OCORA

Open CCS On-board Reference Architecture

CCS Communication Network

Proof of Concept (PoC)

Document ID: OCORA-TWS02-020

Version: 2.00

Release: R1

Date: 03.12.2021

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 is based on the CCN evaluation report [8]. The CCN evaluation report proposes the CCN to be a TSN Ethernet based network with the use of SDTv2 / SDTv4 as safety layers. In order to be able to integrate the CCN on the next generation of train communication network (NG-TCN) every hard-real-time dependent CCS on-board device (e.g. CCU, BTM, etc.) should have at least one TSN-capable Ethernet port, whereas for soft- or non-real-time dependent CCS on-board devices a single standard non-TSN-capable Ethernet port is sufficient. Hard-real-time dependent devices can use both planes of the NG-TCN with two TSN-capable Ethernet ports in order to improve reliability and availability.

On session layer TRDP 2.0, OPC-UA Pub/Sub (over TSN) or DDS/RTPS (over TSN) are suitable solutions.

The proposed protocol stack of CCN is listed in the following table. Highly recommended standards to be used as reference for procurement in OCORA are listed in **bold** font.

Layer	Protocol for hard-real-time data		Protocol for soft- or non-real-time data
(Safety Layer[1])	**(SDTv2 / SDTv4)**		
Session Layer	TRDP 2.0, OPC-UA Pub/Sub or DDS/RTPS		
Transport Layer		UDP TCP	UDP TCP
Network Layer		IPv4	IPv4
Data Link Layer	**Time-Sensitive Networking (TSN) IEEE 802.1**		**Standard Ethernet IEEE 802.3**
Physical Layer	**100BASE-TX or 1000BASE-T**		

Table 1: Protocol Stack CCN

The defined protocol stack allows safety-related and hard-real-time dependent data traffic. For non-safety-related and soft- or non-real-time dependent data, standard TCP/IP or UDP/IP data traffic over standard Ethernet (IEEE 802.3) can be used on the same physical layer of the CCN.

This document is reporting the task definitions and results of the proof-of-concept (PoC). During Release R1 phase, the test equipment was procured, and first tests were executed. With the PoC, the feasibility of the CCN as a TSN-Ethernet network was shown. Furthermore, TSN was shown being a good solution for hard-real time data transfer. Certain amount of bandwidth can be guaranteed with very low network latency and jitter even in congested network situations. The planned end-to-end TSN tests with an implemented session layer protocol (TRDP 2.x) could not be finished yet. These will be finalized in the subsequent phase of OCORA.

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

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

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

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 1.0, 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 a time-sensitive 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 bus 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 technical view of the OCORA architecture [7] is shown. The CCN connects all components of the future CCS on-board system. The most important CCS on-board components are:

- CCS computing units (CCUs)
- Driver Machine Interfaces (DMIs)
- Vehicle Locator (VL)
- Balise Transmission Module (BTM)
- Loop Transmission Module (LTM)
- National Train Control System (NTC) or Specific Transmission Module (STM)
- Cab Voice Device (CVD)
- Train Recording Unit (TRU)
- Input / Output Ports (I/O Ports)
- Gateway to Train Control Management System Network, Operator Network or Communication Network (ECN/ECN Security Gateway)

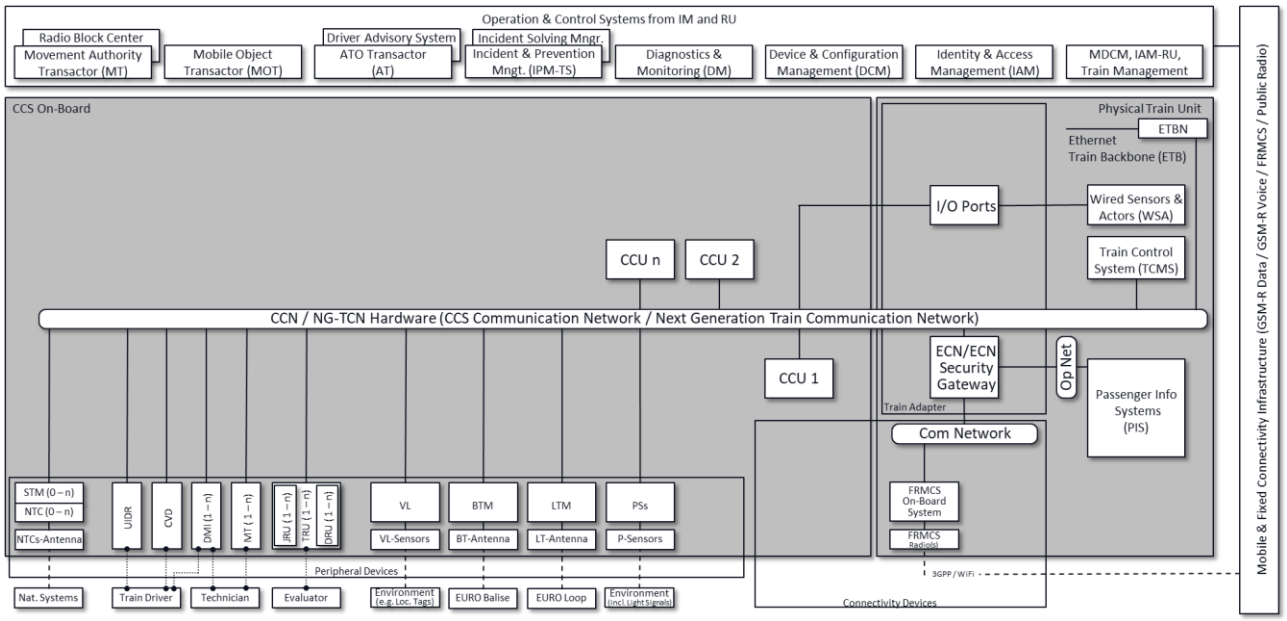


Figure 1: Technical architecture from [7]

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 all 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. The CCN itself will be modifiable in accordance with future technological evolutions by means of strict separation of the different communication layers (OSI Layers).

The CCN evaluation of the lower communication layers proposes the CCN to be a TSN Ethernet based network with the use of SDTv2 / SDTv4 as safety layer. In order to be able to integrate the CCN on the next generation of train communication network (NG-TCN) every hard-real-time dependent CCS on-board device (e.g. CCU, BTM etc.) should have at least one TSN-capable Ethernet port whereas for soft- or non-real-time dependent CCS on-board devices a single standard non-TSN-capable Ethernet port is sufficient. Hard-real-time dependent devices can use both planes of the NG-TCN with two TSN-capable Ethernet ports in order to

improve reliability and availability.

On session layer TRDP 2.0, OPC-UA Pub/Sub (over TSN) or DDS/RTPS (over TSN) are suitable solutions. These three options will be further investigated considering the system architecture with platform/CCU and the subcomponents.

The proposed protocol stack of CCN is listed in the following table. Highly recommended standards to be used as reference for procurement in OCORA are listed in **bold** font.

Layer	Protocol for hard-real-time data		Protocol for soft- or non-real-time data
(Safety Layer[2])	**(SDTv2 / SDTv4)**		
Session Layer	TRDP 2.0, OPC-UA Pub/Sub or DDS/RTPS		
Transport Layer		UDP TCP	UDP TCP
Network Layer		IPv4	IPv4
Data Link Layer	**Time-Sensitive Networking (TSN) IEEE 802.1**		**Standard Ethernet IEEE 802.3**
Physical Layer	**100BASE-TX or 1000BASE-T**		

Table 2: Protocol Stack CCN

The defined protocol stack allows safety-related and hard-real-time dependent data traffic. For non-safety-related and soft- or non-real-time dependent data, standard TCP/IP or UDP/IP data traffic over standard Ethernet (IEEE 802.3) can be used on the same physical layer of the CCN.

1.7 Goal

Remaining tasks from Delta Release phase were handled in Release R1 phase. These are the investigation of network architecture with detailed technical implementation of CCN in NG-TCN (network configuration) and cyber-security as well as to align new standards and regulations.

Further, a demonstrator shall show the feasibility of the CCN with logically separated networks on a common physical train communication network (NG-TCN). The setup of the demonstrator helps to investigate the technical implementation details of the CCN in NG-TCN.

The following tasks have been performed during Release R1 phase:

- **Evaluation CCN**
  - **Network Architecture**: Network architecture regarding connections from train to trackside (mobile communication gateway) must be further investigated together with TCMS domain / CONNECTA. Also, DMI concept with different displays from different domains on the driver desk must be clarified in order to define network architecture with its zones and conduits.
  - **Detailed technical Implementation of CCN in NG-TCN**: The detailed technical implementation of CCN in NG-TCN shall be elaborated together with CONNECTA. The composition of the demonstrator helps to investigate the technical implementation details.
  - **Cyber-security**: The impact of the security concept of CONNECTA (NG-TCN) on the CCN shall be investigated. As a result of the investigation, the security requirements on the CCN (layers 1 to 6) shall be defined.
  - **New Standards and Regulations (e.g. TSI 2022)**: The work done in different working groups (e.g. IEC TC9 WG43 or ERA TWG Archi) shall be aligned in order to get consistent new standards and regulations (e.g. IEC 61375, TSI 2022, ERA Subsets, OCORA specifications).
  - **Evaluation of Data Serialization Formats**: Data serialization formats of application data shall be evaluated. Possible solutions are: bitstream like in today's subset specifications, XML, JSON, CBOR, Apache Thrift, Protobuf.

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

- **Proof of Concept / Demonstrator CCN**: A demonstrator shall show the feasibility of the CCN with logically separated networks on a common physical train communication network (NG-TCN). Also, the impact of TSN-Ethernet versus non-TSN-Ethernet in congested network situation shall be demonstrated. The setup of the demonstrator helps to investigate the technical implementation details of the CCN in NG-TCN.

This document contains the current results for the Proof of Concept / Demonstrator. The evaluation part is documented in [8].

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.

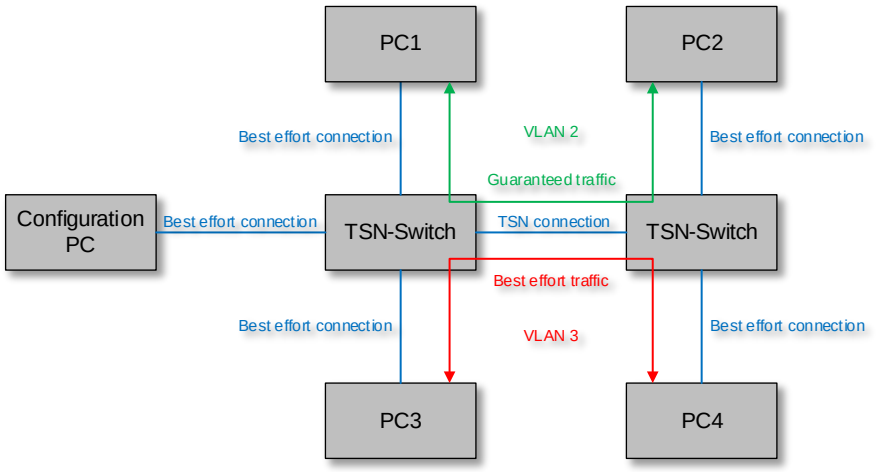


Figure 2: Network topology task 1

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 2.

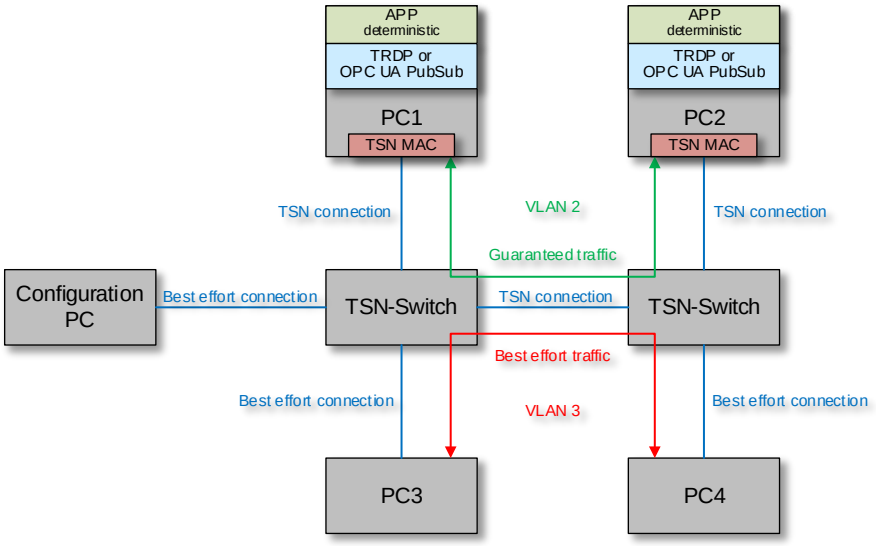


Figure 3: Network topology task 2 a)

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.

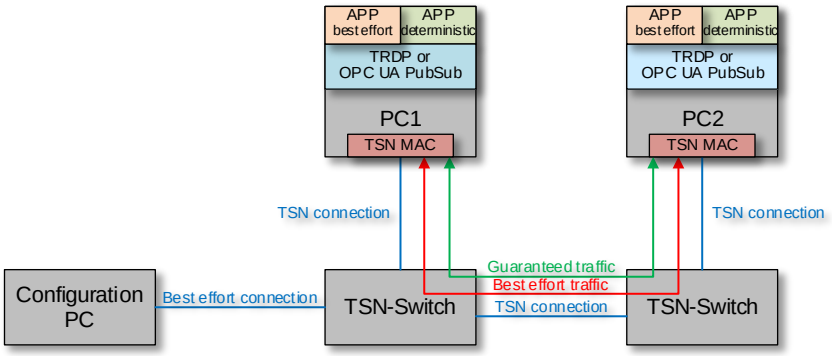


Figure 4: Network topology task 2 b)

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.

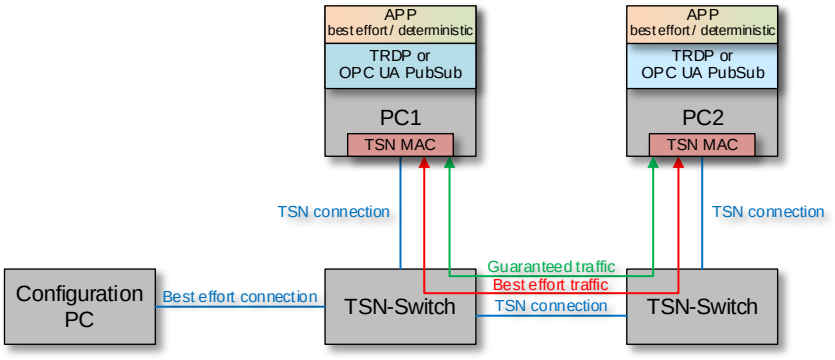


Figure 5: Network topology task 2 c)

3 Task 1: Test results

3.1 Best effort traffic generated locally

Task 1 was realised using the 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 ([9]) 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.

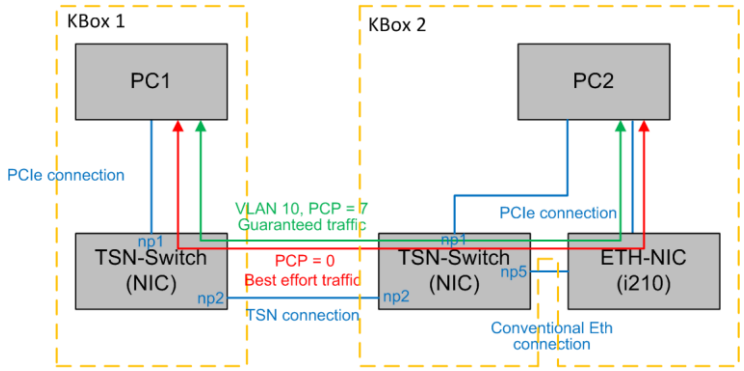


Figure 7: Connections of the computers, basic setup.

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.

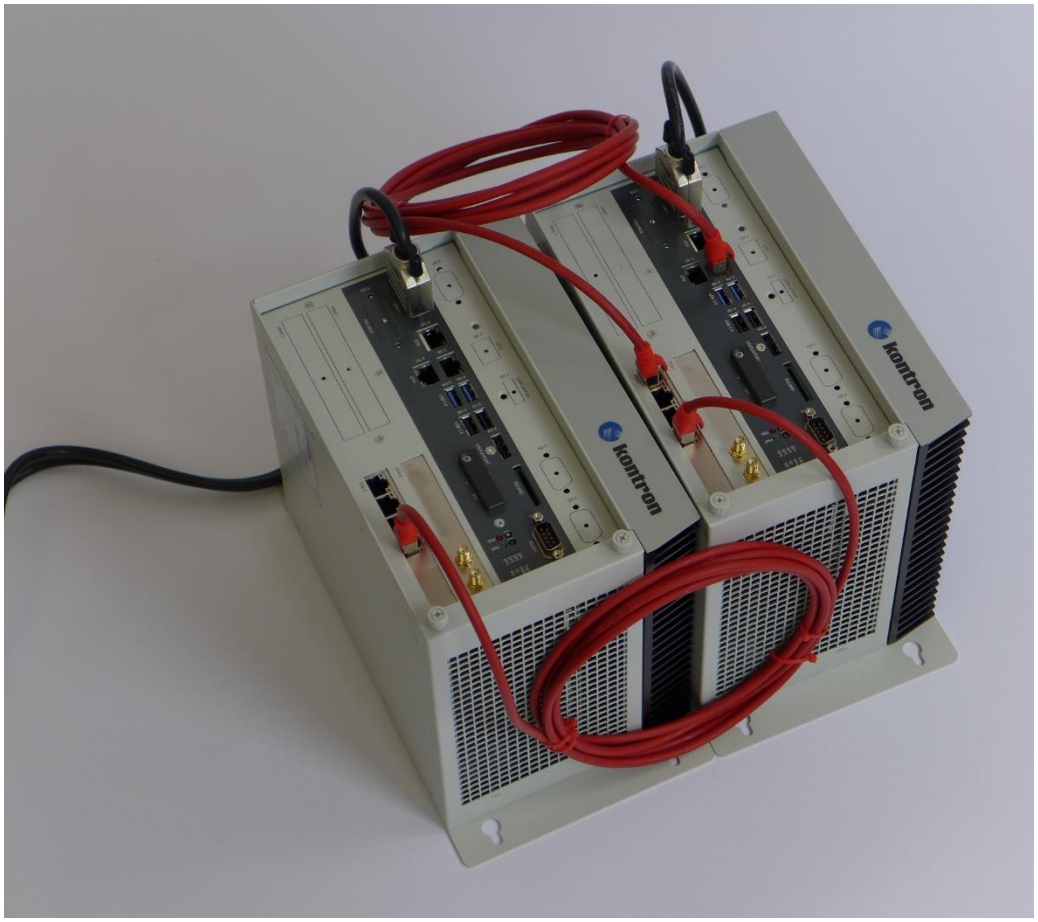


Figure 8: Photo of the test setup shown in Figure 7. (See also Figure 6 for details on the interfaces)

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.

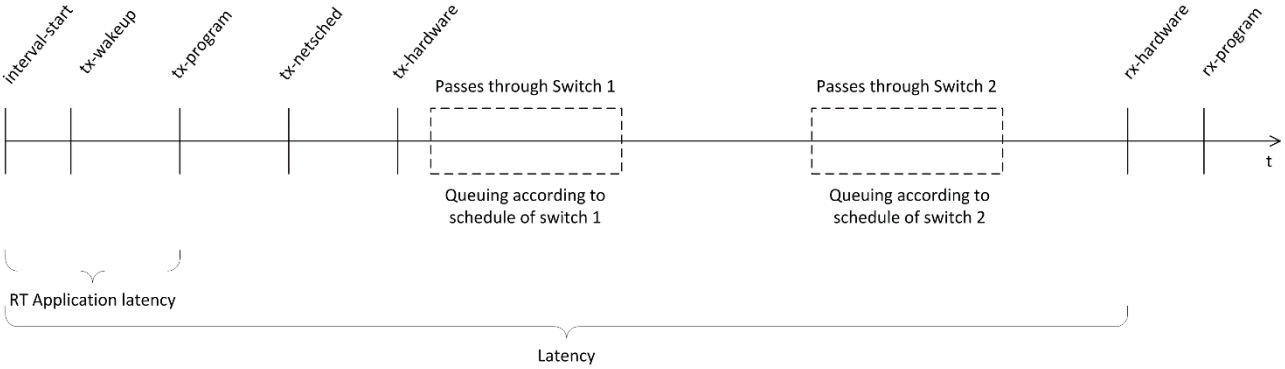


Figure 9: Message path, with time-stamp events. Time differences are not to scale.

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:

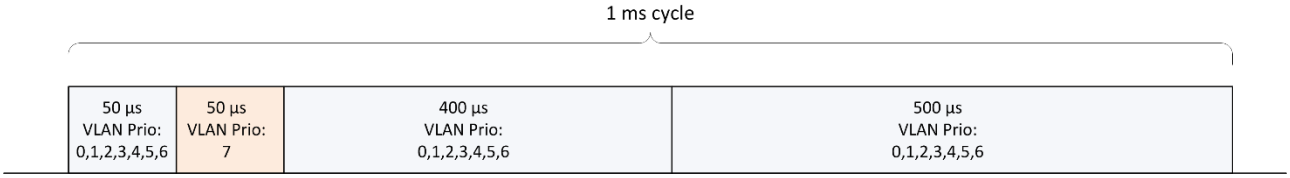


Figure 10: Schedule example used to execute the test cases. The last 900 µs of the schedule are configured in two slices as there is a maximum slice length that needs to be respected for the definition of the schedule configuration.

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.

Figure 11 gives the results for a connection with only scheduled traffic. Unsurprisingly the network performs well with low jitter of less than 1 µs. The latency is at 52 µs to 53 µs. The results were produced with 10'000 messages sent every 1 ms.

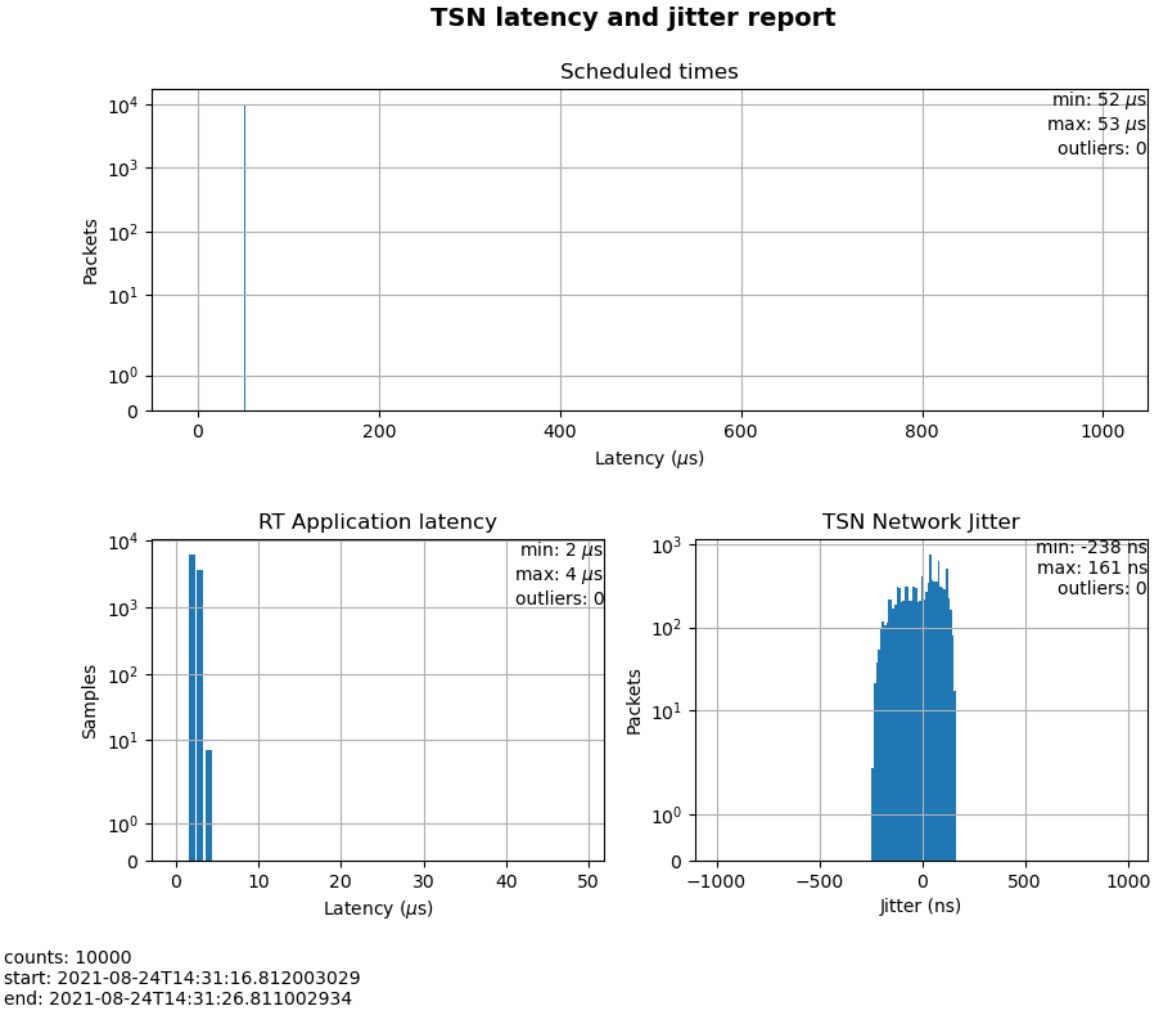


Figure 11: Histograms of latency, application latency and network jitter for scheduled traffic alone.

Figure 12 gives 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 latency for scheduled messages of 52 µs to 53 µs and similarly small jitter.

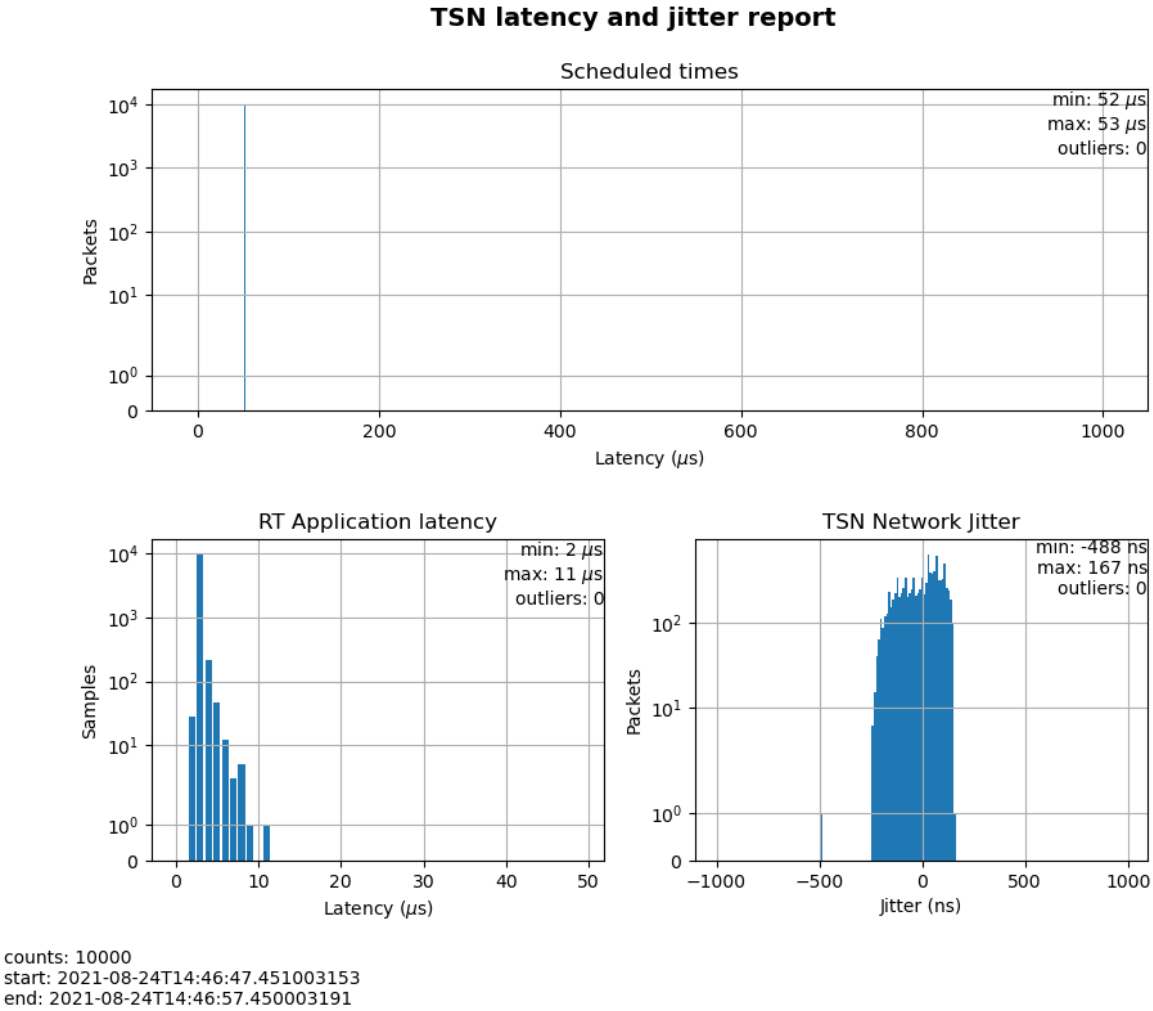


Figure 12: Histograms of latency, application latency and network jitter for scheduled traffic alongside high-volume best effort traffic.

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 13. We see that the latency is reduced compared to the scheduled case (see Figure 11). 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 14. 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 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.

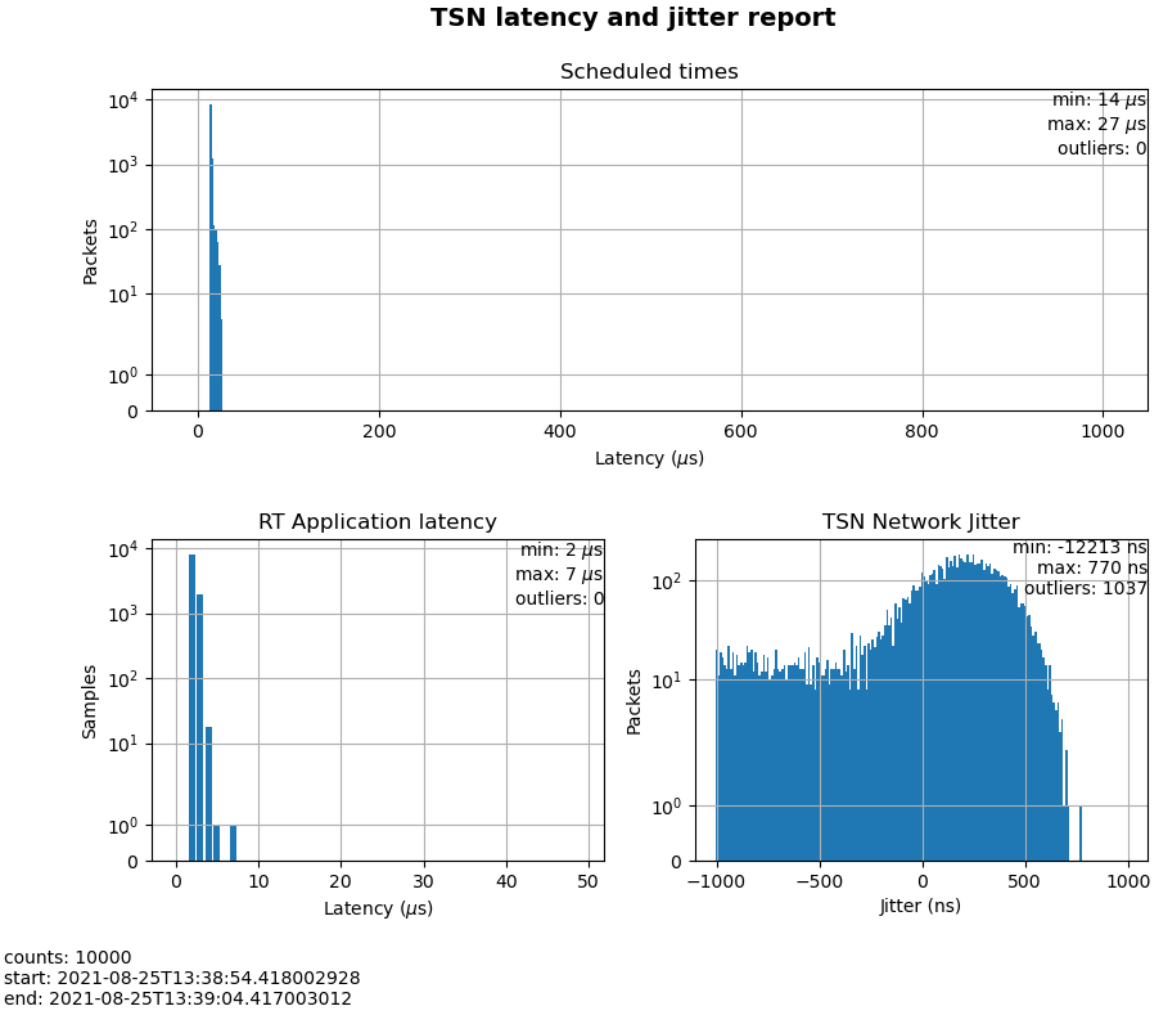


Figure 13: Histograms of latency, application latency and network jitter for scheduled traffic with VLAN priority 2 without best effort traffic.

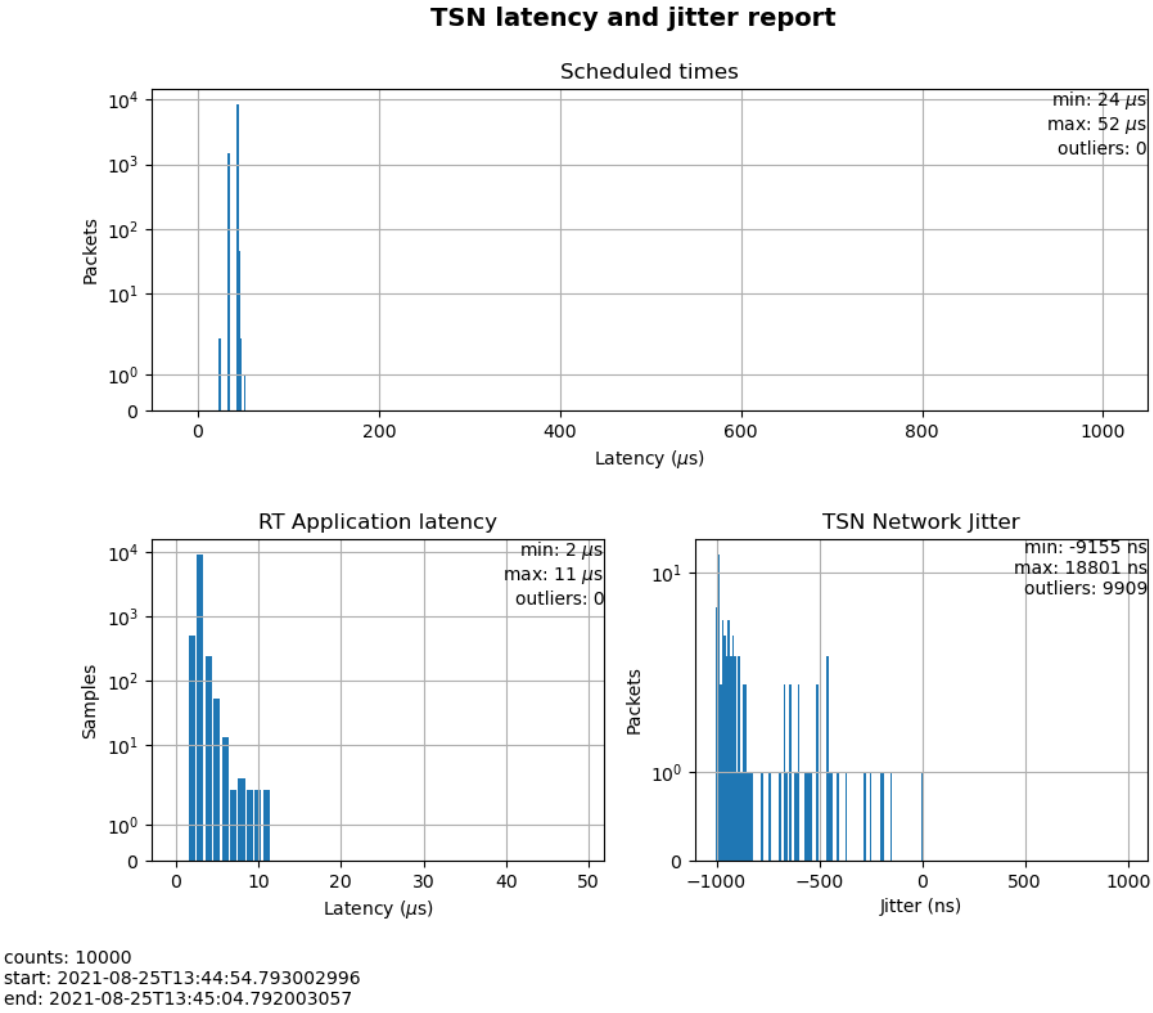


Figure 14: Histograms of latency, application latency and network jitter for scheduled traffic with VLAN priority 2 alongside high-volume best effort traffic.

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 15. This schedule has the advantage of reducing the 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.

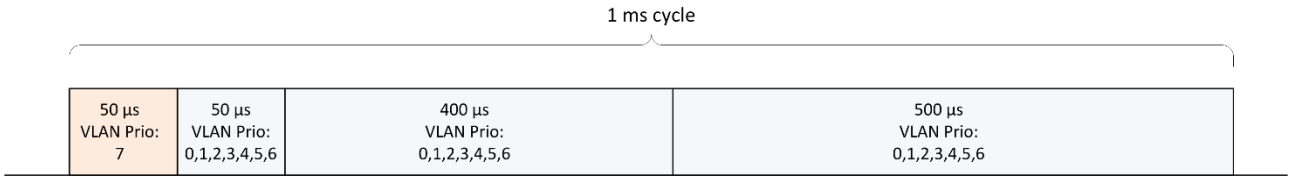


Figure 15: Schedule example 2 to send messages with a latency as small as possible.

Figure 16 shows the result without best effort traffic and Figure 17 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 13). With the best effort traffic however, we can observe a different reaction to the one seen previously (Figure 14). As the traffic classes should remain separate 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 16 and Figure 17 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 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.

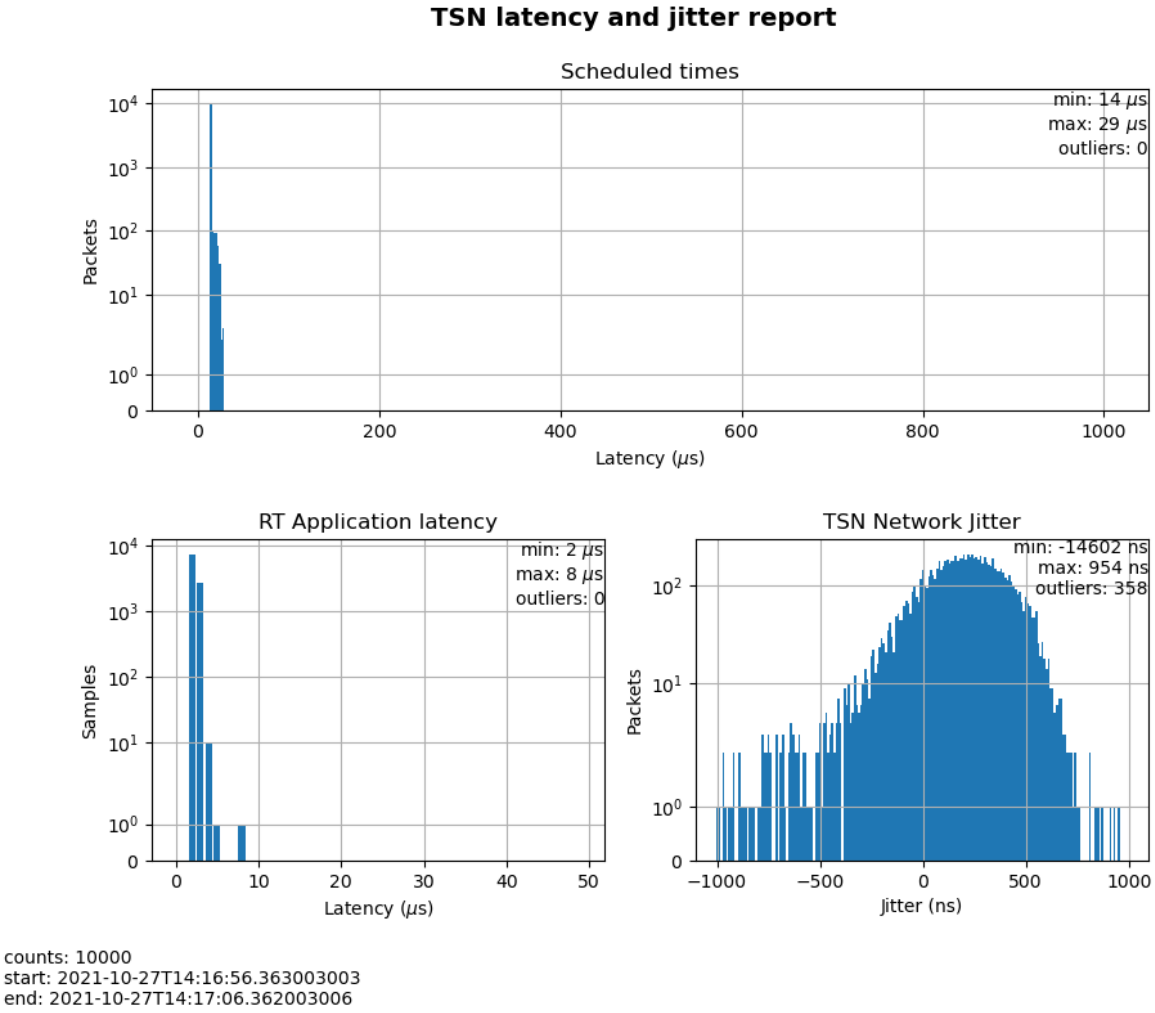


Figure 16: Histograms of latency, application latency and network jitter for traffic with VLAN priority 7 without best effort traffic. Schedule 2 from Figure 15 was used.

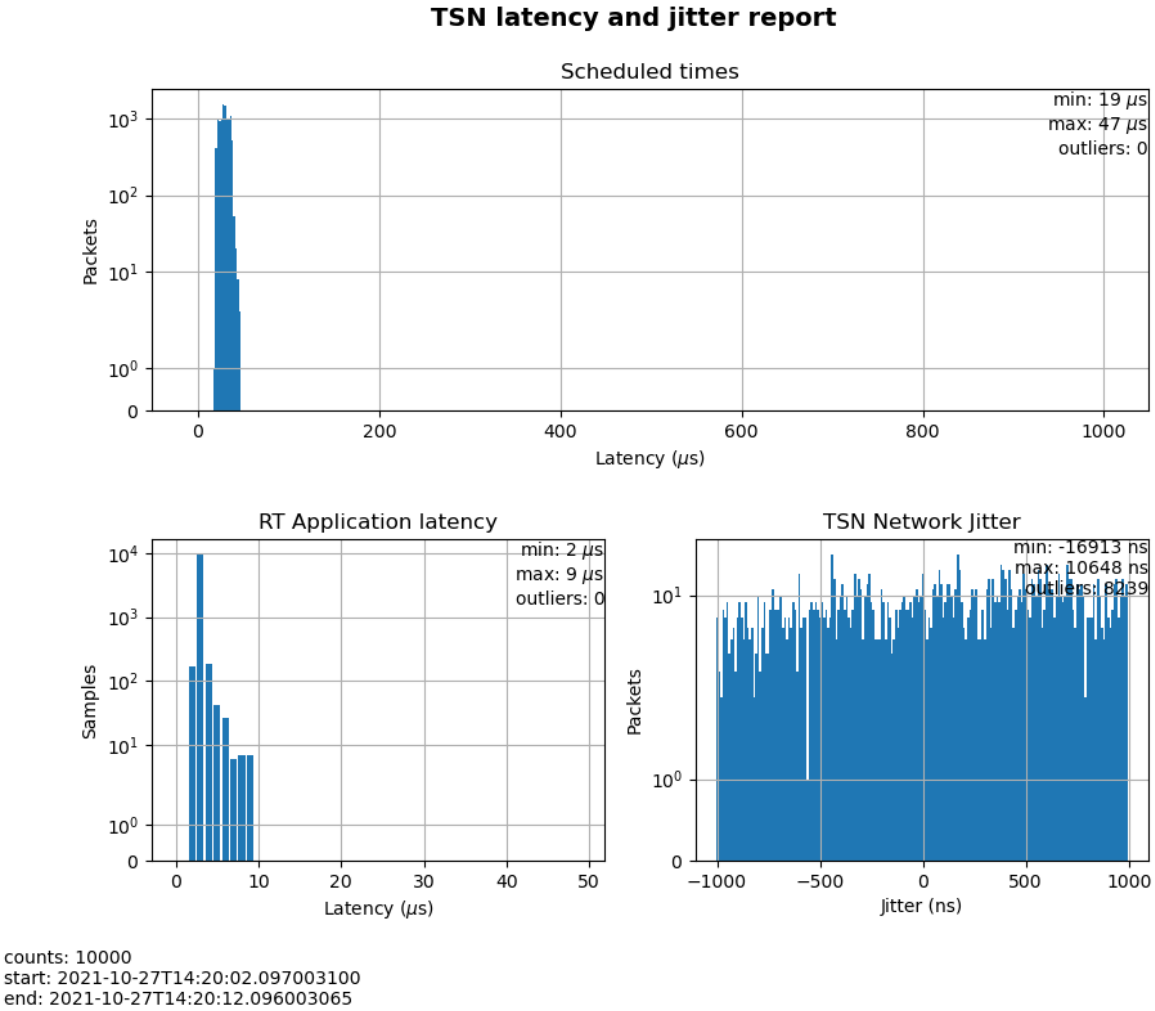


Figure 17: Histograms of latency, application latency and network jitter for traffic with VLAN priority 7 alongside high-volume of best effort traffic. Schedule 2 from Figure 15 was used.

3.2 Best effort traffic generated externally

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. Thus, 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 bottle neck of the network under test. The setup is shown in Figure 18 schematically, a photo can be seen in Figure 19.

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 15).

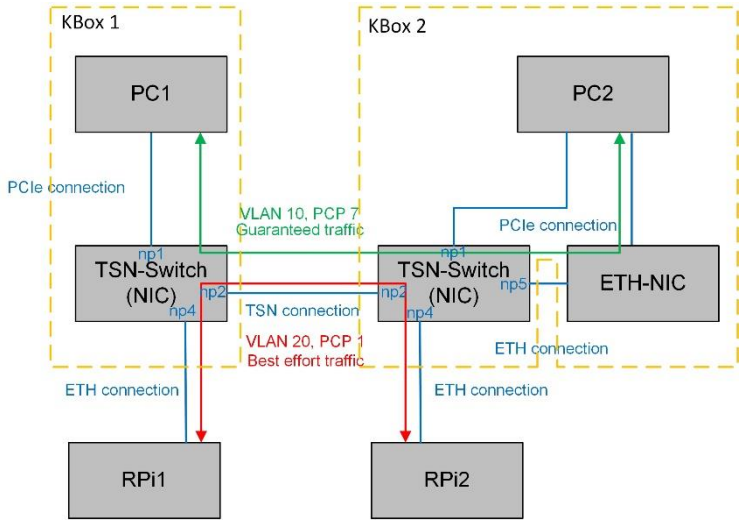


Figure 18: Connections of the computers, setup with dedicated load generation computers.

The scheduled traffic is routed as 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.

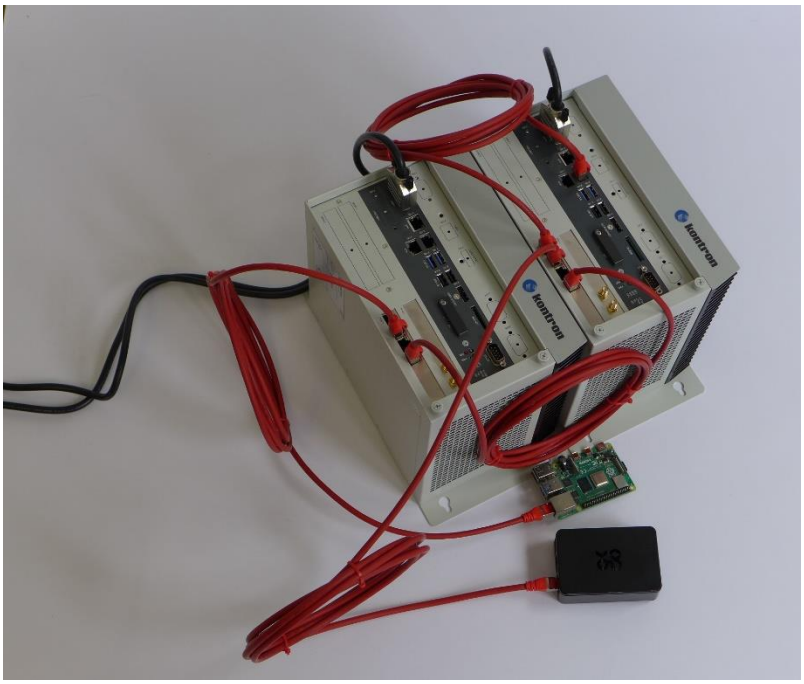


Figure 19: Photo of the test setup shown in Figure 18. (See also Figure 6 for details on the interfaces)

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 20. As expected, the results are comparable to the ones from Figure 11, 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 21. 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.

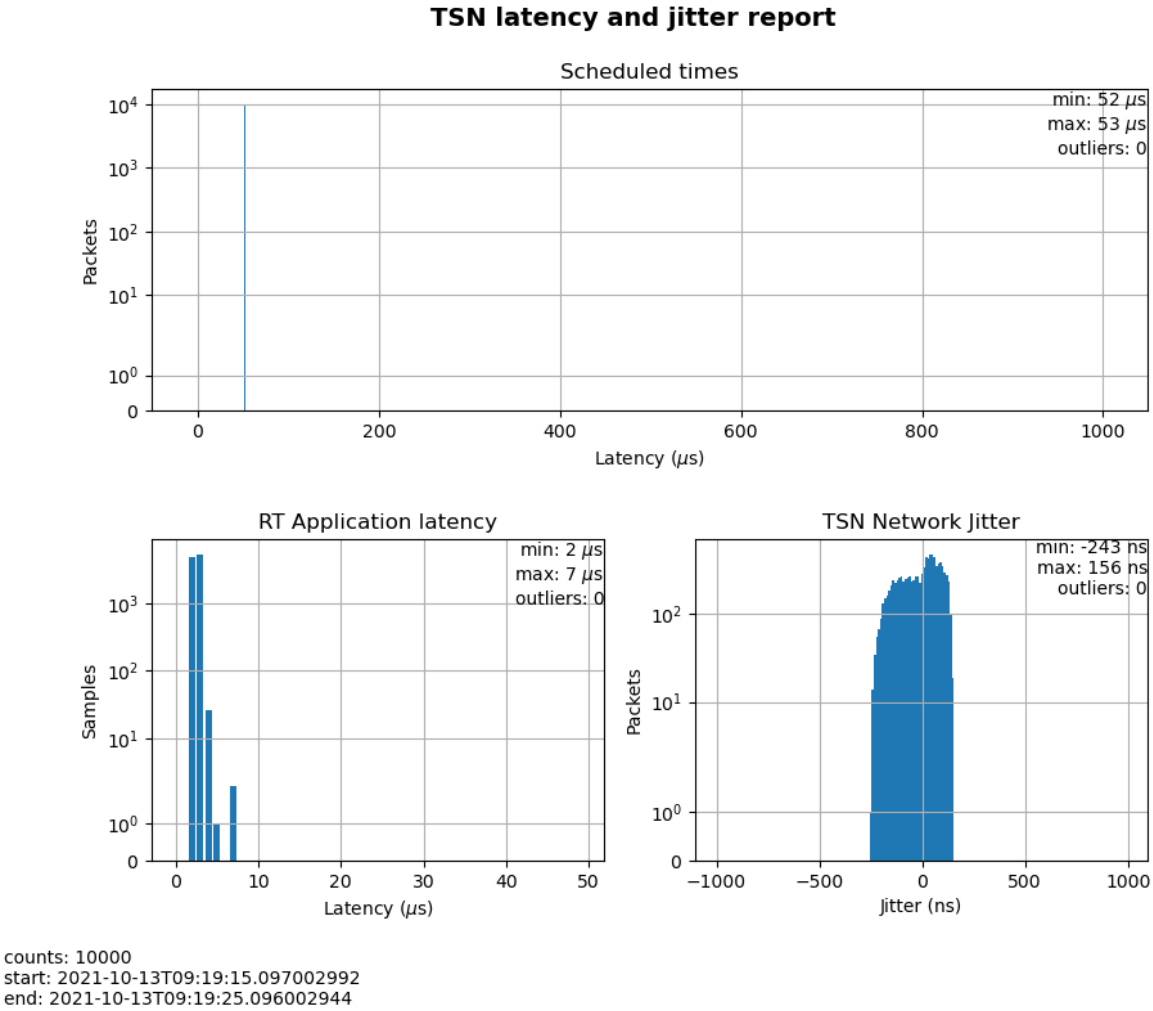


Figure 20: Histograms of latency, application latency and network jitter for scheduled traffic alone with the RPi computers connected but not generating traffic. This is essentially the same case as in Figure 11 and gives comparable results.

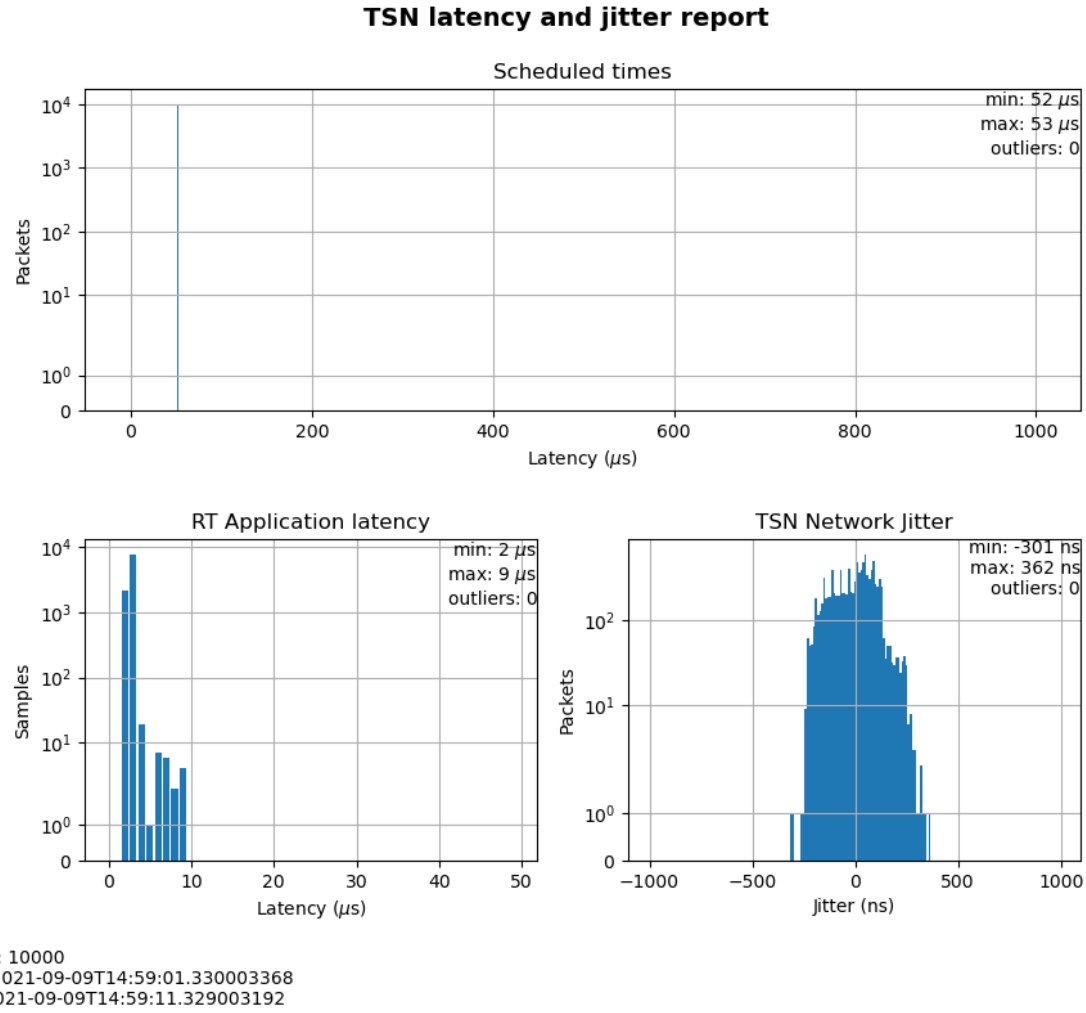


Figure 21: Histograms of latency, application latency and network jitter for scheduled traffic with the RPi computers connected and generating high-volume best effort traffic.

Using schedule 2 (see Figure 15) 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 22 (without best effort traffic) and Figure 23 (with best effort traffic alongside). As expected, the result in the case without best effort traffic is very similar to the previous case (Figure 16). 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 17). 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.

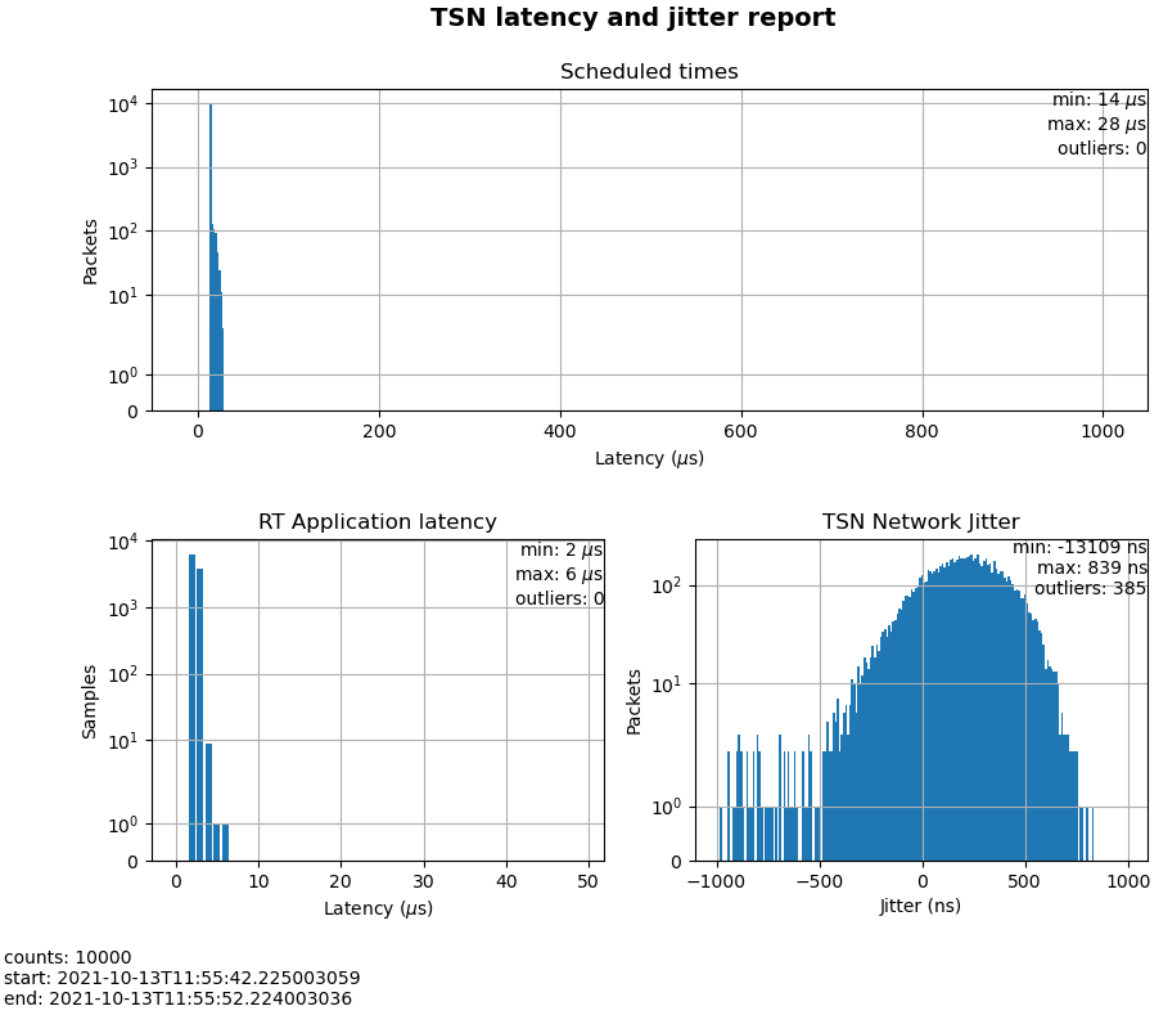


Figure 22: Histograms of latency, application latency and network jitter for traffic with VLAN priority 7 without best effort traffic. Schedule 2 from Figure 15 was used.

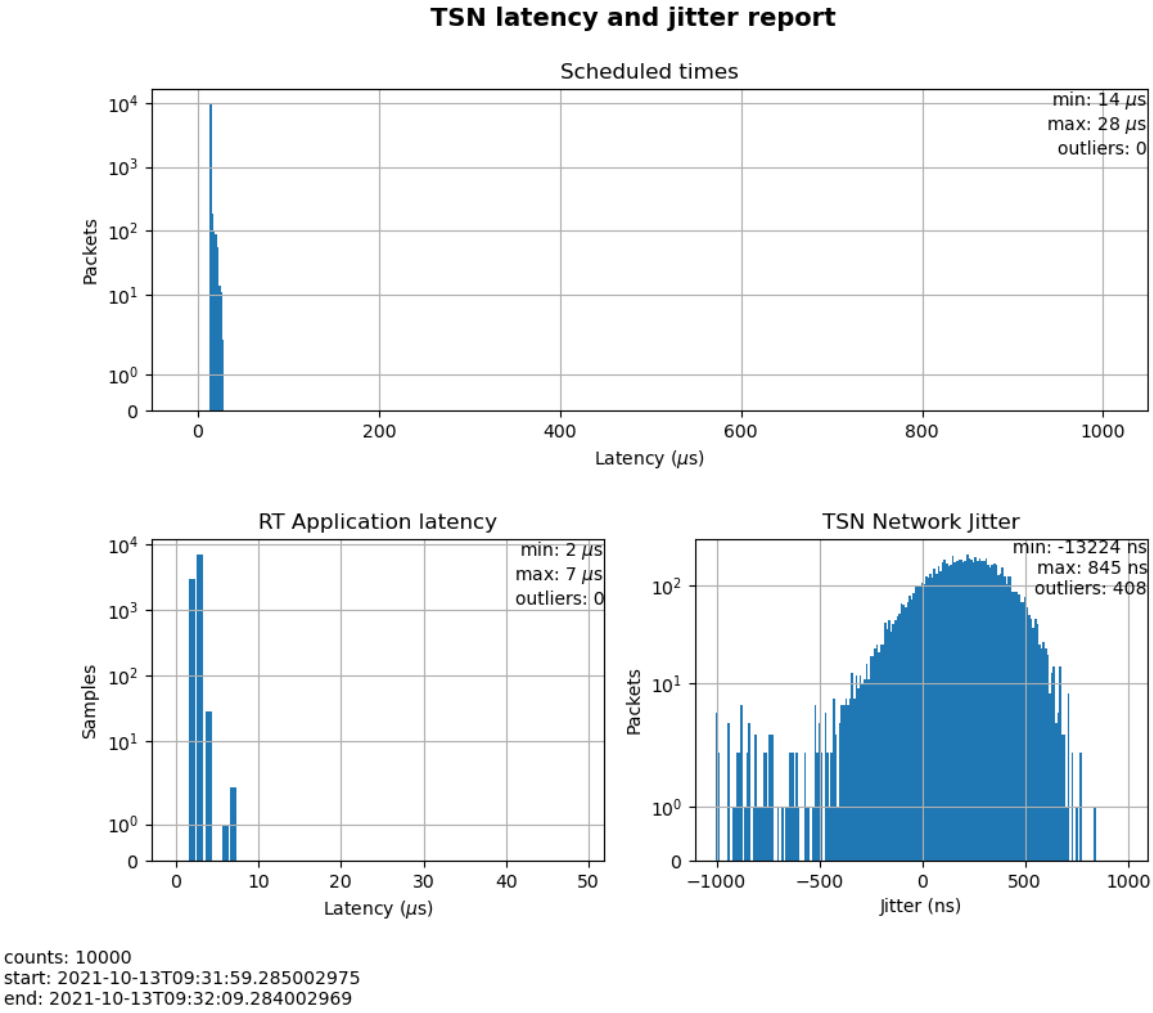


Figure 23: Histograms of latency, application latency and network jitter for traffic with VLAN priority 7 with high-volume best effort traffic between the two RPis. Schedule 2 from Figure 15 was used.

3.3 Remark Task 1

Here we discuss a situation that happened several times in the test case with using schedule 2 (Figure 15) and with bulk traffic (as in Figure 17). 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[4], is shown in Figure 24. 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 taken into account 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 messages and that the time buffer is sufficient so that one can expect all the messages to be sent within the cycle. Moreover, it is also necessary to take into account 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 times involved are 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 like Slate XNS from TTTech, which will be tested at a later point in time, are programmed to take these shifts into account,

[4] The datapoint corresponding to this message is plotted in the latency graph of Figure 24 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.

thus simplifying the scheduling process.

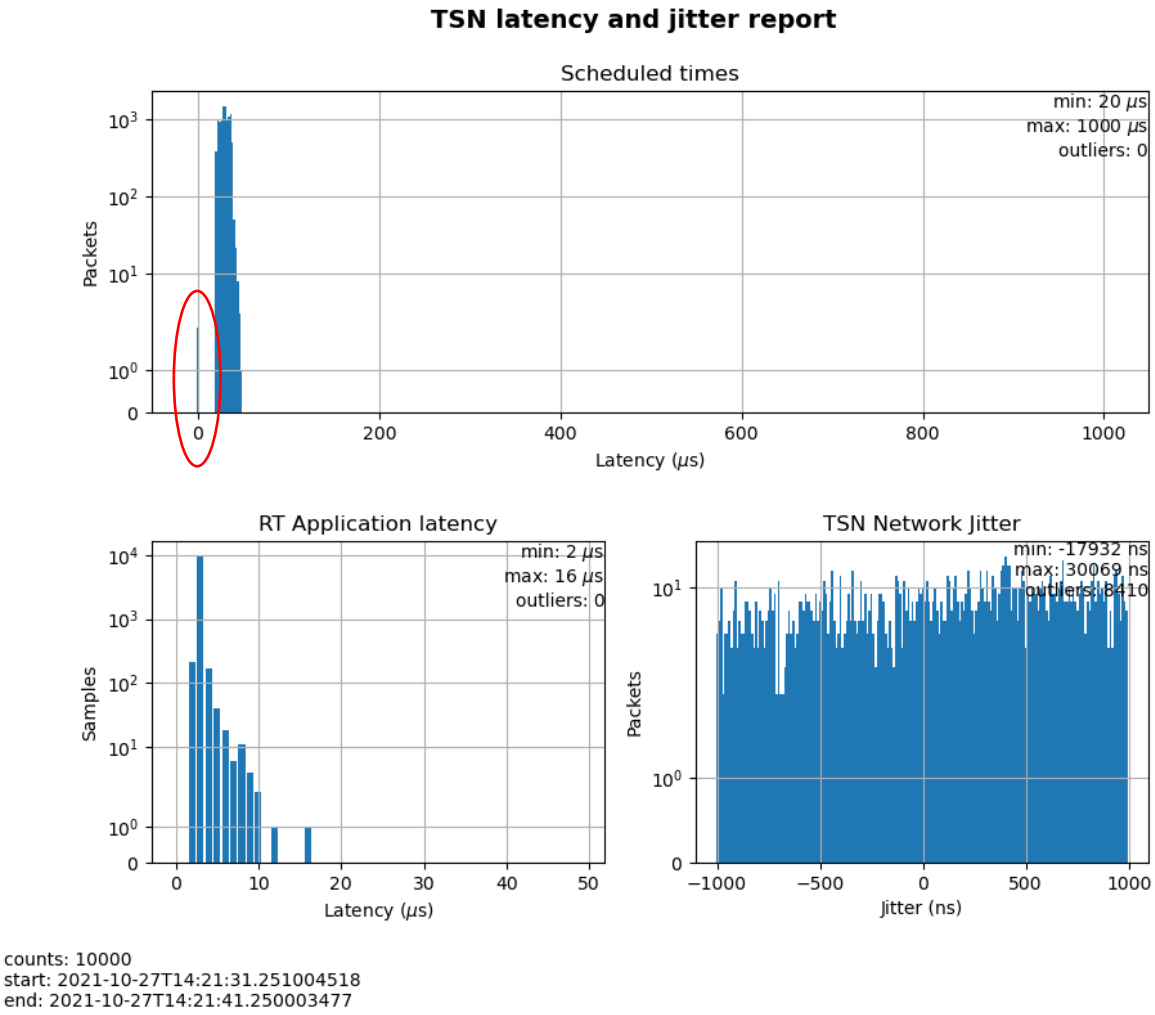


Figure 24: Histograms of latency, application latency and network jitter for traffic with VLAN priority 7 with best effort traffic alongside. Schedule 2 from Figure 15 was used. Two messages did not manage to make it through the network on time with the schedule and were thus withheld until the next cycle (see red mark in latency plot).

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

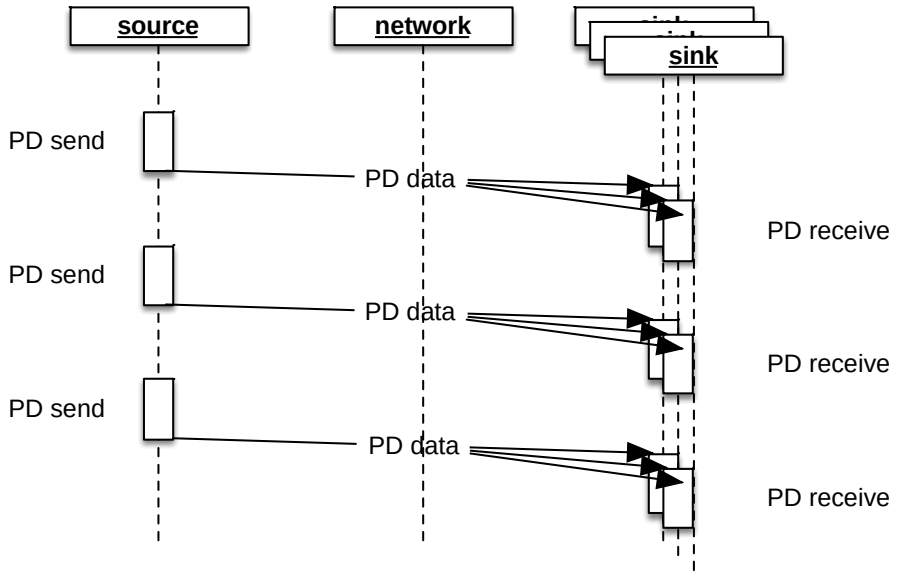
4.1 TCNOpen TRDP stack with TSN – inside

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 25: 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[5]: vlan, TSN**)
- management parameters (session/internal handle, callback, topography counters)

[5] 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[5]: 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 26 shows the hierarchy and layers of a TRDP application. The naming of the TRDP Light API[6] 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.

[6] 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.

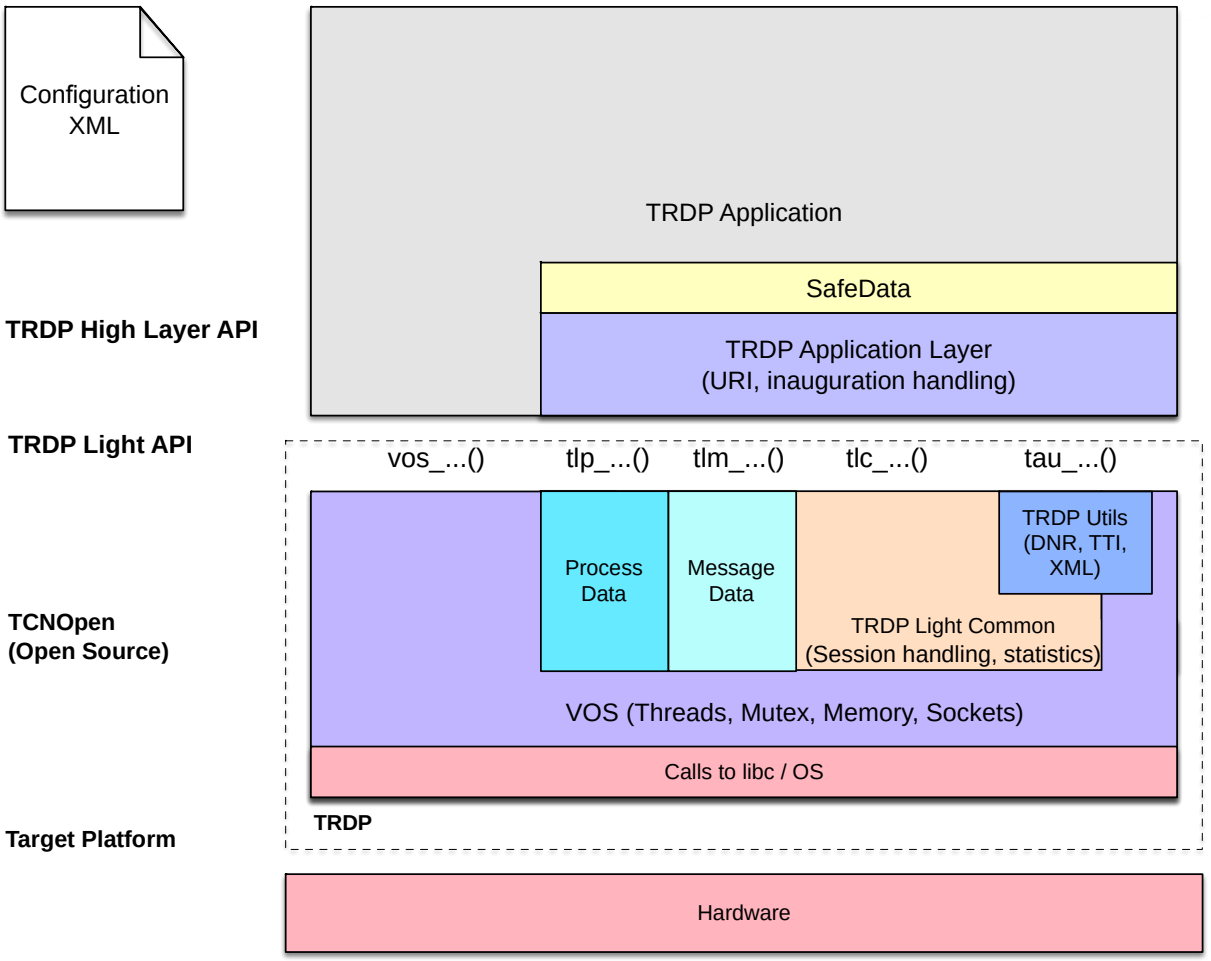


Figure 26: Hierarchy of the TCNOpen TRDP stack

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.

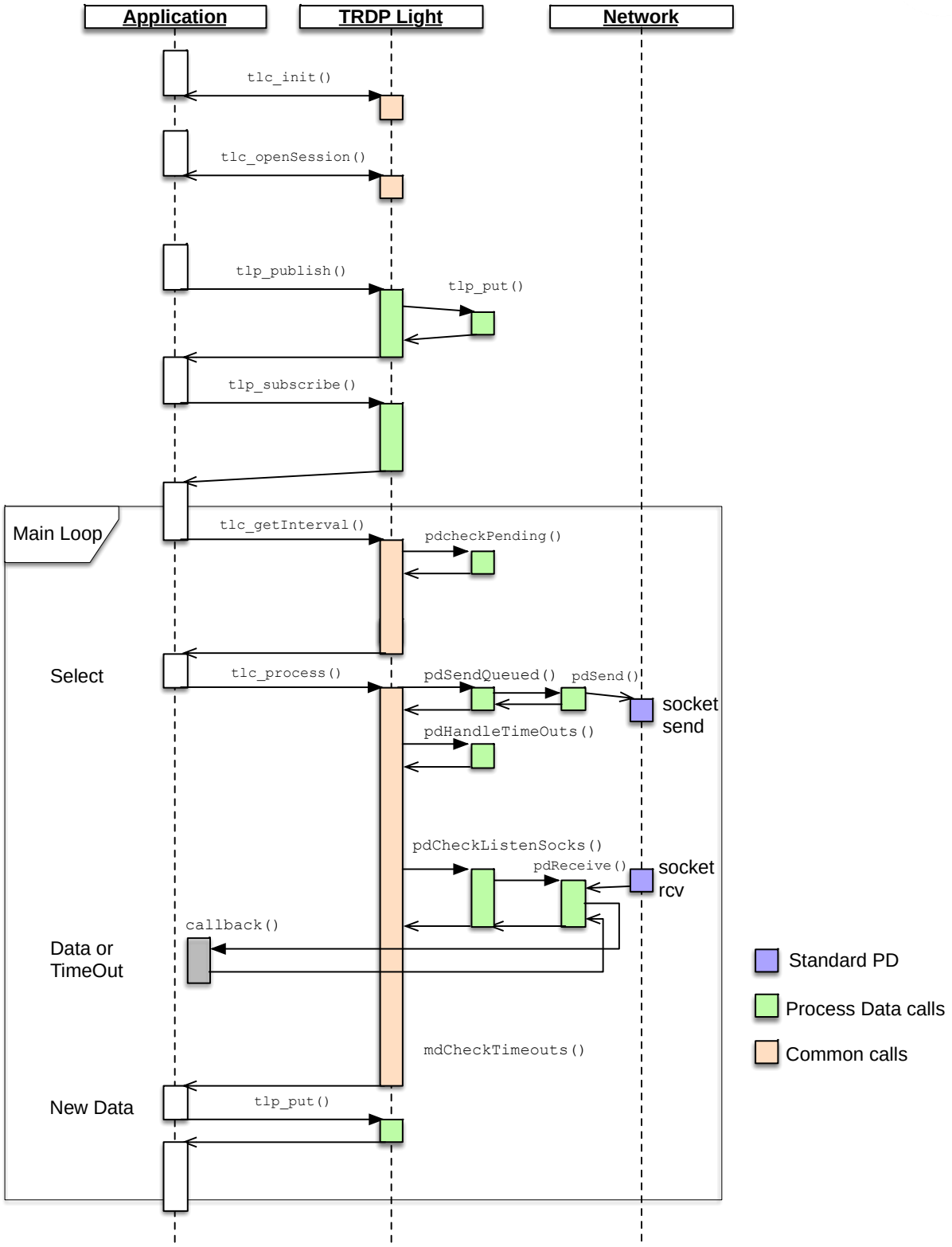


Figure 27: Legacy PD handling

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 27 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)

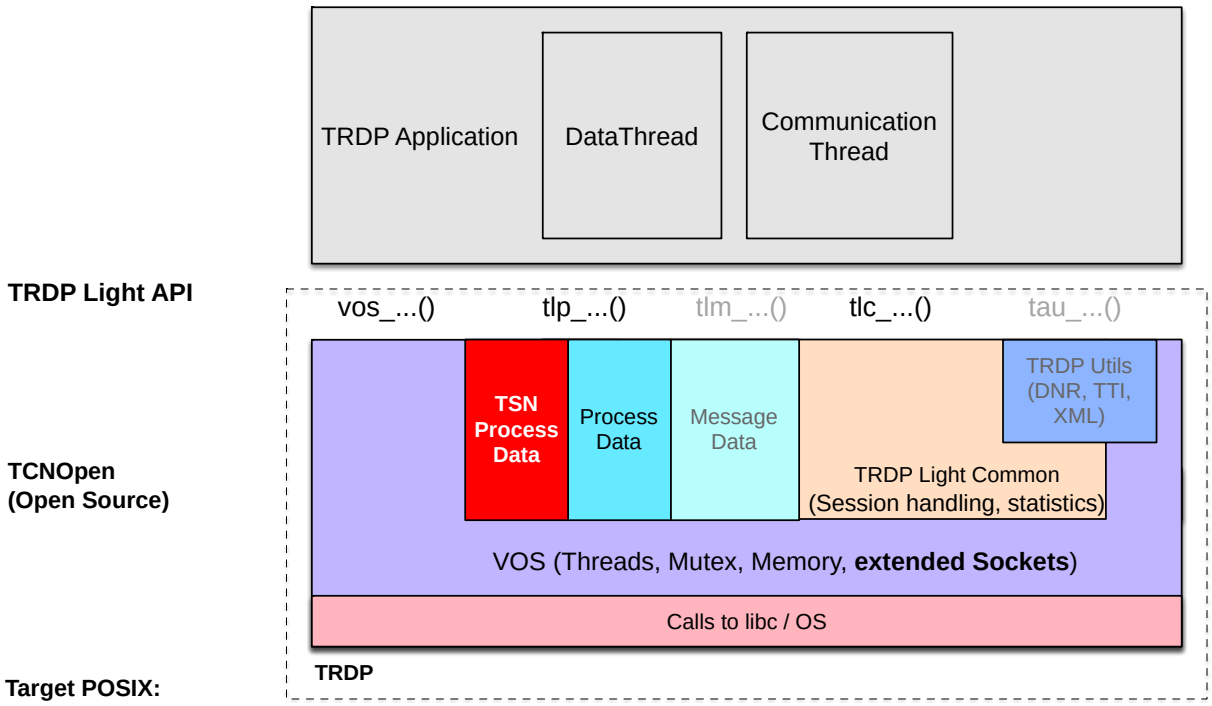


Figure 28: TRDP with TSN

The TSN Process Data in Figure 28 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 29).

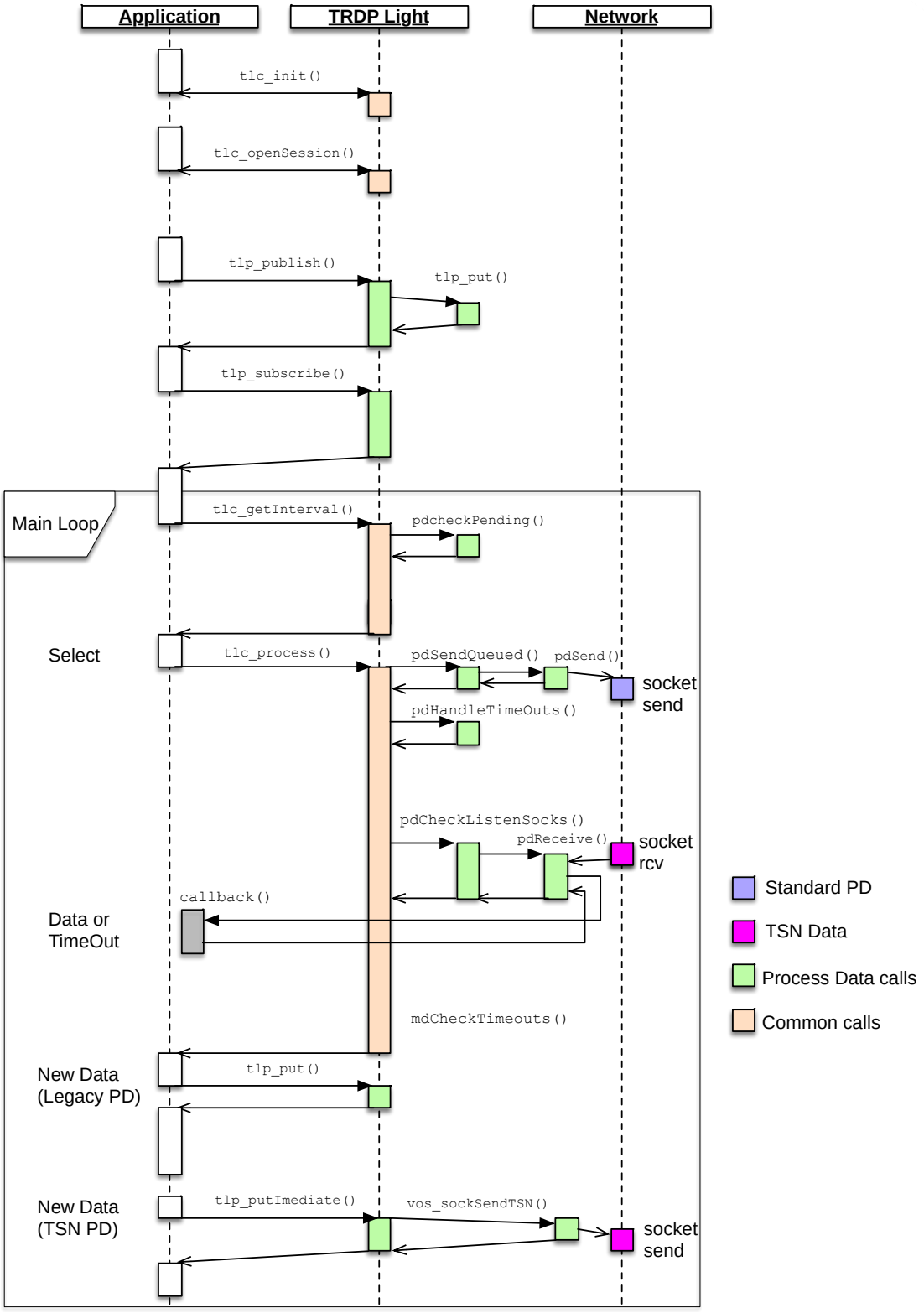


Figure 29: Sending TSN PD-PDU

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.

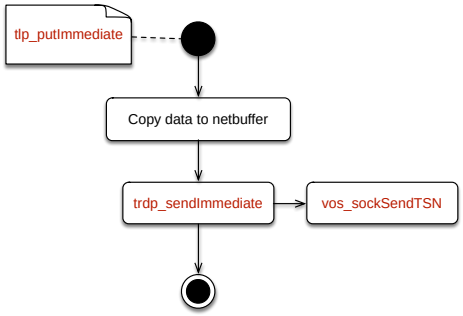


Figure 30: tlp_putImmediate

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 31.

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()`.

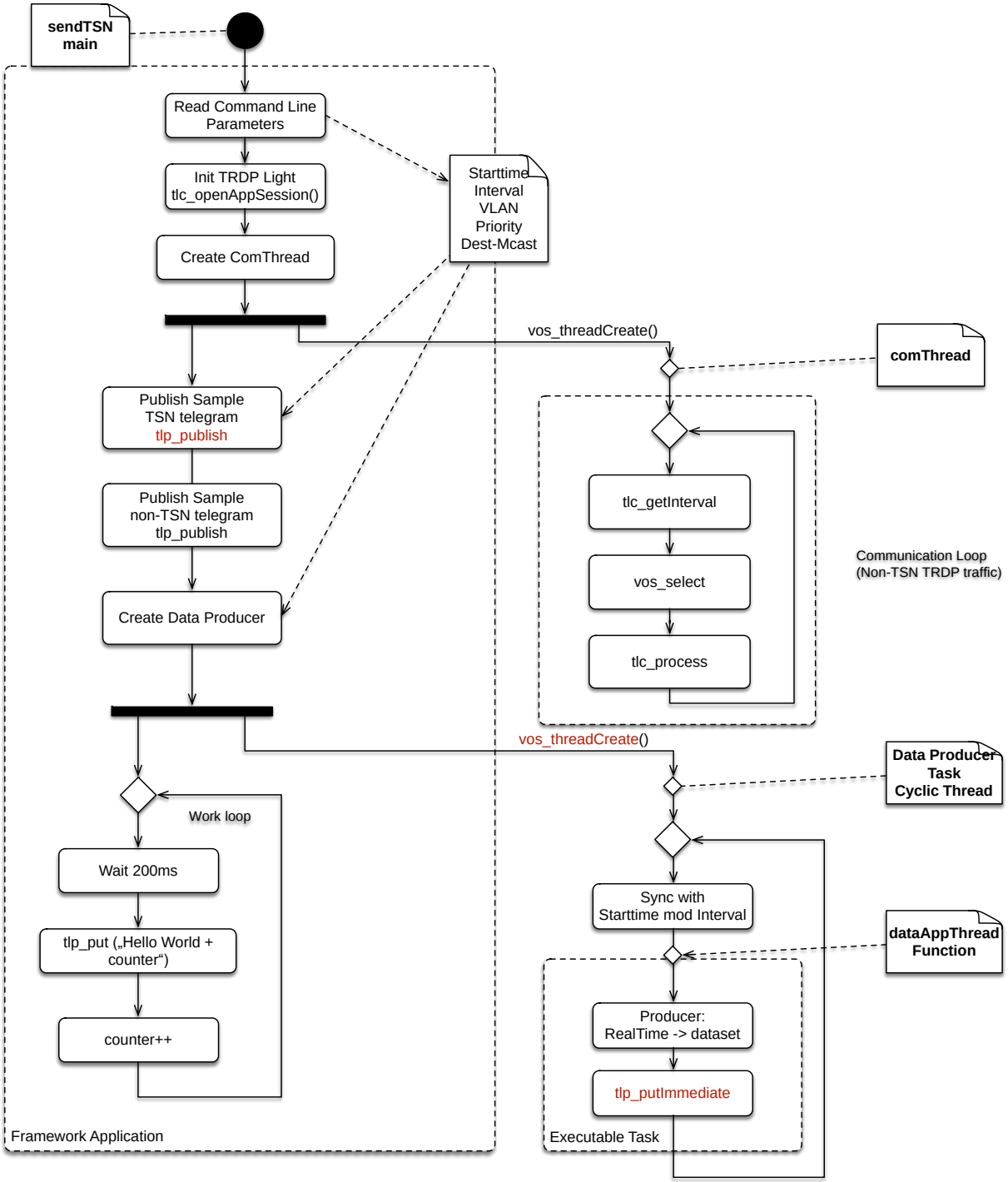


Figure 31: Activity diagram of sendTSN

Note: In Figure 31 and Figure 32, 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 32.

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…).

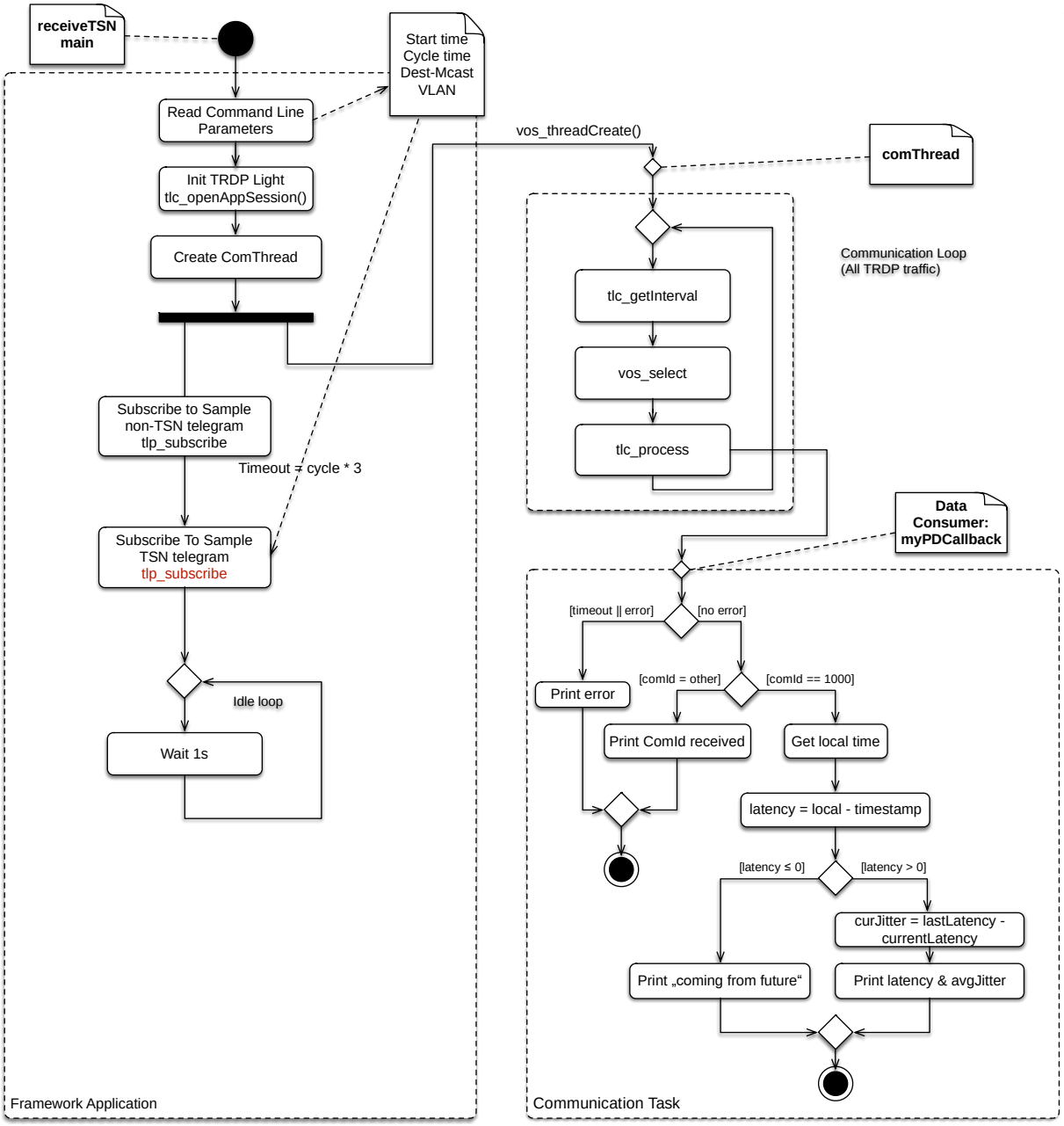


Figure 32: Activity diagram of receiveTSN

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.

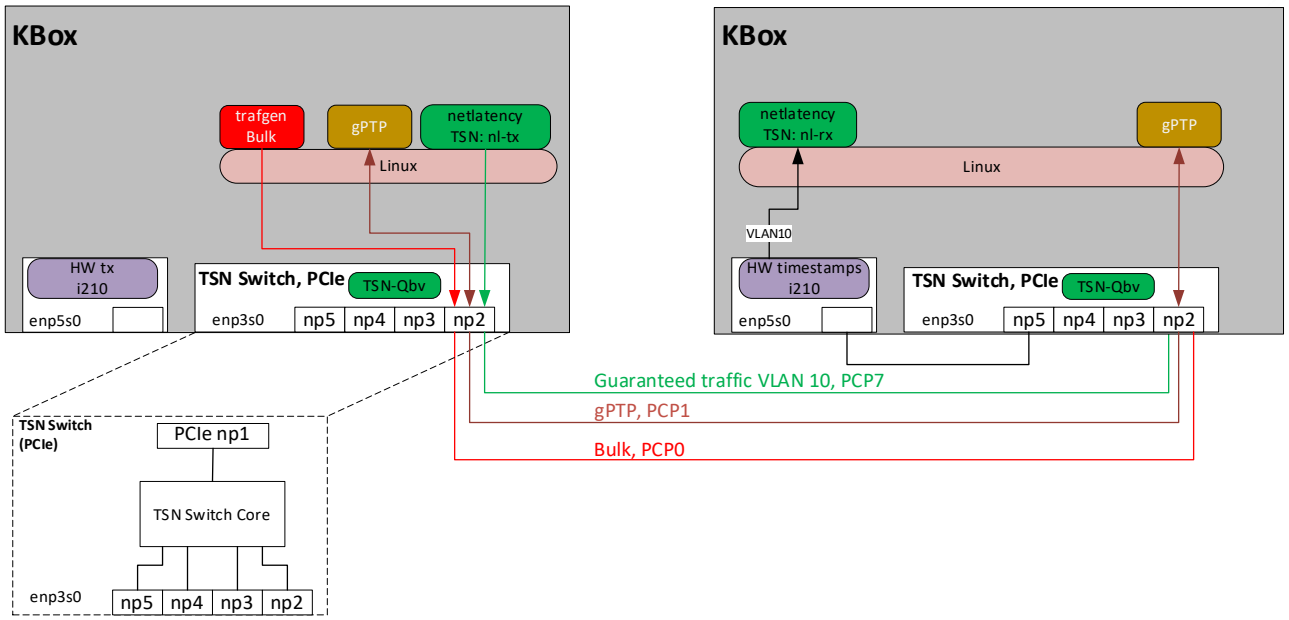


Figure 33: Recap of Set-up Task 1

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

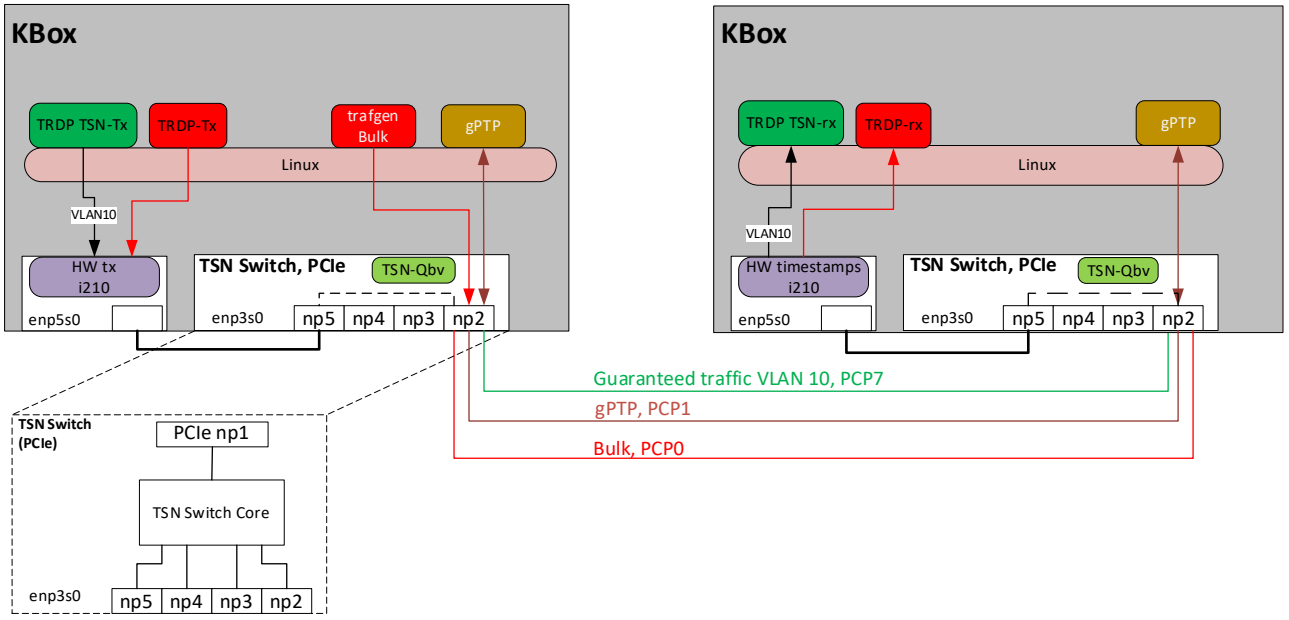


Figure 34: Test Set-up Task 2, bulk generated locally

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

Table 5: Terminology TRDP functions of Figure 34

4.4 Test Results

The tests for task 2 are planned to be similar to the tests performed inside task 1.

For each test configuration the following values will be reported in the subsequent phase of the OCORA initiative:

- TSN send latency and jitter (packet count/time)
- RT application latency (samples/time)
- TSN network jitter (packet count/time)

The following test configurations are planned:

- Best-effort TRDP-PD only
- TRDP-TSN only
- Best-effort TRDP-PD and TRDP-TSN in parallel
- TRDP-TSN with bulk traffic (best effort high volume)
- Best effort TRDP-PD with bulk traffic (best effort high volume)
- Best-effort TRDP-PD and TRDP-TSN with bulk traffic (best effort high volume)