Test Configuration

The following two test setups show one variant α with the TSN- and PD-packets transmitted by the PCIe 0400TSN switch directly to the second switch and finally to the receiving i210 end-device in Figure 46, and the other variant β in Figure 47 with the packets transmission starting at the first i210 end-device before it is passed to the first switch and taking the same way as in the first setup. In both setups, the scheduling of the generation and transmission of the TSN-and PD packets is distributed on different threads in the same application. The thread for PD-packets is configured with lowest priority ‘0’ and with normal Linux policy ‘SCHED_OTHER’. For TSN packets, the priority is set to the maximum of ‘99’ and the Linux real-time scheduling policy is set to SCHED_FIFO with 1ms period. This real-time thread is blocking all other threads as long as it pre-empts itself by the system call clock_nanosleep(). In Figure 46 and Figure 47 the threads are displayed as TRDP-tx in red and TRDP TSN-tx in green. On the receiver side is just one thread with normal policy for both packet-types.

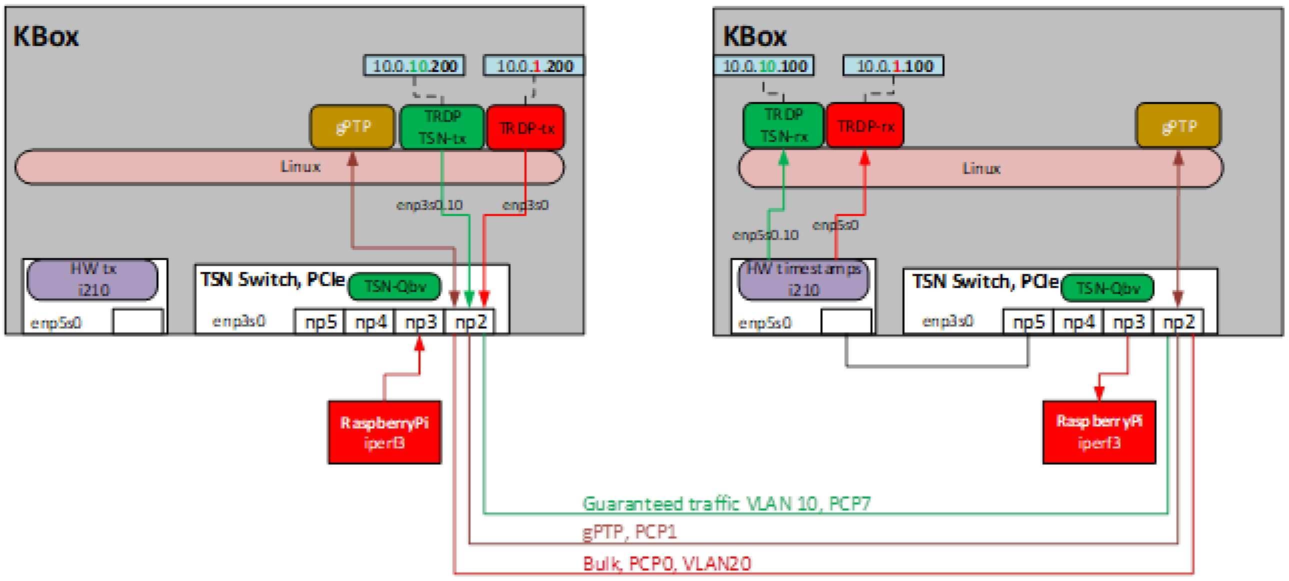


Figure 46: Test setup α with switch as sender

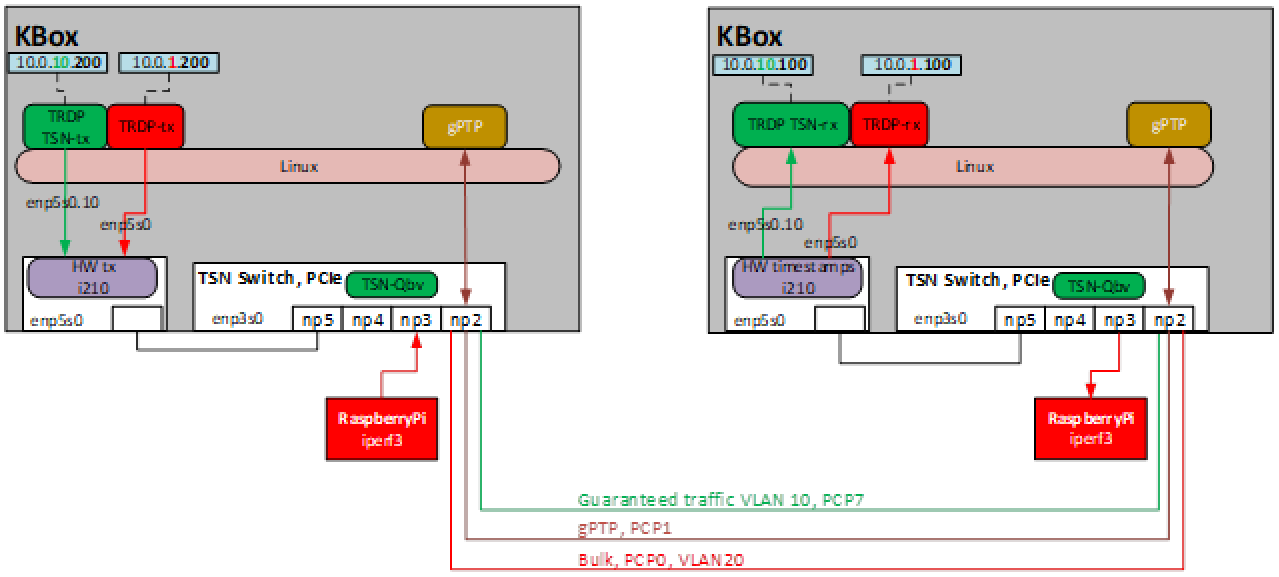


Figure 47: Test setup β with end-device (ED-i210) as sender

For the setups without Bulk-traffic, the 2 RaspberryPis are obsolete.

The setups α and β more or less cover the cases a) and b) identified in chapter 2.1.2. The difference is that two separate sources of best effort traffic were considered for all cases, namely the normal TRDP PD and the network load generated by the iperf3 scripts externally on the 2 RaspberryPis. Thus, effectively a mix of cases a) and b) was studied. Case c) was not explicitly covered, however the separation of the TSN and usual traffic was done on thread level and the question of how to segregate between the two traffics from within an application had to be addressed by the setup under investigation. Thus, a separate case was not needed to address these questions.

In order to run the TRDP-Stack on the Kontron HW and to produce timing outputs that are comparable to the Kontron demonstration example, the following subsections contain the general applied changes. All tests are performed using IP-raw-sockets in one step and Ethernet-raw-sockets in a second step.

5.1.1 Network Interface Configuration

The TRDP-Stack default interfaces were set to use the same network-topology as used by the Kontron example in task 1. On the transmitter side it is the device enp3s0 as provided by Fedora Linux for sending the packets and on receiver side the software receives the packets at device enp5s0, which is the port of the i210 HW for timestamping. See also Figure 46. These devices are set as defaults in the C-code with #defines. There is a function that scans devices, that are configured externally as VLAN interface. If no VLAN interface is configured, the coded default devices are used to create the VLAN interfaces. These changes are essential to scenarios, where the VLAN configuration is not given by the external network configuration as it is the case in the tests with raw-ethernet-sockets.

In the first approach to implement TSN communication into the TRDP-Stack the example application used IP multicast addressing as described in chapter 4. The filtering of the telegrams on the receiver side (different VLANs, different PD-telegrams …) increases the complexity and has influence on the timing. Therefore, unicast source and destination IP-addresses are configured for all test scenarios.

The tests are carried out with separate IP-addresses for TSN- and PD-packets. This provides the identification of the TSN-packets by IP-address rather than by their VLAN-ID. If the switch doesn’t detect the destination NIC’s VLAN configuration it strips the VLAN tag and priority, when forwarding the VLAN packets to the receiving NIC. The packet is then delivered as standard packet and not recognised as TSN-packet. The PD-packets, received by the same NIC, require a real standard ethernet interface. To distinguish between these two interfaces, the packets have different destination IP-addresses. The transmitter source IP-address is 10.0.1.200 for PD-packets and 10.0.10.200 for the TSN-packets.

The receiver IP-address is configured as 10.0.1.100 for PD-packets and 10.0.10.100 for TSN-packets.

The test scenario uses 2 VLAN IDs. The VLAN-ID 10 is provided for the TSN-packets and VLAN-ID 20 is configured for use in scenarios with additional bulk-traffic.

The VLAN tags required for the TSN-packets are configured for each switch. The NIC interfaces are either configured by the system configuration or by the TRDP-Stack at start-up.

Usually, the network configuration is done by the user in a command shell or by a configuration file at system level. In a system with multiple end-devices, it would be convenient that the protocol-stack itself is providing the setup. For the TSN implementation the parameters for the extra virtual interface are set by the TRDP-Stack when requesting the TSN socket. The parameters are the IP-address and the VLAN ID and priority. This can either be done by calling the system-commands or by using the Linux “ioctl” library.

The Kontron interface configuration scripts (configure_transmitter and configure_receiver) are modified to suit each test scenario.

5.1.2 Raw-ethernet-socket Implementation

The first approach to adapt the TRDP-Stack to TSN communication (chapter 4) used IP Raw sockets. Now, raw-ethernet-sockets are implemented for transmitting and receiving TSN-packets. On one hand this offers the ability to use the Kernel SW and HW timestamp functionality, that is provided by the Intel i210 network controller and on the other hand it is possible to compose and send packets that do not require any interface configuration. Especially for the TSN protocol the packets can be tagged with VLAN-ID and priority without changing the interface configuration. Also, the usage of the precise timing of the transmission according to the queue-discipline “Earliest TxTime First” (ETF) requires socket option SO_TXTIME that is only supported if raw-ethernet-sockets are applied. Also, hardware support is required for ETF. Although the i210 NIC is announced to support SO_TXTIME (see [13]), it was not possible to activate it according to tc-etf(8) of the Linux manual page [12].

5.1.3 Time Synchronisation

As explained in 3.1.2 the time is synchronised between PTP-clients and the systems CLOCK_REALTIME. Additionally, there is also a direct access to the network device “/dev/ptp3” implemented, that is provided by the ptp-daemon. Accessing this device would make the synchronisation to CLOCK_REALTIME or CLOCK_MONOTONIC obsolete, but there is a disadvantage when it comes to thread synchronisation. The clock_nanosleep() function requires a clock_id parameter and the provided ptp-device is not supported. Due to this the synchronisation with CLOCK_REALTIME is kept as time base for the thread scheduling.

5.1.4 Bulk-traffic Generation

Two compact RPi computers are used as transmitter and receiver of additional “Bulk”-traffic. The traffic is generated by the LINUX utility “iperf3”. It is a typical client-server application. The client is transmitting the packets to the listening server with a high network load. Between the client and the server, the bulk-traffic passes the two switches in the same direction as the TSN-packets. See the red connection of the KBoxes in Figure 46 and Figure 47.

5.1.5 Implementation of Timestamping and Excel-Export

The timestamps recorded at transmitting and receiving of the scheduled TSN traffic are written to a comma separated (.csv) file. This type of file is supported by the MS Excel import functionality. To perform the data postprocessing in Excel saves runtime at the reception of the TRDP-Stack. The application layer of the Stack provides a call-back function to interpret the protocol headers and finally store the timestamps into a buffer-array. In line with the Kontron example, after storing the data of 10000 packets, a loop is triggered to write the data to the .csv file. There is also some calculation necessary:

1. The timestamps are read as 32bit integer number representing the seconds plus a 32bit number representing the µs. These need to be written to file as ASCII string with a dot between them, and a comma for separation to the next timestamp.
2. Additionally, due to the limitation of Excel in representation of numbers up to 15 digits, the digits of seconds plus the 6 digits for µs are exceeding the limit and the last digit of the µs is mainly cut-off. To overcome this reduction of precision, the seconds of all timestamps are converted from absolute system-time to numbers related to the first interval-start timestamp as 0. Recording 10000 packets at 1ms cycle-time takes 10s. The number of required digits is reduced to 8, in total without any loss of information.

Excel is used to calculate the average latency of the packets as well as their mean jitter. Mean jitter is calculated as the mean deviation to the average latency.

For the measurements of latency and jitter, the TRDP-Stack is extended by the following timestamps that were intended to measure the same events of the transition of a packet as measured by the Kontron example.

The Kontron timestamps are mapped to the TRDP-Stack as follows:

1. Interval-Start-Time: Equally to the Kontron scheduling, the first timestamp is the calculated wake-up time when the cyclic thread shall return from the clock_nanosleep() function.
2. Wake-Up-Time: Immediately after wake-up from nanosleep state, this timestamp is recorded to measure the deviation of the real behaviour from the requested Interval-Start time. This timestamp is recorded in vos_runCyclicThread(), See also chapter 4.2.2.
3. Program Send Time: This is the last timestamp before entering the Linux Kernel by calling sendmsg() in vos_sockSendTSN(). See also chapter 4.1.5.4
4. Kernel-Shed-Tx-Time: This timestamp is taken by the Kernel right before the packet scheduler.

5. Kernel_SW_Tx Time, Kernel_HW_Tx Time: Unfortunately, the kernel transmit latency is not generated, because this timestamp is not working as described by the Linux kernel documentation.

   See also https://www.kernel.org/doc/Documentation/networking/timestamping.txt.
6. Kernel_HW_Rx Time: This timestamp is generated immediately by the network adapter HW, when receiving a packet.
7. Kernel_SW_Rx Time: This is the time, when the packet is passed from the device driver to the kernel receive stack.
8. ApplTime: In addition to the Kontron example, an extra timestamp is provided in the end of the reception stack at application level to be able to calculate the total latency and jitter of the TRDP-Stack. This is equivalent to the operation “getLocalTime” in Figure 43 and the calculation of latency and jitter shifted from the stack to the postprocessing in Excel.

The predefined cycle times for TSN packets are reduced from 10 ms to 1 ms cycle, also in alignment with the Kontron example. The cycle times for the non-real-time PD packets are reduced from 100 ms to 10 ms.

The scheduling of the VLAN priority 7 window for TSN-communication between the 0400TSN switches is taken over from the Kontron-example as described in 3.1.1. All time periods refer to the 1 ms Cycle-Start of the synchronized system time. The scheduling of the application real-time thread is also based on the same system time. In order to launch the TSN-packet as close as possible just before the switch enables the TSN-traffic, the real-time thread is started early enough to cover the transmit-latency and the transmit-jitter for the preparation of the TSN-packet by the TRDP/TSN stack. During the tests this preparation period is determined as a mean value of 120µs. An extra 80µs gives enough room to allow for jitter in the TRDP-Stack without missing the transmission window. Although, there is a very small statistically risk that a packet leaves the end device, too late. Throughout all tests it was never the case, that any outlier missed the transmission window. That means for these tests, the Interval-Start is set to 200µs before the switch puts the packet on the network and the schedule of the application-thread is 150µs before the 1ms Cycle-Start.

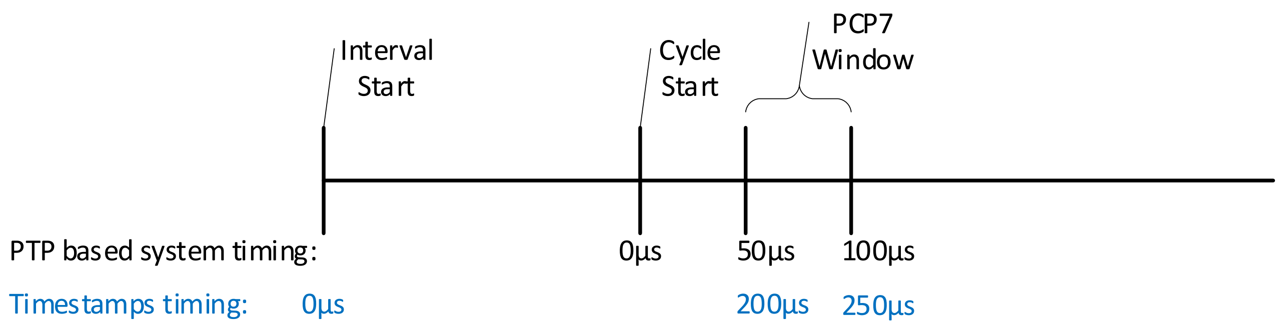


Figure 48: Different time bases for NIC cycle (PTP based system timing, used in chapter 3) vs. system interval (timestamps timing, used in this chapter)

So, the schedule in the TSN switches remains untouched to the tests in task 1 described in chapter 3. Therefore, all timestamps shown for task 2 are based on the interval-start time 150µs in advance to the cycle start of the network configuration. This means the VLAN priority 7 window in the diagrams of the test results under 5.2 starts at 200µs and ends at 250µs. This is shown with the overview of different time bases in Figure 48.

5.2 Test Results

The measurements were made using 10000 packets. For each packet the different timestamps were recorded to be evaluated at a later stage, to determine the latency and jitter of the communication. The post-processing and illustration of the results was done using Microsoft Excel. During the evaluation the first 350 packets were discarded, due to a transient effect that was observed during the early development of the tests but was fixed before the tests reported here.

The used Excel diagram-type is a boxplot. For each timestamp, the latency is plotted for each packet and that way, the whole extent from minimum to maximum latency is visible. Within the box the median line marks the value in the data set. The box lower end marks the first quartile, i.e., the median of the lower half of the dataset, respectively the upper end of the box marks the third quartile, i.e., the median of the upper half of the dataset. The two "whisker" lines are calculated based on the interquartile range IQR. This range is the difference between the first Q1 and the third quartile Q3. The lower whisker is defined as Q1 – 1.5 * IQR and the upper whisker respectively Q3 + 1.5 * IQR. The points which are exceeding the whiskers are called outliers. In situations, when there are no outliers, the whiskers are even decreased to mark the most extreme point of the data set. The small 'x' inside of the box is the mean latency of the received packets. Unfortunately, the mean jitter is not displayed in the diagram, but the quartiles and the whiskers give a good tendency of the situation. These diagram boxes and lines match the true mean values better, the smaller the jitter is.

Unfortunately, the socket option required for HW timestamping is not supported by IP-raw sockets. That means the TSN timing analysis is based on the SW timestamps only and the readings of these are expected to provide a lower level of accuracy than it will be by HW timestamping as done in scenarios with raw-ethernet sockets, as they include latency and jitter accrued in the SW-stack.

5.2.1 Set-up with sending directly from switch using IP-raw-sockets without bulk-traffic.

This setup shall demonstrate the TRDP-Stack that establishes a TSN-connection between the two Kontron switches and the receiving switch forwards the packets to the i210 NIC. TSN-packets are transferred in parallel to the non-real-time PD packets, but no bulk-traffic. PD-packets are sent and received by IP-sockets, while TSN-packets are received by IP-sockets, but sent by raw-IP-sockets.

Figure 49 gives the results of this setup. Due to the lack of the HW timestamp the network performance cannot be observed in isolation. Similarly, as in the Kontron example, the results were produced after 10000 TSN messages sent every 1ms.

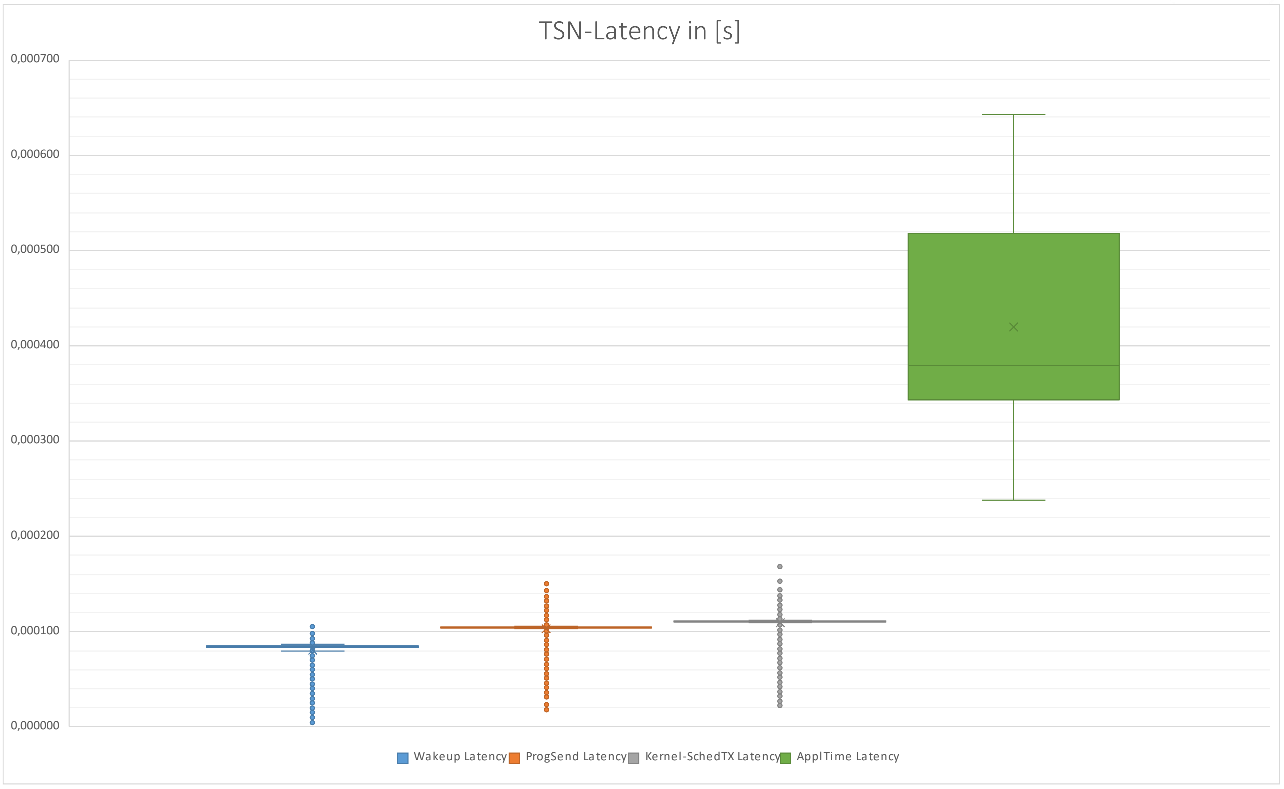


Figure 49: Set-up with sending directly from switch using IP-raw-sockets without bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Wakeup	80	6
Program Send	103	4
Kernel-SchedTX	109	4
Application RX	420	86

Table 6: Set-up with sending directly from switch using IP-raw-sockets without bulk-traffic.

In the scale on the left of the diagrams the µs are given and within that the window for enabling of the TSN-packets starts at 200µs and ends at 250µs.That means there is an average latency of 220µs of the receiving stack and an overall average latency of 420µs with a jitter of 86µs.

5.2.2 Set-up with sending directly from switch using IP-raw-sockets plus bulk-traffic.

The second setup is based upon the same configuration as the first one, but additionally external bulk-traffic is applied to packet transfer from one switch to another.

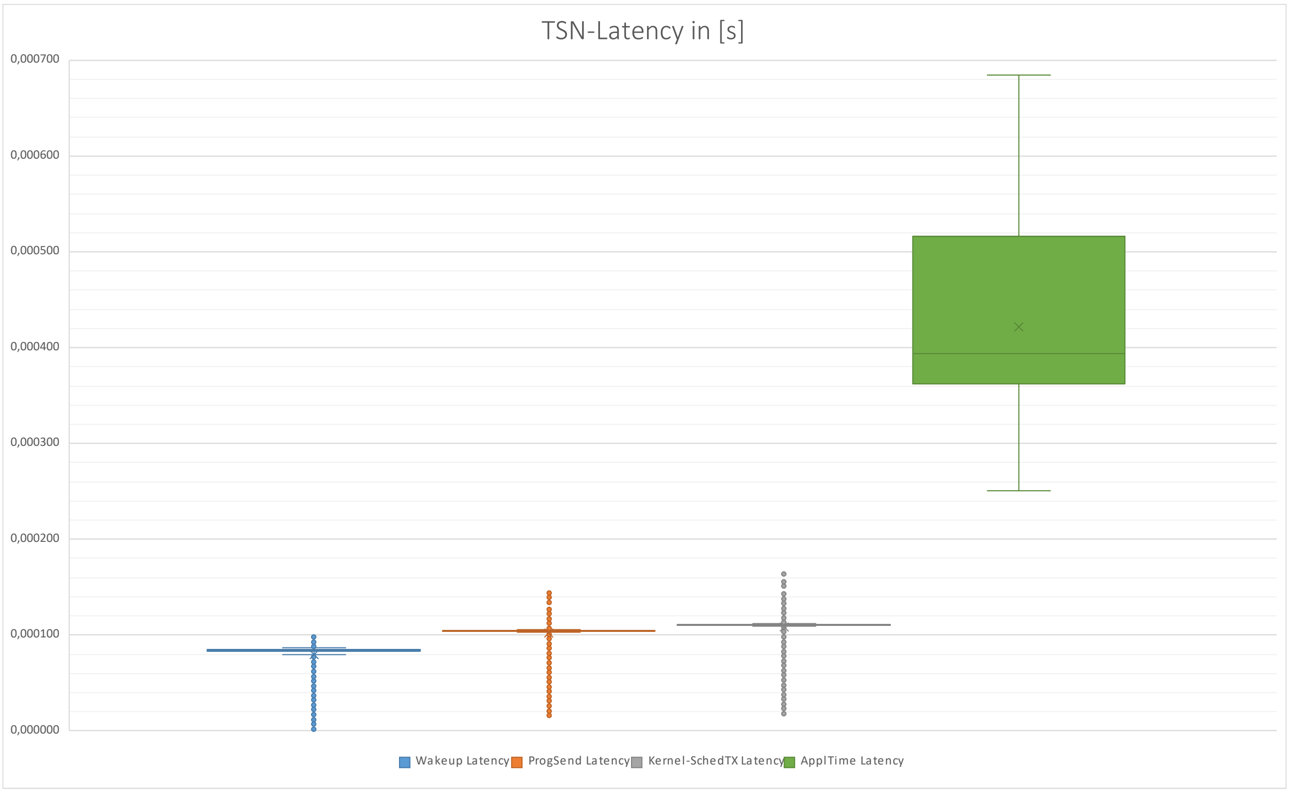


Figure 50: Set-up with sending directly from switch using IP-raw-sockets plus bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Wakeup	80	7
Program Send	102	5
Kernel-SchedTX	108	5
Application RX	422	68

Table 7: Set-up with sending directly from switch using IP-raw-sockets plus bulk-traffic.

Comparing the resulting timestamps of these TSN-packets to the first scenario shows, that the low priority bulk-traffic is not affecting the overall latency or jitter. Conceptually there is no disturbance possible to the software as the bulk-traffic is generated externally by single-board PCs. The transmitting part of the TRDP-Stack is running exclusively on one Kontron K-Box and the receiving part is not running as real-time thread but also running exclusively on the other K-Box.

5.2.3 Set-up with sending directly from switch using Ethernet-raw-sockets without bulk-traffic.

The following 2 scenarios with HW-Timestamps will show the robustness of the TSN-transmission across the link of the two switch-boards. Figure 51 shows the timing results without bulk-traffic.

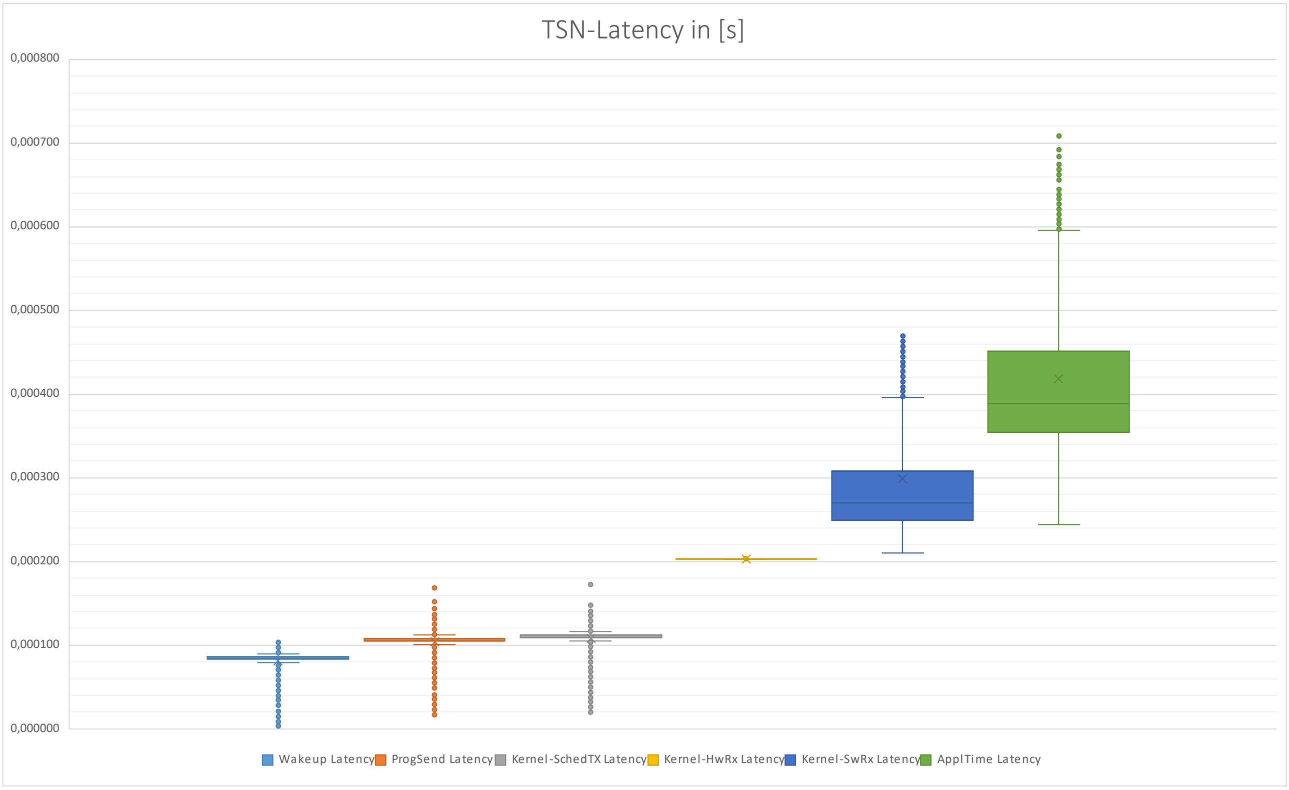


Figure 51: Set-up with sending directly from switch using Ethernet-raw-sockets without bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Wakeup	82	7
Program Send	104	5
Kernel-SchedTX	108	5
Kernel-HwRx	203	<1
Kernel-SwRx	299	64
Application RX	418	75

Table 8: Set-up with sending directly from switch using Ethernet-raw-sockets without bulk-traffic.

With the Kernel-HwRx timestamps the reception times of the TSN-packets are read with high accuracy. In Table 8, the mean jitter is below 1µs. The latency of this timestamp is always 203µs after Interval-Start timestamp and the packets are put on the network by the transmitting switch at 200µs after Interval-Start. The latency from switch to switch and forward to the i210 NIC is 3µs. This is comparable to the timing of the Kontron example, and it is expected, because the same HW chain is involved.

5.2.4 Set-up with sending directly from switch using Ethernet-raw-sockets plus bulk-traffic.

This test is the same configuration as 5.2.3 with additional bulk-traffic. It shall demonstrate the robustness of the TSN communication against parallel bulk-traffic at maximum rate.

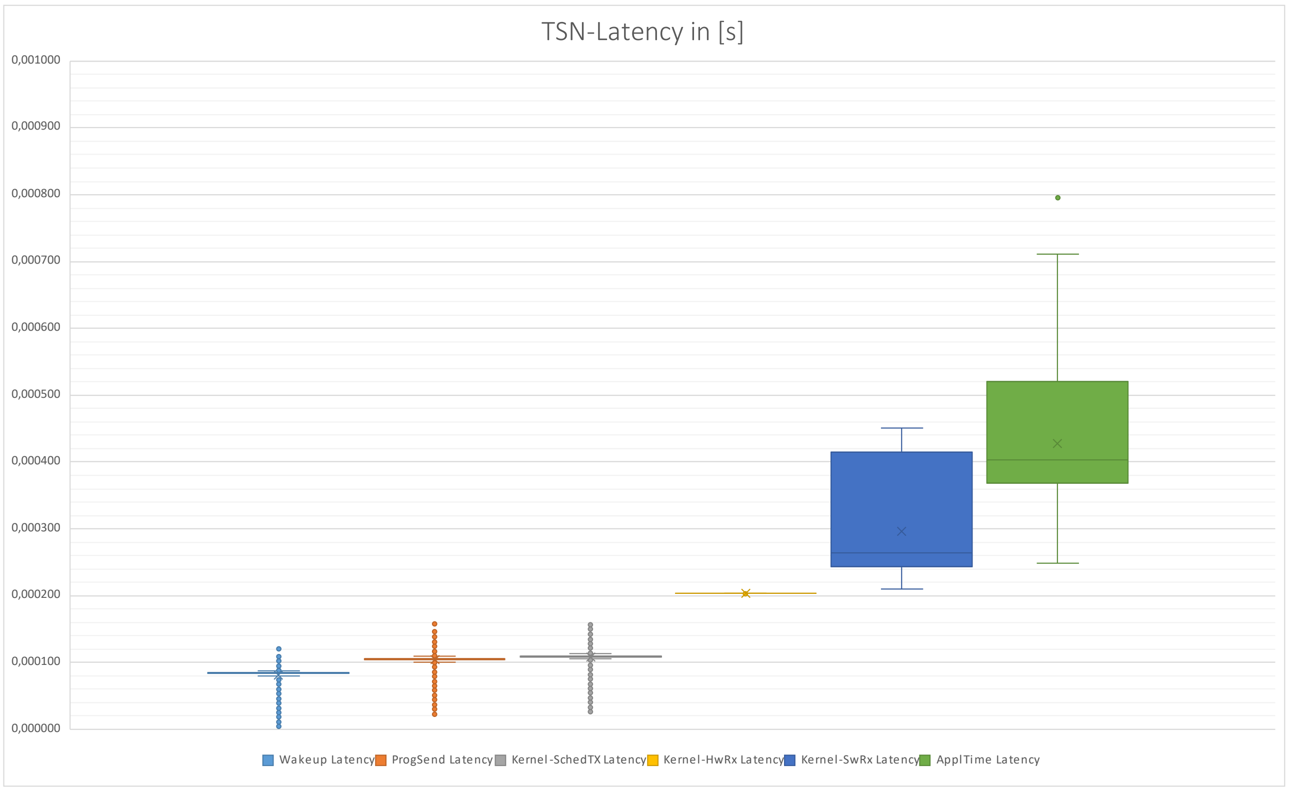


Figure 52: Set-up with sending directly from switch using Ethernet-raw-sockets plus bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Wakeup	81	6
Program Send	104	5
Kernel-SchedTX	108	5
Kernel-HwRx	203	<1
Kernel-SwRx	296	66
Application RX	428	68

Table 9: Set-up with sending directly from switch using Ethernet-raw-sockets plus bulk-traffic.

The Mean Latency and the Mean Jitter of the Kernel-HwRx timestamp are the same as in test 5.2.3 within the measurement precision of 1µs. This proves that the additional bulk-traffic is not interfering the TSN-transmission at all.

5.2.5 Setup with sending directly from i210-ED using IP-raw-sockets without bulk-traffic.

The following scenarios demonstrate the full link from one i210 interface across 2 switches to the other i210 interface. Apart from this, the configuration equals 5.2.1.

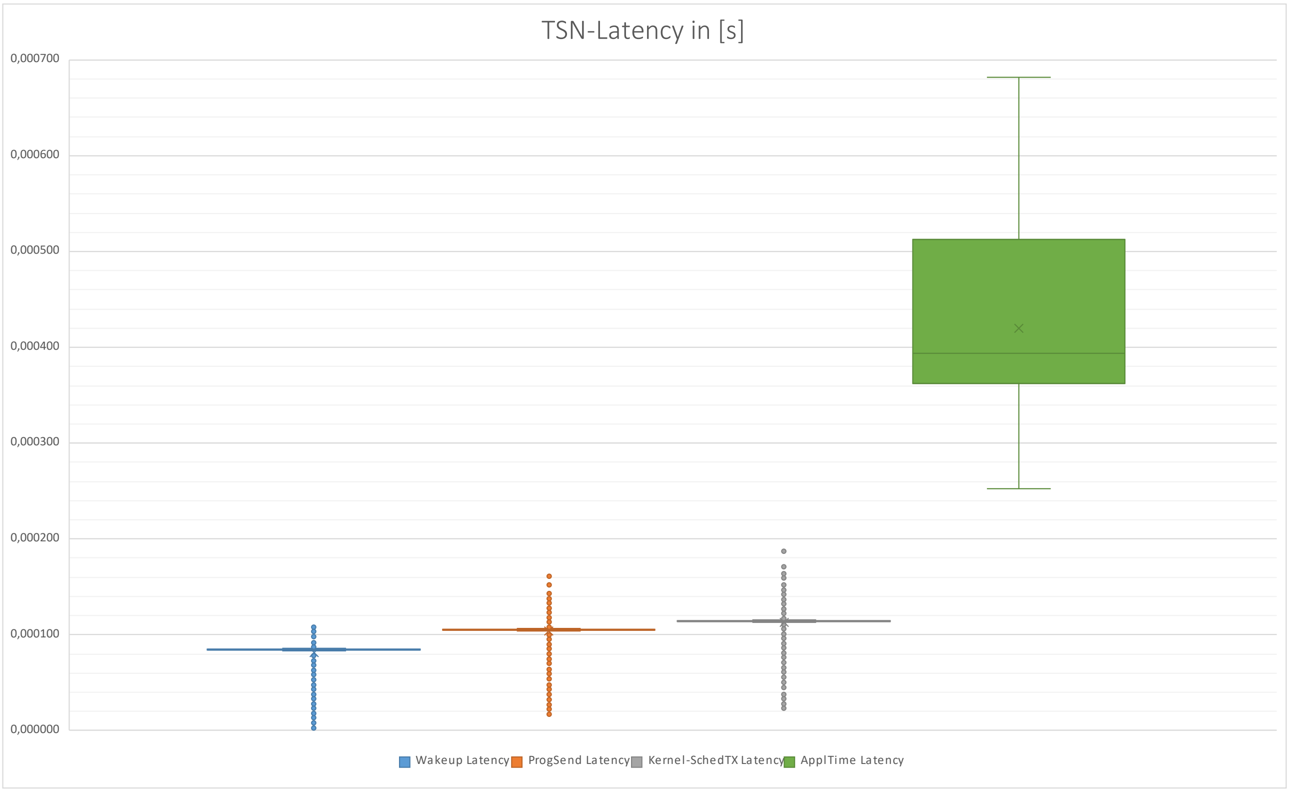


Figure 53: Setup with sending directly from i210-ED using IP-raw-sockets without bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Wakeup	81	6
Program Send	104	4
Kernel-SchedTX	113	4
Application RX	420	66

Table 10: Setup with sending directly from i210-ED using IP-raw-sockets without bulk-traffic.

5.2.6 Setup with sending directly from i210-ED using IP-raw-sockets plus bulk-traffic.

This is the same test configuration as 5.2.5, but with extra bulk-traffic.

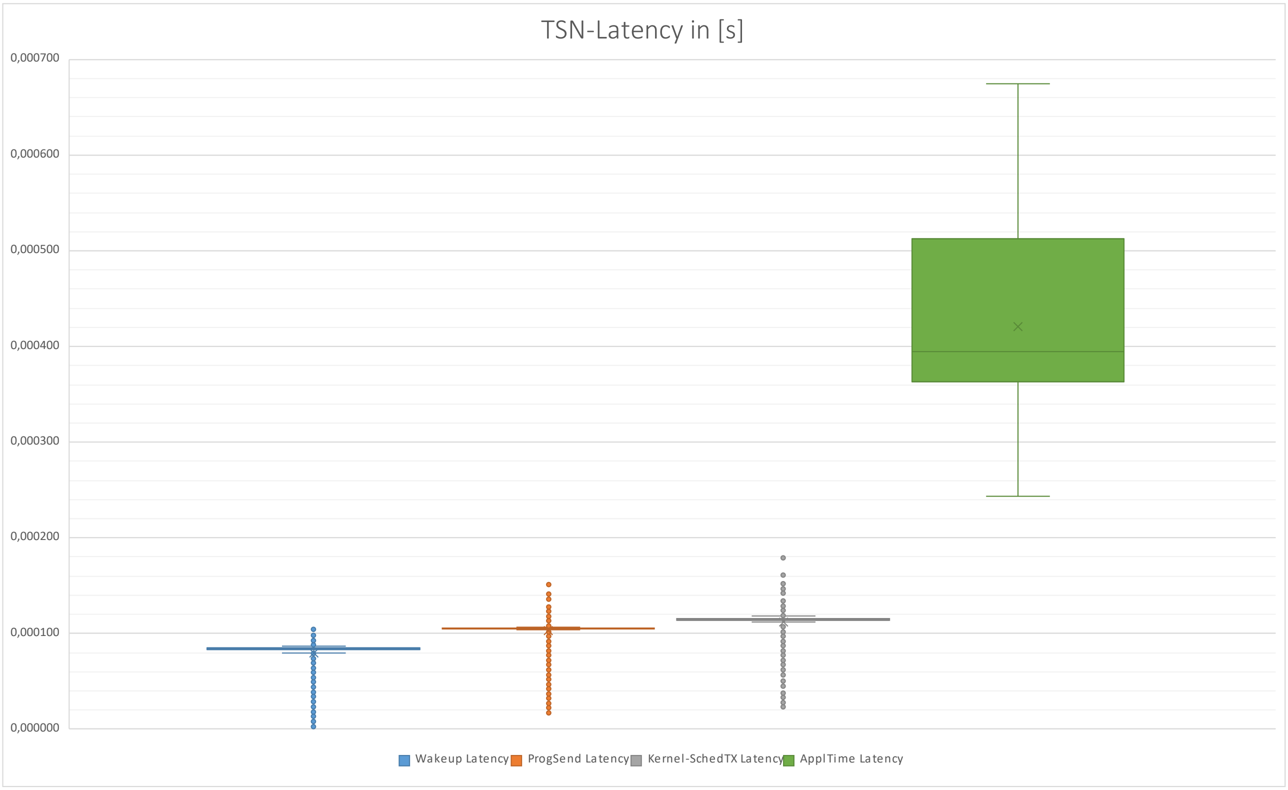


Figure 54: Setup with sending directly from i210-ED using IP-raw-sockets plus bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Wakeup	80	8
Program Send	103	6
Kernel-SchedTX	112	6
Application RX	421	67

Table 11: Setup with sending directly from i210-ED using IP-raw-sockets plus bulk-traffic.

This configuration doesn’t show any significant deviations from other tests with IP-raw-sockets. Also, there is no noticeable delay with the transmission from the i210 adapter in comparison to the direct transmission from the switch.

5.2.7 Setup with sending directly from i210-ED using Ethernet-raw-sockets without bulk-traffic.

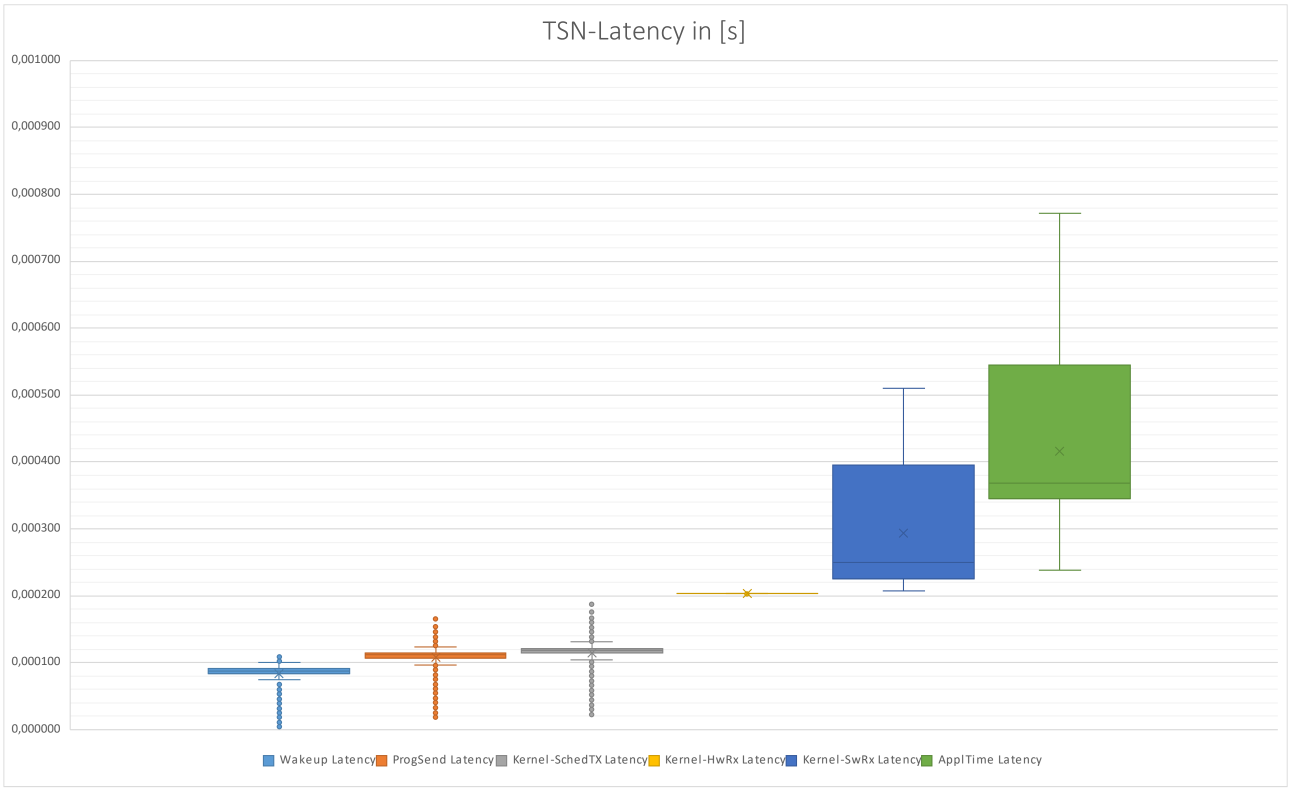


Figure 55: Setup with sending directly from i210-ED using Ethernet-raw-sockets without bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Wakeup	83	9
Program Send	108	8
Kernel-SchedTX	115	8
Kernel-HwRx	203	<1
Kernel-SwRx	294	78
Application RX	416	87

Table 12: Setup with sending directly from i210-ED using Ethernet-raw-sockets without bulk-traffic.

Due to the lack of a hardware timestamp on transmission side, an exact delay, that is caused by the extra link between the i210 adapter and the first switch, is not detectable. The delay is estimated to be in the range of 1µs or below and is usually not causing the transmission stack to exceed the 200 µs period. The Kernel-HwRx timestamp is as stable as in 5.2.3 without the extra link.

5.2.8 Setup with sending directly from i210-ED using Ethernet-raw-sockets plus bulk-traffic.

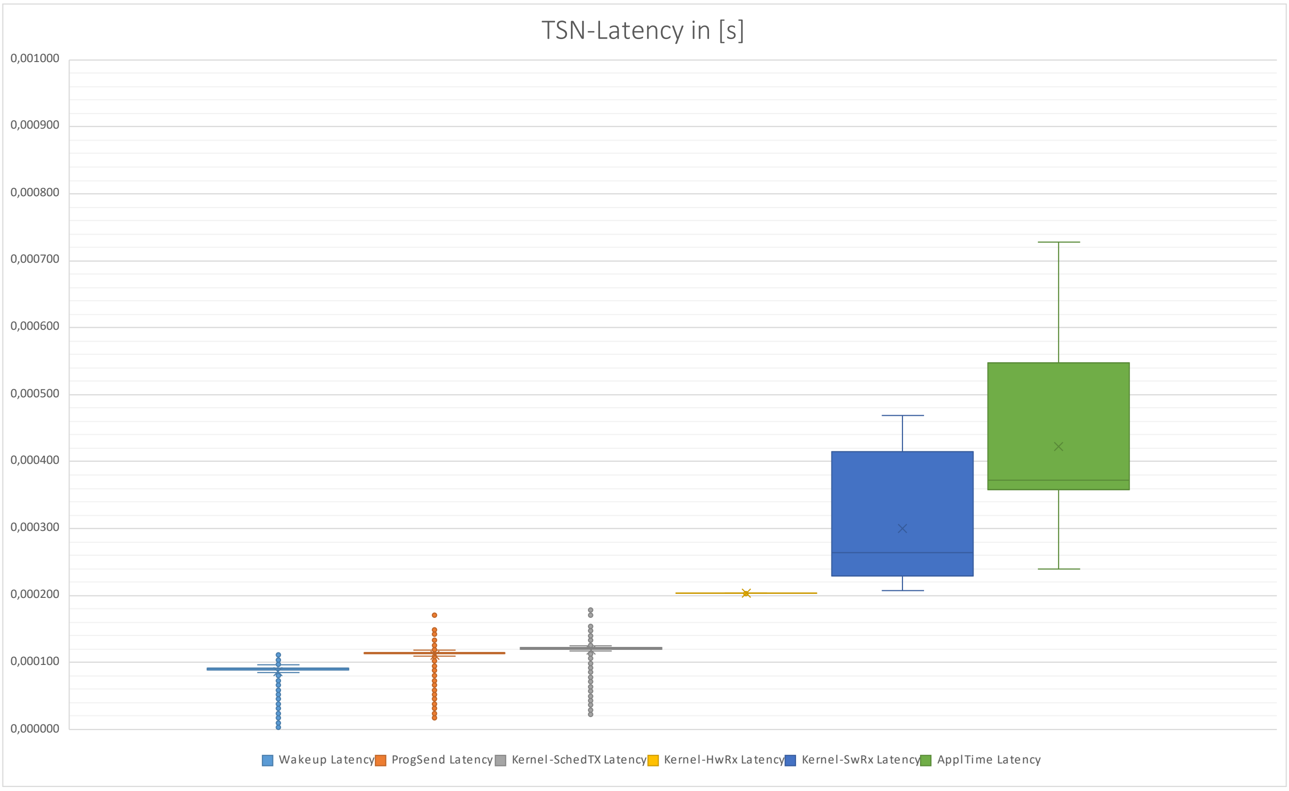


Figure 56: Setup with sending directly from i210-ED using Ethernet-raw-sockets plus bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Wakeup	86	8
Program Send	111	7
Kernel-SchedTX	118	6
Kernel-HwRx	203	0
Kernel-SwRx	300	78
Application RX	423	84

Table 13: Setup with sending directly from i210-ED using Ethernet-raw-sockets plus bulk-traffic.

In this configuration, the bulk-traffic doesn’t cause any significant deviations from other tests without bulk-traffic. Also, there is no noticeable delay with the transmission from the i210 adapter in comparison to the direct transmission from the switch.

5.3 Conclusion Task 2

Overall, the different test scenarios are quite consistent. Variations in latencies of the different packets due to the SW processing, which can be attributed to performance and timing variation of the operating system and the hardware, were observed in the form of large contributions to the overall jitter.

There are common effects discovered during the tests, such as a large wake-up latency, which disturb the analysis of the TRDP and TSN timing in isolation. These are discussed here.

The mean latency of the thread wake-up-time is 82 µs with respect to the interval-start-time. This is a lot larger compared to the wake-up-time of the Kontron examples 2.5 µs. This is the case throughout all the TRDP tests, although it is implemented almost in the same way: The clock_nanosleep() is set to the desired interval-starttime and on wake-up from sleeping the wake-up-time is registered.

It can be noticed that the thread scheduling on the reception side has larger jitter values, because there is no high priority real-time thread provided.

The TRDP-Stack is not yet optimized to operate on the hardware and operating system. Particularly, the thread-scheduling offers large potential for improvements to the thread-wakeup latency.. A deeper analysis of the difference of this aspect between the Kontron example and the TRDP-Stack could solve this issue.

On the reception side it can be assumed that the thread-scheduling offers reductions in latency and jitter by using a real-time thread. The current implementation is designed to receive the PD packets of multiple subscribers (TSN or non-TSN), each of them having its own cycle. First, the queue is searched for a suitable socket and in a second step the queue is searched again for a packet, that is ready for processing by the application. Another function is examining the queue for overdue packets.

In more detail this means that a receive queue ensures the treatment of the packets before an individually defined timeout expires. A handler who walks through the receive lists, predefined by the different subscribers, determines the maximum idle time. The POSIX select function is used to wait until the idle time is exceeded or any socket descriptor is ready. Afterwards the socket descriptor is checked whether it is one of the sockets created by the application. This socket is finally used for reading. With the current implementation of the TRDP-Stack there is no special “by-pass” for the TSN-PD-telegrams.

Beside the inadequate thread scheduling in terms of TSN objectives, there is no disturbance by the bulk-traffic observed. The hardware timestamp Kernel-HwRx is very stable and results in the same latency (203µs) at every test repetition. It demonstrates very well, that the configured PCP7 window is kept by the FPGA hardware of the 0400TSN switches.

Comparing the test-setups with either the i210-ED or the TSN switch as the transmission interface, there is not a significant difference in the transmission timing. The extra latency, that is expected with the extra link between the end device and the switch is equalized by the FPGA hardware, that buffers the packets until the transmission timeslot for PCP7 is enabled. The exact latency can’t be determined because the hardware TX timestamp at the i210-ED can’t be activated.

The raw Ethernet sockets show a comparable latency of the TSN-packets to the raw IP sockets. The variation between the scenarios with raw Ethernet sockets seem a bit larger, which might be attributed to the manual treatment of the Ethernet header and VLAN tag. However, in order to conclude this for certain, a larger number of tests would be needed.

The integration of the TSN functionality in the TRDP-Stack required mainly the adaption of the “vos” layer to the used hardware. TSN uses VLAN IDs for identification. This requires a change in the addressing concept of the participating devices in a TSN-network. Either separate IP-addresses are required for VLAN interfaces as it is applied in these tests, or the OS dependent interfaces need to be used as configuration parameters. If multiple VLANs shall be addressed and the network configuration shall be kept simple, raw ethernet sockets will be required in combination with special interface configuration such as bulk VLAN. The integration of these hardware features into the operating-system is a demand for future development, to enable the use of high layer sockets, which provide better performance of filtering TSN specific parameters and increase the independence from the hardware details.

For the synchronization of the application to the PCP7 window, that is configured at the switches, at least a parameter shall be provided that defines the latency of the transmission cycle. This latency is dependent on the performance of the hardware, the operating system and the implementation of the TRDP-Stack.

This Proof-of-Concept also shows that some of the functions in use e.g., the “ioctl” library inside the Ethernet-raw-sockets cannot be adapted in a generic way within different LINUX distributions. In this case the necessary provisions were made for “Fedora-LINUX”. In addition, some configurations depend on the chip set in use.

The whole concept of TSN is still in a flow. The standards are gradually enhanced and adapted due to the experience collected with real implementations. There are diverse activities ongoing for the improvement of chip sets supporting TSN as well as for the enhancements which can be expected within the LINUX community or with other operating systems. Therefore, also the TCNOpen TRDP-Stack will be improved with increasing provisions by the hardware and operating systems as well as the possibility to examine further, larger test set-ups.

This CCN PoC shows very well that scheduled traffic according to IEEE 802.1Qbv can be realized using TSN-TRDP-telegrams via switches supporting scheduling.

6 Task 3: End-to-End Standard Ethernet Connection with TCNOpen TRDP

Due to the re-evaluation [8] caused by the newly established SS-147 intended to be published as part of the TSI CCS 2022 the PD-packet timing transferred by the standard ethernet connection is analysed. To be able to compare the results with and without TSN, the TRDP-transmit and -receive stacks are executed on the same hardware, same operating system, and same configurations as the TSN-connection in chapter 5.

Additional timestamps for PD-packets are implemented into the program sequences that are comparable to the timestamps of the TSN-packets. As the HW-timestamps only work in combination with raw-ethernet-sockets, but standard PD-packets make use of IP sockets, the PD-packets only provide the “Wakeup” timestamp, the “ProgSend” timestamp at transmitter and the “ApplRxTime” at the receiver.

As described in chapter 4.1.4 a send queue is applied for PD-packets. Contrary to the TSN-packets, the payload of the PD-packets is provided asynchronously to the cyclic transmission times of these packets.

In a pre-test scenario, the wakeup timestamp is implemented at application level. It is registered at the time, when the packet-data is refreshed by tlp_put(). Although this update of the packet data is done cyclically in this pre-test application, it is not synchronized to the low level send cycle. This results in a mean jitter, as big as half of the cycle-time, as the time of the packet-data generation is equally distributed relative to the time of sending the packets out of the send queue on the network. As this effect spoils all subsequent timing, the activity of updating the packet data with tlp_put() is ignored and the first timestamp considered for the following tests with PD-packets is taken when the low-level thread, that is fetching the packets from the send-queue, is waking-up from sleep.

As well as in chapter 5, each test is repeated with additionally applied bulk-traffic, that is generated externally with VLAN 20 and PCP1. See chapter 5.1.4 for details.

Two scenarios are applied:

1. PD-Packets with timestamps and no TSN-Packets in parallel.
2. TSN-Packets sent with VLAN priority 6 (time-scheduling according to IEEE 802.1Qbv has open gates for any priority at any point in time and thus does not affect this traffic). All schedule settings in the original file “Scheduler.cfg” from Kontron are adapted to allow all priority values throughout the cycle and the network configurations as described in chapter 5.1.1, that include this file, shall be executed again. To ensure, that obsolete settings do not remain from previous configurations, a reboot of the PCs is recommended, in the first place.

6.1 Tests with PD-Packets without TSN-Packets in parallel

6.1.1 Test Configuration

This scenario is run using the following two application example files, which were modified accordingly:

- `sendPDtimestamp` (from sendTSN)
- `receivePDtimestamp` (from receiveTSN)

The publishing of the TSN-packets is removed as well as the call of the dataAppThread function.

The cycle time for the PD-packets is kept at 10ms for publishing, while the cycle time of the comThread is reduced from 10ms to 1ms. The statistics in the diagrams are based upon a 100s period that corresponds to the transmission of 10000 packets.

The Wakeup timestamp is taken and added to the PDs payload in trdp_pdSendQueued() just before the call to trdp_pdSend(). In trdp_pdSend() the ProgSendTime timestamp is taken and added to the PDs payload just before the call to vos_sockSendUDP() function. Both timestamps are implemented in the trdp_pdcom.c file and the access to them is simplified by casting of the pPacket parameter of trdp_pdSend().

The ApplRxTime timestamp is taken for the PD-packets in the same way as it is done with ApplTime for the TSN-packets. See also chapter 5.1.5.

After the reception of 10000 packets, all data is recorded in the file MsgTimingPD.txt.

The PD-packets are sent with QoS priority 3. The bulk-traffic is generated as explained in chapter 5.1.4 and uses QoS priority 1.

The latency diagrams have the Wakeup time as reference (0 s). The two latencies ProgSendTime and ApplRxTime are given relative to the Wakeup time.

6.1.2 Test Results

6.1.2.1 Set-up with PD sending directly from switch without bulk-traffic.

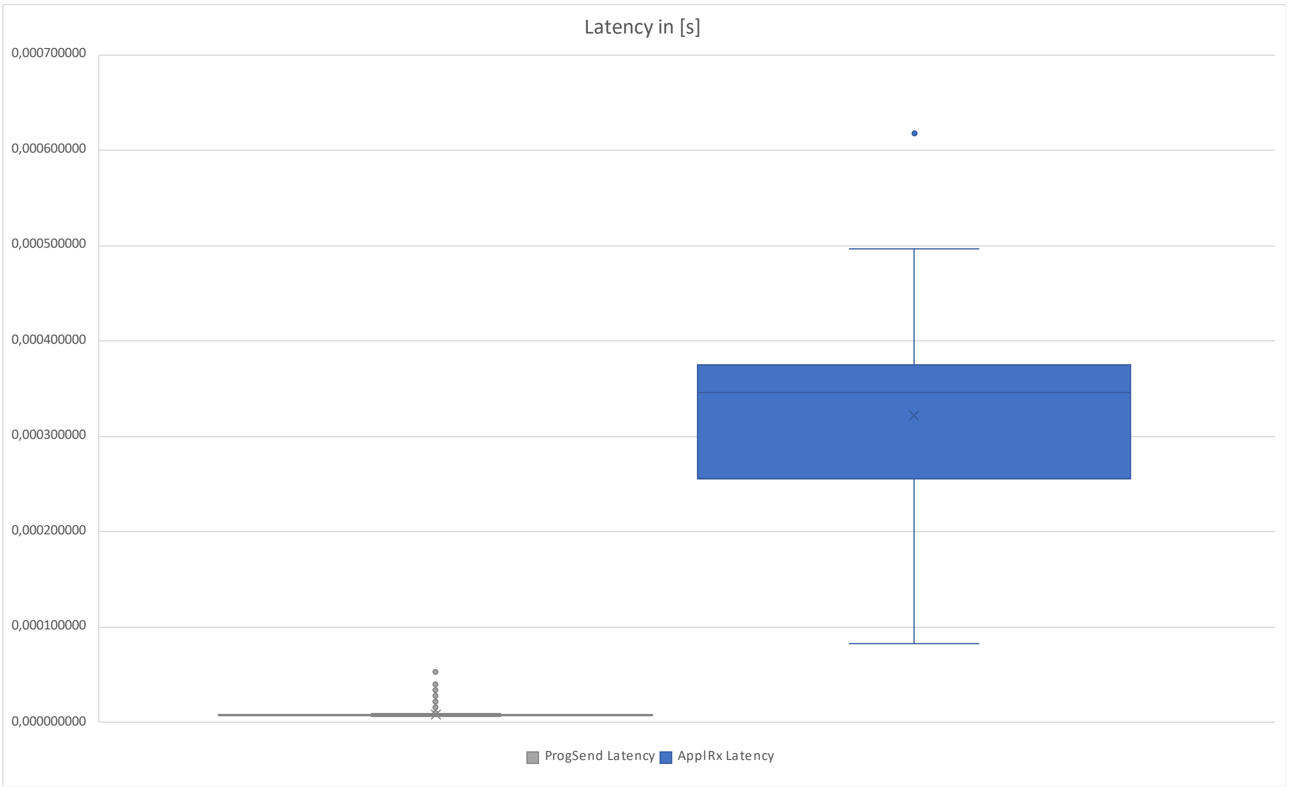


Figure 57: Set-up PD with sending directly from switch without bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Program Send	9	2
Application RX	322	65

Table 14: Set-up PD with sending directly from switch without bulk-traffic.

6.1.2.2 Set-up with PD sending directly from switch plus bulk-traffic.

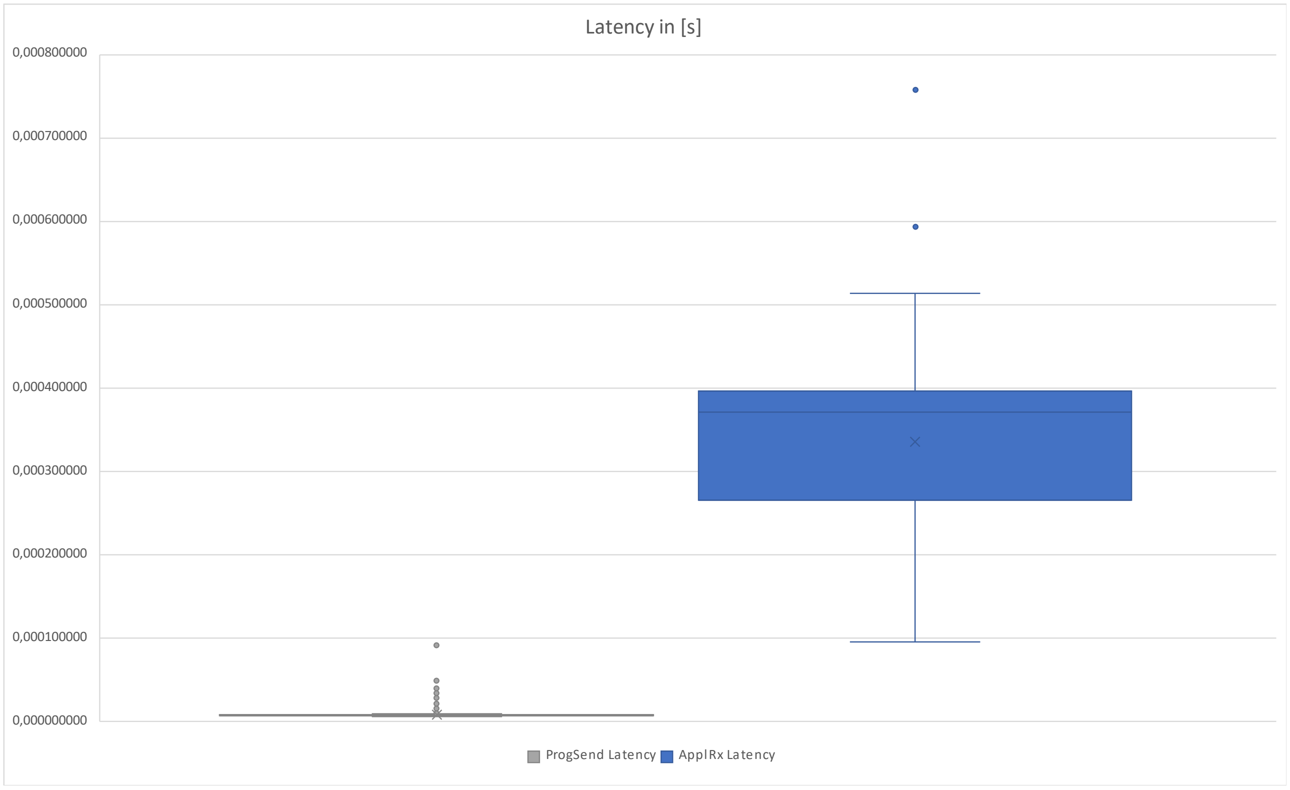


Figure 58: Set-up PD with sending directly from switch plus bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Program Send	9	2
Application RX	335	69

Table 15: Set-up PD with sending directly from switch plus bulk-traffic.

6.1.2.3 Set-up with PD sending directly from i210-ED without bulk-traffic.

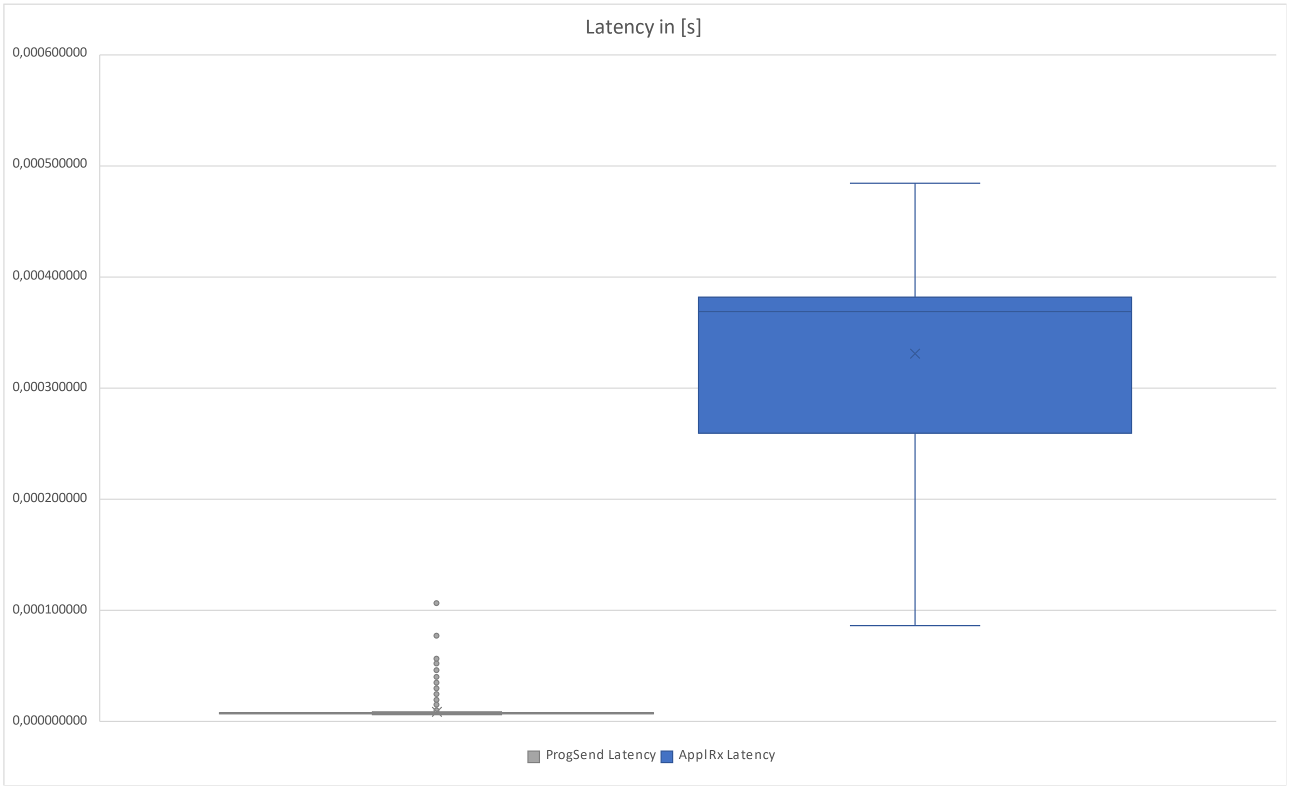


Figure 59: Set-up PD with sending directly from i210-ED without bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Program Send	9	2
Application RX	331	66

Table 16: Set-up PD with sending directly from i210-ED without bulk-traffic.

6.1.2.4 Set-up with PD sending directly from i210-ED plus bulk-traffic.

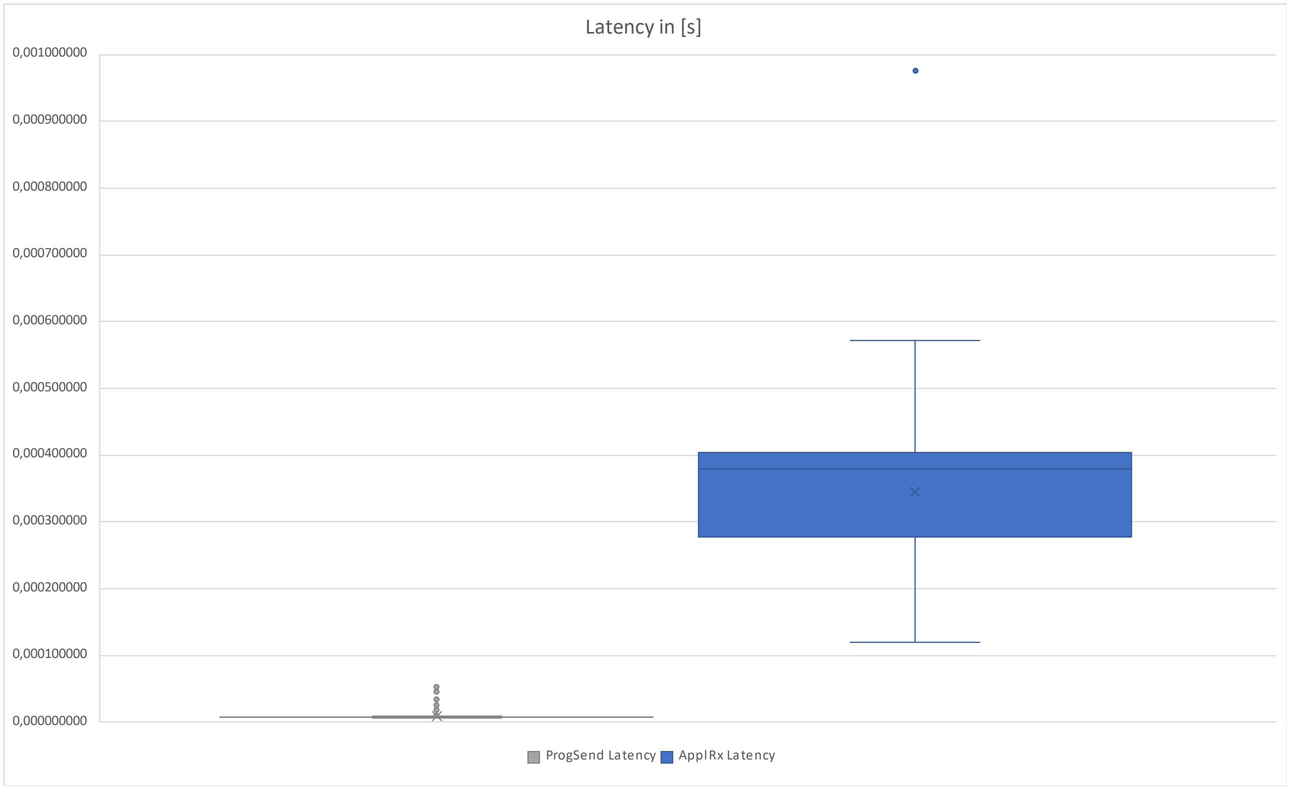


Figure 60: Set-up PD with sending directly from i210-ED plus bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Program Send	9	2
Application RX	345	67

Table 17: Set-up PD with sending directly from i210-ED plus bulk-traffic.

6.1.2.5 Conclusion Task 3.1

To compare the latencies of the PD-packets to those of the TSN-PD-packets, the Wakeup time of the TSN-PD packets shall be ignored. So, all the latencies of the TSN-PD packets shall be reduced by the Wakeup time. In the two following tables the (reduced) mean latencies of Program Send and Application RX for TSN-PD and normal PD packets are compared.

Chapter of TSN set-up	Reduced Program Send TSN-PD [µs]	Program Send PD [µs]
5.2.1	103-80 = 23	9
5.2.2	102-80 = 22	
5.2.3	104-82 = 22	
5.2.4	104-81 = 23	
5.2.5	104-81 = 23	
5.2.6	103-80 = 23	
5.2.7	108-83 = 25	
5.2.8	111-86 = 25	

Table 18: Overview of the mean latencies, reduced by wakeup time.

The latency for sending TSN-packets is about 14µs longer than the latency of the PD-packets. The main reason for this is the extra treatment of the additional 2 Kernel timestamps, that are read back from the socket before sending the packet. This applies to both socket types (raw ethernet and raw IP-sockets).

Chapter of TSN set-up	Reduced Application RX TSN-PD [µs]	Application RX PD [µs]	Application RX difference between TSN-PD and PD [µs]
5.2.1	420-80 = 360	322	38
5.2.2	422-80 = 362	335	27
5.2.3	418-82 = 356	322	34
5.2.4	428-81 = 367	335	32
5.2.5	420-81 = 359	331	28
5.2.6	421-80 = 361	345	16
5.2.7	416-83 = 353	331	22
5.2.8	423-86 = 357	345	12

Table 19: Overview of the application receive times, reduced by the wakeup time in the case of TSN-PD.

The values in the column “Application RX PD” of Table 19 are sorted to match the TSN test cases in the way, that the value of each PD test is opposed to the test with raw-IP-socket as well as to the test with raw-ethernet-socket. All tests with PD packets use the UDP-socket.

Although there is an extra waiting time (mean value of about 90 µs) in the TSN-stack for buffering the TSN-PD packets by the TSN-switch until the PCP7 gate opens, the difference in Application RX latency between TSN-PD packets and normal PD packets is quite small. The difference is even smaller than the extra waiting time caused by the buffering. One reason can be the unpredictable impact of the receive stacks as discussed in 5.3 although the implementation on the receiver side does not distinguish between the type of the packets or their priority. Another reason can be on the sending side where PD packets do not have the same high priority as the TSN-PD packets.

6.2 Timestamps of TSN-Packets with VLAN priority 6 (no time-scheduling (IEEE 802.1Qbv))

6.2.1 Test Configuration

This test is carried out in the same way, as the TSN-tests in chapter 5 with the differences, that the VLAN priority is changed from 7 to 6 and the schedule is adapted as explained above. Consequently, the TSN-Packets are not scheduled by the switch card. The priority change is done by parameter when starting the sender stack with -p 6.

The application example files are sendTSN and receiveTSN – the same as in chapter 5, as well as all other configurations and setups.

6.2.2 Test Results

6.2.2.1 Set-up with TSN priority 6 sending directly from switch without bulk-traffic.

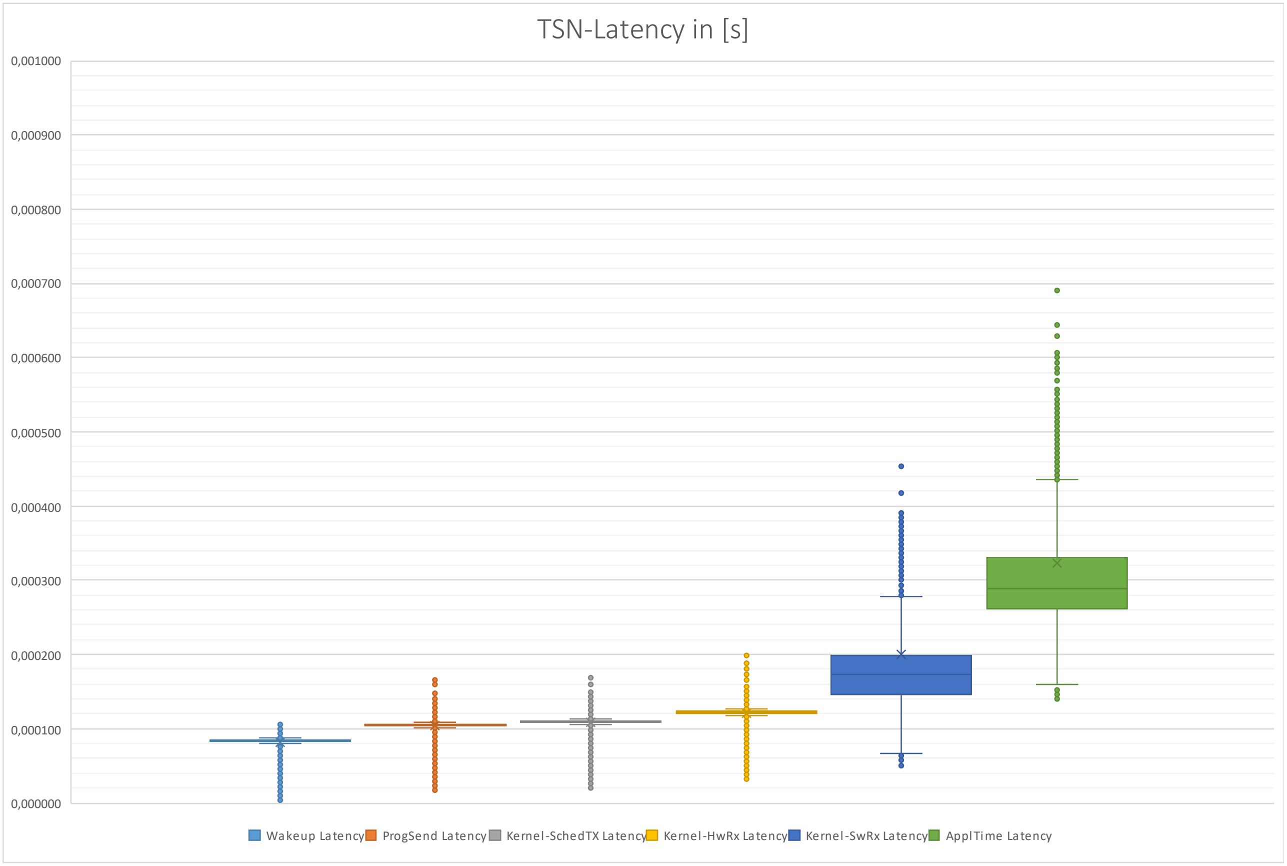


Figure 61: Set-up with TSN priority 6 sending directly from switch without bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Wakeup	82	6
Program Send	104	4
Kernel-SchedTX	108	121
Kernel-HwRx	121	4
Kernel-SwRx	200	60
Application RX	324	77

Table 20: Set-up with TSN priority 6 sending directly from switch without bulk-traffic.

6.2.2.2 Set-up with TSN priority 6 sending directly from switch plus bulk-traffic.

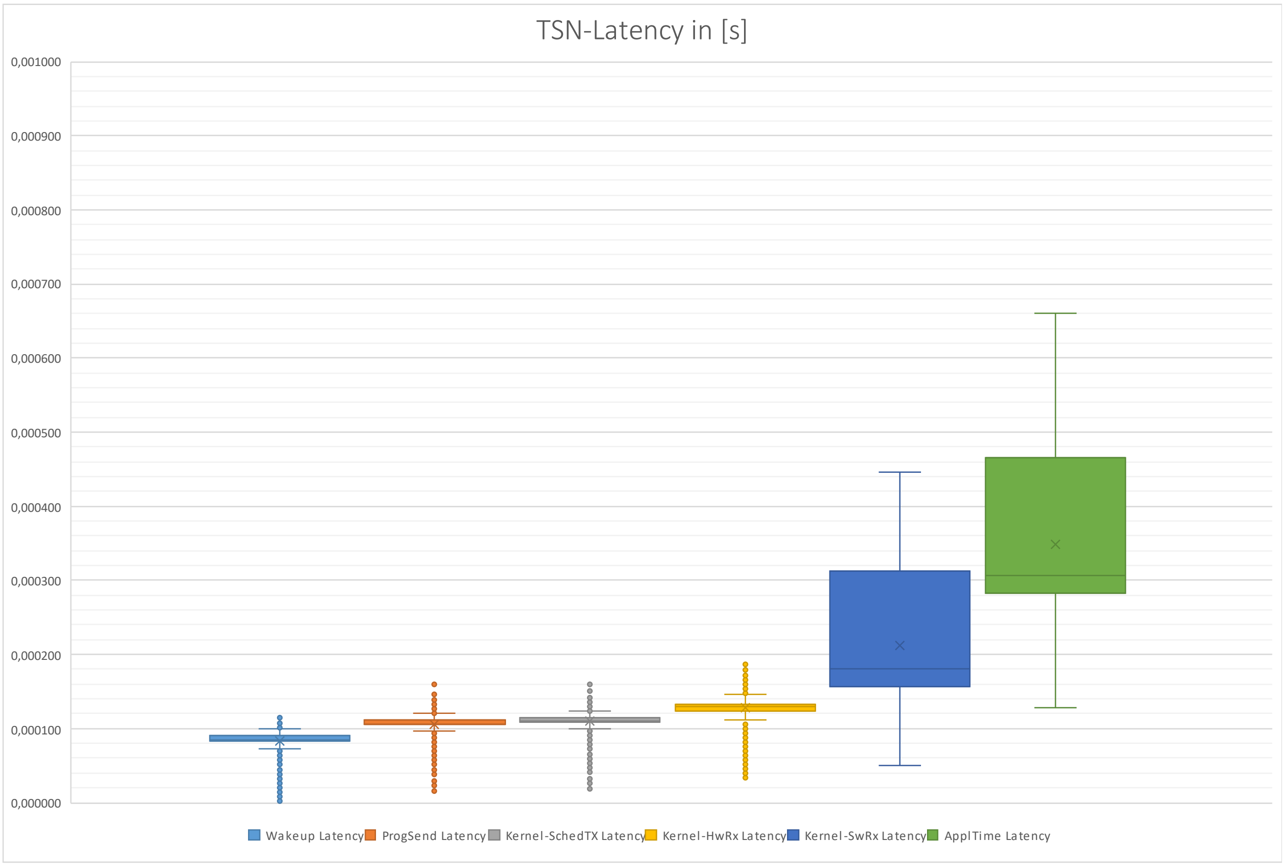


Figure 62: Set-up with TSN priority 6 sending directly from switch plus bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Wakeup	83	6
Program Send	106	5
Kernel-SchedTX	110	5
Kernel-HwRx	128	7
Kernel-SwRx	212	66
Application RX	349	85

Table 21: Set-up with TSN priority 6 sending directly from switch plus bulk-traffic.

6.2.2.3 Set-up with TSN priority 6 sending directly from i210-ED without bulk-traffic.

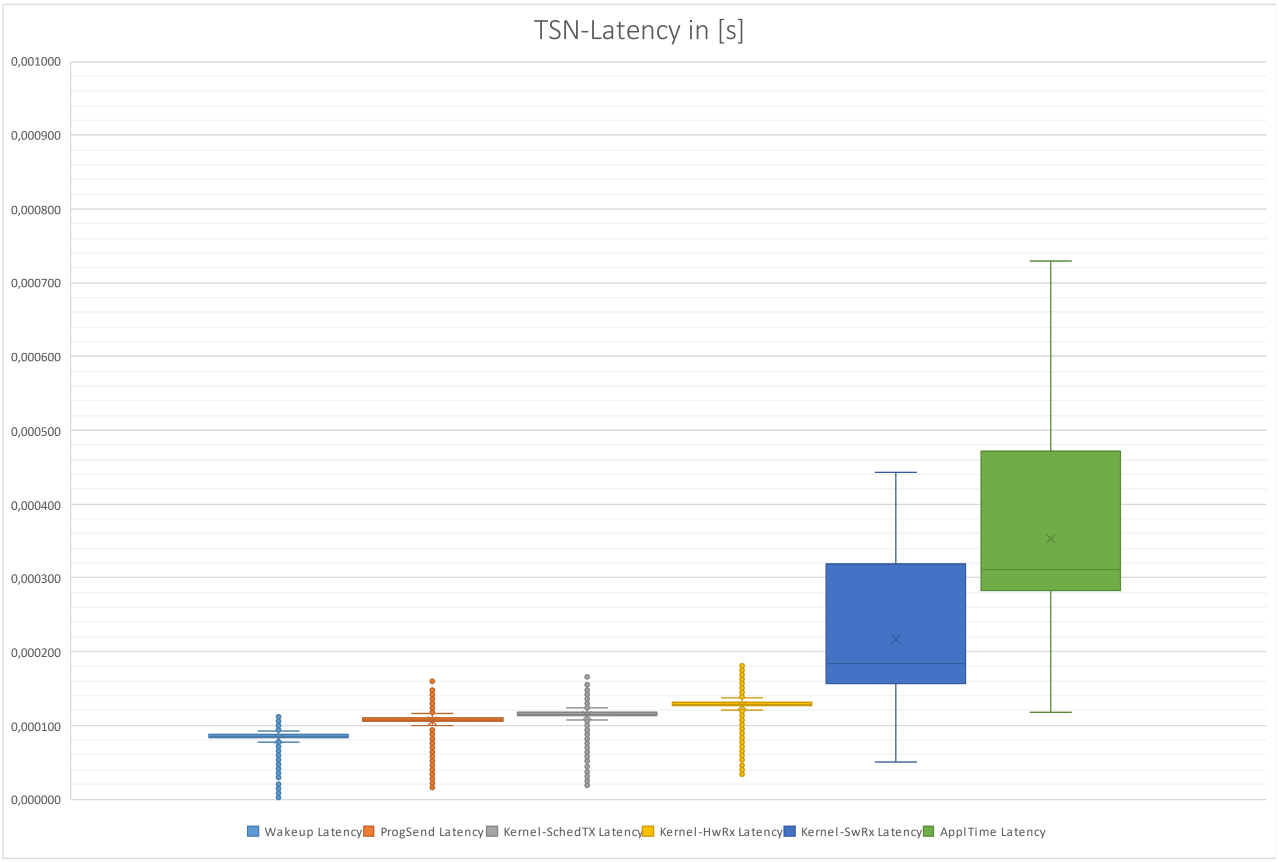


Figure 63: Set-up with TSN priority 6 sending directly from i210-ED without bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Wakeup	82	8
Program Send	105	6
Kernel-SchedTX	113	6
Kernel-HwRx	127	6
Kernel-SwRx	217	69
Application RX	354	88

Table 22: Set-up with TSN priority 6 sending directly from i210-ED without bulk-traffic.

6.2.2.4 Set-up with TSN priority 6 sending directly from i210-ED plus bulk-traffic.

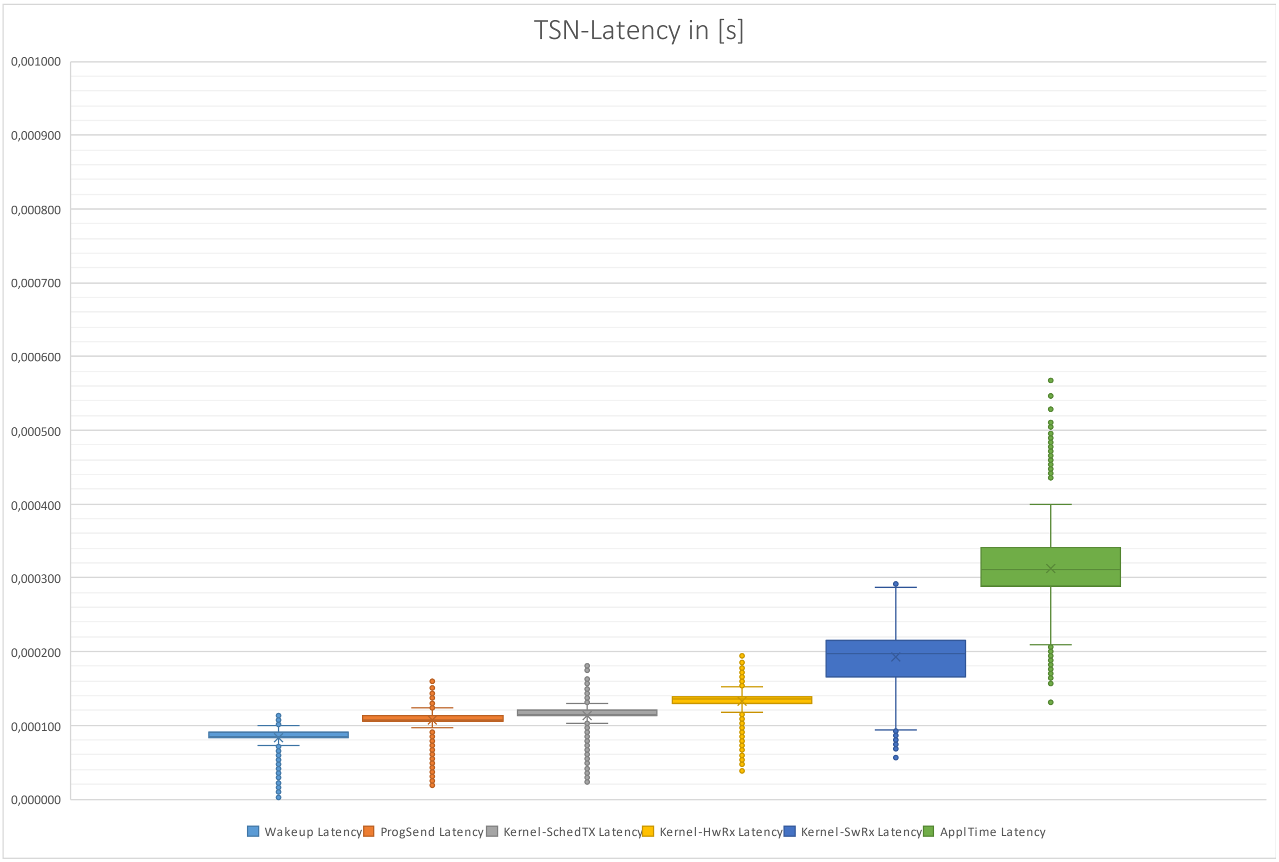


Figure 64: Set-up with TSN priority 6 sending directly from i210-ED plus bulk-traffic.

Timestamp	Mean Latency [µs]	Mean Jitter [µs]
Wakeup	83	7
Program Send	107	6
Kernel-SchedTX	114	6
Kernel-HwRx	133	7
Kernel-SwRx	193	25
Application RX	313	33

Table 23: Set-up with TSN priority 6 sending directly from i210-ED plus bulk-traffic.

6.2.2.5 Conclusion Task 3.2

The results with all tests have many things in common with the TSN-tests, that send the packets with VLAN priority 7. The Wakeup time related to the Interval time is comparable as well as the other latencies of the sender stack as ProgSendTime and the KernelTxSW timestamp. This is expected as the VLAN priority is not intended to have any effect on the sender's software behaviour. When it comes to the KernelHwRx timestamp, the change of the VLAN priority is obvious. The configuration of the TSN slot at the switch is ensuring the deterministic cyclical send and receive timing with a jitter below a microsecond. Packets that do not correspond to the priority of the TSN slot are treated as ordinary IP-packets and produce a larger jitter of 4 to 7µs or more at the receiver.

The average latency of the Application RX timestamp ranges from 313µs to 354µs in these tests as well as the jitter is between 33µs and 88µs. Compared to the PCP7 tests in chapter 5.2 this means a larger instability of the receive stack.

The fastest overall latency was measured in a scenario with bulk traffic (see Table 23). This is unexpected as the bulk traffic should impede the transmission of the packets. However by comparing the time the message spent in the network (difference between Kernel-HWRx and Kernel-SchedTX) the case without bulk traffic (see Table 22, 14 µs) is faster by about 5 µs than with bulk traffic (Table 23, 19 µs), which is in line with what would be expected if the messages are blocked by bulk traffic at the switch. The smaller overall latency for the case with bulk traffic must thus be attributed to the variation in software processing time by the receiving stack, which has a larger impact on the end result in this setup than the interference of the bulk traffic.

The configuration of the gate timing is the same for packets of all priorities. Due to the lack of packets with PCP7, the PCP6 packets are treated equally.

7 Final Conclusion

The setups and tests of task 1, were able to verify and give a deeper insight into the different TSN-mechanisms. These mechanisms are quite specific and add functionalities to the ethernet protocol that cannot be achieved otherwise, such as seamless redundancy on network level, time scheduling of the streams and the preemption of frames. The aim of these mechanisms is to make the network more robust (seamless redundancy), fast and predictable (scheduling, frame preemption). The different mechanisms were investigated in isolation so that the resulting effects could be clearly seen and attributed to the mechanism under consideration.

The main value of the considered TSN protocols is to improve the predictability of the network. By being able to completely separate different streams in the time domain and by providing a way to preempt frames of lower priority while they are being sent, eliminates random delays of messages and makes it possible to define upper bounds on the latency, thus making the network deterministic. However, the random delays eliminated are on the order of the length of a message. For fast networks (> 1 Gbit/s), these delays are rather small and on the order of a few microseconds. Even if compounded over a path with several switches the gains remain small if an overall latency in the millisecond range is aimed at.

Next to the working of the mechanism the relative complexity of the solutions could be noticed. It is not clear how configurations can be generalised and scaled up to a complex network in operation. This is especially valid for the time-scheduling (IEEE 802.1Qbv) and the redundancy protocols (IEEE 802.1CB) as these protocols, need a network wide coordination of the devices. IEEE 802.1Qbu can be configured on a per link basis and thus involves less oversight for it to be implemented.

Moreover, as was seen when integrating TSN into a network stack like TRDP, the latency of the network is small compared to the latencies of the software stacks on the sending and receiving side, even though the stack was executed on a Linux system with a real time patch. Thus, the full potential of the TSN protocols is limited by these factors outside of the networks scope, and the advantage of TSN over usual QoS techniques is lessened. Indeed, tests using usual TRDP Process Data and QoS techniques showed a similar overall performance in the test setup under consideration.

Another finding was that the TRDP protocol stack, even though it starts to support TSN messages, needs a lot of application dependent configuration and adjustments to fully work. A stack with TSN support ready out of the box does not seem to exist yet.

For current requirements on latencies (10-100 ms), the additional complexity and effort that comes with a TSN network does not seem reasonable for the relatively small gains compared to a well set up priority system using classical QoS techniques. This applies especially to the scheduling (IEEE 802.1Qbv) and seamless redundancy (IEEE 802.1CB). From the tests it can be concluded that standard ethernet with classical QoS techniques provides sufficient performance for the needs of the CCN. Additional frame preemption (IEEE 802.1Qbu) to reduce the mean time a priority message takes to transfer through a switch could however be considered as an option.